Distributed RDF Query Processing and Reasoning in Peer-to-Peer Networks

PhD Thesis

Zoi Kaoudi

National and Kapodistrian University of Athens
Department of Informatics and Telecommunications

Athens
July 2011

Distributed RDF Query Processing and Reasoning in Peer-to-Peer Networks

Zoi Kaoudi

Ph.D. Thesis, Department of Informatics and Telecommunications
National and Kapodistrian University of Athens, July 2011

Abstract

With the interest in Semantic Web applications rising rapidly, the Resource Description Framework (RDF) and its accompanying vocabulary description language, RDF Schema (RDFS), have become one of the most widely used data models for representing and integrating structured information in the Web. RDF provides a simple and abstract knowledge representation for resources on the Web, while RDFS defines the terms that will be used in RDF statements and gives specific meaning to them. The Linked Data initiative, which aims at connecting data sources on the Web, has already become very popular and has exposed many datasets using RDF and RDFS. With the vast amount of available RDF data sources on the Web increasing rapidly, there is an urgent need for RDF data management. RDF storage, query processing and reasoning have been at the center of attention during the last years in the Semantic Web community and more recently in other research fields as well.

This thesis presents results that advance the state-of-the-art in the research area of distributed RDF query processing and reasoning in peer-to-peer (P2P) networks. We fully design and implement a P2P system, called Atlas, for the distributed query processing and reasoning of RDF and RDFS data. Atlas is built on top of distributed hash tables (DHTs), a popular case of P2P networks. The indexing scheme we deploy in our system is the triple indexing algorithm originally presented in RDFPeers where each RDF triple is indexed in the DHT three times. An important aspect of our indexing scheme is that data and schema information is handled uniformly. Although other distributed approaches assume that each node keeps all RDF schema information, we adopt a more generic approach where no global knowledge about the schema is required. In this way, our system can also handle scenarios with very big ontologies where other systems might not scale.

Initially, we study algorithms for distributed RDFS reasoning in peer-to-peer networks. We design and develop distributed *forward* and *backward* chaining algorithms on top of a DHT. Our backward chaining algorithm is the first distributed top-down

algorithm proposed for RDFS reasoning in a decentralized environment in general. Current forward chaining approaches in various distributed architectures demonstrate a big rate of redundant information occurred from the inferred RDF triples. Our forward chaining algorithm is the first one that deals with an important case of generating redundant RDF information. In addition, we present an algorithm which works in a bottom-up fashion using the magic sets transformation technique, a technique that has not been studied in the literature for distributed RDFS reasoning. We study theoretically the correctness of our reasoning algorithms and prove that they are sound and complete. We also provide a comparative study of our algorithms both analytically and experimentally. In the experimental part of our study, we obtain measurements in the realistic large-scale distributed environment of PlanetLab as well as in the more controlled environment of a local cluster.

Subsequently, we propose a query processing algorithm adapted to use a query graph model to represent SPARQL queries in order to avoid the computation of Cartesian products. SPARQL queries are answered in our system in a way that take into account both RDF data and RDFS ontologies, as well as the RDFS entailment rules. In addition, we show how to benefit from a mapping dictionary to further enhance the efficiency of the query processing algorithm. Although mapping dictionaries are by now standard in centralized RDF stores, our work is the first that discusses how to implement one in a DHT environment. Our experiments conducted in both PlanetLab and a local cluster showcase the importance of having a distributed mapping dictionary in our system.

Finally, we fully implement and evaluate a DHT-based optimizer. The goal of the optimizer is to minimize the time for answering a query as well as the bandwidth consumed during the query evaluation. We propose three greedy optimization algorithms for this purpose: two static and one dynamic. The static query optimization is completely executed before the query evaluation begins, while the dynamic query optimization take places during the query evaluation creating query plans incrementally. These algorithms use selectivity estimates to determine the chosen query plan. We propose methods for estimating the selectivity of RDF queries utilizing techniques from relational databases. We discuss which statistics should be kept at each network node and use histograms for summarizing data distributions. We demonstrate that it is sufficient for a node to create and maintain local statistics, i.e., statistics for its locally stored data. The evaluation of the query optimization algorithms and techniques is performed in a local cluster.

Acknowledgements

First of all, I would like to express my sincere thanks to my advisor, Prof. Manolis Koubarakis for his support and guidance throughout the years of my PhD studies. His enthusiasm and desire for perfection was an inspiration for me. I will always consider him a friend and a valuable source of advice. I would also like to thank the members of my doctoral committee, Prof. Alex Delis, Prof. Ioannis Ioannidis, Ass. Prof. Mema Roussopoulou, Prof. Timos Sellis, Prof. Vassilis Christophides and Lecturer Georgos Stamou for their comments and suggestions on improvements and extensions of this work.

I would also like to give many thanks to all my colleagues in the Department of Informatics and Telecommunications for their continuous support and help. Special thanks to my colleague and best friend, Iris, for her valuable help and support that kept me going to the end. Throughout this thesis I received financial support from European and national projects and I would like to thank all the people in the institutions for these research grants.

Special thanks go to my parents, Anna and Kostas, and my brother, Markos, for their endless love and encouragement throughout all my life endeavors. Their support in all my pursuits has been tremendous. I hope they are proud of me.

The good times spent with Iris and Matoula made this thesis an easier task. A big thanks for your indispensable friendship.

Words cannot always express the feelings for special persons that make your life meaningful. Persons that are a stand in difficult personal moments and a relief after long working hours. Spiros, a heartfelt thanks for your unconditional support and encouragement all these years.

To Spiros,
for his everlasting encouragement and support.

Contents

List of Tables

List of Figures

List of Abbreviations

RDF	Resource Description Framework
SPARQL	SPARQL Protocol And RDF Query Language
P2P	Peer-to-Peer
DHT	Distributed Hash Table
URI	Universal Resource Identifier
OWL	Web Ontology Language
SON	Semantic Overlay Network
FC	Forward Chaining
BC	Backward Chaining

LIST OF ABBREVIATIONS

Chapter 1

Introduction

With the emergence of the World Wide Web, a vast amount of information is now available online accommodating all aspects of human activities, knowledge and experiences. The original vision for the Web by its inventor Tim Berners-Lee [20] was to enable the publication and interconnection of documents through the Internet. With the number of documents increasing constantly over the years, search engines were invented to aid users to discover information that is relevant to their needs. Search engines crawl and index documents from the Web and provide keyword-based searching to users. Using sophisticated algorithms, they match the keywords given by the user with the text of the documents and return pointers to the documents that match better the given keywords. However, search engines do not take into consideration the meaning behind the text and the keywords, nor can answer more complex questions and return more useful information than just a pointer to a web page. For example, assume that a user wants to find out about the events happening in Chania during the period of her vacation. How can the user do that with a keyword search? How much effort does she have to spend browsing through websites until she finds an answer? Therefore, a serious problem facing search engines is that they often deliver results of high recall but low precision. This means that the results returned to a user contain a lot of relevant information but are mixed up with a lot of useless information as well. In this way, users are forced to manually browse through the returned results in order to find the right piece of information.

Apart from the constantly increasing number of documents on the Web, there is a current trend to publish *data* on the Web and make it easily discoverable, accessible, and available to people without any restriction. This Open Data movement has led to a plethora of data sources becoming available on the Web from various domains, from

government data to scientific datasets. Therefore, the Web is evolving to be not just a Web of documents but also a *Web of data*. In a Web of data infrastructure, data should be interlinked and integrated to enable the users to combine information from different data sources and extract composite knowledge. Today Web data integration is performed by the users themselves. An automatic process for the integration and interoperability of all available online data would simplify our everyday lives.

The vision of the *Semantic Web* has been articulated to offer solutions to the above problems [21]:

> *"The Semantic Web is an extension of the current web in which information is given well-defined meaning, better enabling computers and people to work in cooperation."*

The Semantic Web gives the opportunity to search engines to be able to answer effectively queries such as the ones we discussed above, and takes the burden off the users by automating many processes. The Semantic Web extends the current Web by introducing machine-readable data and metadata of the documents and information of how they are interconnected. It evolves the Web in order to accommodate intelligent, automatic processes that perform tasks on behalf of the users.

In order to realize the Semantic Web, a standardized means of representing structured and meaningful information on the Web has been envisioned by its advocates. Several data models have been proposed for representing such information on the Web. The most prominent one is the Resource Description Framework (RDF) [88]. RDF provides a simple and abstract knowledge representation for resources on the Web which are uniquely identified by Universal Resource Identifiers (URIs). Each statement in RDF can be encoded as an RDF triple. In addition, ontologies are used to give meaning to resources, group them into concepts and identify the relationships between these concepts. Two commonly used languages for encoding ontologies are RDFS and OWL. RDF Schema (RDFS) [28] is the vocabulary language of RDF. It defines the terms that will be used in RDF statements and gives specific meaning to them. Web Ontology Language (OWL) [60] can also be used for conceptualization and provides further expressivity in stating relationships among the resources. In order to be able to explore and query structured information expressed in RDF, a query language is necessary. During the past years, many query languages have been proposed for the RDF data model. Since January 2008, SPARQL has been the official W3C recommendation language for querying RDF data [119].

More than just a vision nowadays, the Semantic Web has begun to be realized by the

publication of large datasets according to the principles of the Linked Data initiative[1]. The Linked Data initiative aims at connecting data sources on the Web and exposing real life data using semantic technologies offering a new way of data integration and interoperability. The result of this effort is a Web of Data, where URIs identify real life things, dereferencing URIs returns RDF information about those things, and this RDF information contains related URIs which are links to other resources enabling further exploration. The Linked Data community have established a set of best practices for collaboratively publishing and interlinking structured data on the Web [10, 25]. There are numerous sources that expose their data on the Web in the form of Linked Data ranging from community-driven efforts to governmental bodies or scientific groups. DBpedia[2] [26], BBC music information [82], open government data[3] are only a few examples of the constantly increasing Linked Data cloud[4].

In addition to these efforts, on the 2nd of June 2011, the three main search engines, i.e., Google, Yahoo! and Bing, announced an initiative to create and support a common vocabulary that can be used by developers for marking up their web pages. In this way search engines can crawl and process structured meaningful data from the web pages. This initiative, called `schema.org`, provides a collection of open schemas for describing the content of the HTML web pages using Microdata format [59], a data model similar to RDF. Although this effort is still in its infancy, it is one of the many recent efforts towards realizing the Semantic Web vision. For a recent look as to where we stand regarding this vision, the interested reader might refer to the talk by Jim Hendler at ESWC 2011[5].

1.1 RDF Data Management

With the vast amount of available RDF data sources on the Web increasing rapidly, there is an urgent need for RDF data management. RDF storage, query processing and reasoning have been at the center of attention during the last years in the Semantic Web community and more recently in other research fields as well. Many systems have been developed for storing and querying RDF data. The first attempts were centralized approaches, such as Jena [168], Sesame [30] and RSSDB [7]. However, managing the avalanche of available RDF data has become a challenge for such RDF

[1] `http://linkeddata.org/`
[2] `http://dbpedia.org`
[3] `http://www.data.gov/`, `http://data.gov.uk/`
[4] `http://www4.wiwiss.fu-berlin.de/lodcloud/state/`
[5] `http://videolectures.net/eswc2011_hendler_work/`

stores. This has necessitated the careful performance evaluation of existing RDF stores on appropriately designed benchmarks and very big data sets [24, 38, 48, 131] and the development of novel implementations based on efficient indexing techniques and relational-style statistics-based query optimization [108, 109, 167]. Performance results published very recently indicate that state-of-the-art systems like RDF-3X [109] can execute complex join queries on RDF data sets containing close to a billion triples in a few seconds.

Although some existing RDF stores have excellent performance, they can be overwhelmed by user requests when used in wide-area network applications such as content-sharing, Web/Grid service registries, distributed digital libraries and social networks such as the ones discussed in [36, 75, 96, 129, 148, 158][6]. More generally, since centralized RDF stores are lacking the reliability properties typically associated with large distributed systems, (e.g., fault-tolerance, load balancing, availability) [23], researchers have also studied parallel and distributed solutions for RDF and RDFS query processing and reasoning. These include solutions based on peer-to-peer (P2P) systems, distributed computing platforms built on powerful clusters and, more recently, cloud computing platforms using the MapReduce framework [37]. In this thesis we concentrate on RDF query processing and reasoning using P2P networks.

In a setting where several heterogeneous sources of data are geographically distributed, P2P systems enable the aggregation and integration of these data sources in an efficient way. P2P networks have gained much attention in the last ten years, given all the good features they can provide to Internet-scale applications. There have been several proposals of P2P architectures and amongst them distributed hash tables (DHTs) [12] are the most prominent class. DHTs allow for *full distribution*, *high-performance*, *scalability*, *resilience to failures*, *robustness* and *adaptivity* in applications such as distributed digital libraries and others we mentioned above.

An RDF repository built on top of a DHT simplifies the integration of data from many distributed heterogeneous data sources compared to other distributed approaches. Additionally, DHTs can be used to ensure *efficient* query answering from all these heterogeneous sources. DHTs have been proposed for the storage and querying of RDF data at Internet scale by several works [3, 16, 33, 58, 89]. This thesis also focuses on DHTs as the P2P architecture of choice. Thus, the algorithms of this thesis can run on commodity machines deployed all over the world, as it is the case with many other P2P applications. This is in contrast to other distributed approaches that rely

[6] Although RDF is the chosen metadata model for resource annotation only in [36, 75], we can easily see how it could have been used in [96, 129, 148, 158].

on distributed platforms built on powerful clusters [40, 55, 112, 114] and cloud computing platforms using MapReduce [98, 163], which typically demand high-end, locally deployed infrastructures whose cost can be very high in many cases.

1.2 Challenges

When designing a DHT-based system for RDF data management, there are several challenges that have to be faced. The first one is how to distribute the data among the nodes of the network. DHTs utilize an efficient protocol for indexing data items in the network and thus, an indexing scheme for RDF data that conforms with this protocol may be adopted. Another issue that has to be faced is whether RDF data and RDFS ontologies should be handled uniformly or RDFS ontologies should be globally known by all nodes.

The adopted storage scheme will help us deal with the second challenge: answering SPARQL queries efficiently. A key aspect here is to design efficient query processing algorithms which are able to combine RDF data distributed across different nodes of the network to answer user queries. Another important need in Semantic Web applications is modeling application knowledge and reasoning about this knowledge. Therefore, in the context of RDF, we have to deal not only with a huge amount of distributed data, but also with a set of RDFS ontologies that give meaning to this data. Naturally, SPARQL queries need to be answered in a way that take into account both RDF data and RDFS ontologies, as well as the RDFS entailment rules given in [56].

Another challenging issue for RDF data management in a DHT environment is query optimization. RDF query optimization techniques in a DHT-based system have to be carefully considered given that data are distributed across all nodes of the network. Although query optimization has been extensively studied in the database area and is widely used in modern DBMSs, RDF query optimization has been addressed only recently even in centralized environments [108, 109, 145].

Therefore, the fundamental questions to which this thesis provides solutions can be summarized as follows:

- How can a DHT-based system index RDF data and RDFS ontologies and answer SPARQL queries efficiently?

- What kinds of query optimization techniques are useful in a DHT environment to speed up the query evaluation process?

- How can a DHT-based system provide reasoning mechanisms for the RDF and RDFS data model?

This thesis provides answers to the above questions and presents original results that advance the state-of-the-art in the research area of distributed RDF query processing and reasoning in P2P networks but also in decentralized environments in general. Our algorithms and techniques have been used to build the system Atlas. Atlas has been utilized in the European project OntoGrid where it has been used for realizing a Semantic Grid service registry [75, 87]. More recently, Atlas has also been extended with the capability to store spatial metadata expressed in European project SemsorGrid4Env [44].

1.3 Contributions

The contributions of this thesis are the following:

- In this thesis we fully design and implement a DHT-based system for the distributed query processing and reasoning of RDF and RDFS data. The indexing scheme we deploy in our system is the triple indexing algorithm originally presented in [33] where each RDF triple is indexed in the DHT three times. An important aspect of our indexing scheme is that data and schema information is handled uniformly. Although other distributed approaches such as [42, 163, 166] assume that each node keeps all RDF schema information, we adopt a more generic approach where no global knowledge about the schema is required. In this way, our system can also handle scenarios with very big ontologies where other systems such as the above might not scale.

- With respect to RDFS reasoning, our contribution is the design and development of distributed *forward* and *backward* chaining algorithms on top of a DHT. The forward chaining approach has minimal requirements during query answering, but needs a significant amount of storage for all the inferred data. In contrast, the backward chaining approach has minimal storage requirements, at the cost of an increase in query response time. There is a time-space trade-off between these two approaches [149], and only by knowing the query and update workload of an application, we can determine which approach would suit it better. This trade-off has never been studied in detail in a *distributed Internet-scale* scenario and this is a challenge we undertake. Our backward chaining algorithm is the first

distributed top-down algorithm proposed for RDFS reasoning in a decentralized environment in general. Current forward chaining approaches in various distributed architectures demonstrate a big rate of redundant information occurred from the inferred RDF triples [42, 163, 166]. Our forward chaining algorithm is the first one that deals with an important case of generating redundant RDF information. In addition, we present an algorithm which works in a bottom-up fashion using the magic sets transformation technique [17], a technique that has not been studied in the literature for distributed RDFS reasoning. We study theoretically the correctness of our reasoning algorithms and prove that they are sound and complete. We also provide a comparative study of our algorithms both analytically and experimentally. In the experimental part of our study, we obtain measurements in the realistic large-scale distributed environment of PlanetLab as well as in the more controlled environment of a local cluster.

- We propose a query processing algorithm adapted to use a query graph model to represent SPARQL queries in order to avoid the computation of Cartesian products. In addition, as URIs and literals may consist of long strings that are transferred in the network and processed locally at the nodes, we show how to benefit from a mapping dictionary to further enhance the efficiency of the query processing algorithm. Although mapping dictionaries are by now standard in centralized RDF stores, our work is the first that discusses how to implement one in a DHT environment. Our experiments conducted in both PlanetLab and a local cluster showcase the importance of having a distributed mapping dictionary in our system.

- In the context of query optimization, we fully implement and evaluate a DHT-based optimizer. The goal of the optimizer is to minimize the time for answering a query as well as the bandwidth consumed during the query evaluation. We propose three greedy optimization algorithms for this purpose: two static and one dynamic. The static query optimization is completely executed before the query evaluation begins, while the dynamic query optimization take places during the query evaluation creating query plans incrementally. These algorithms use selectivity estimates to determine the chosen query plan. We propose methods for estimating the selectivity of RDF queries utilizing techniques from relational databases. We discuss which statistics should be kept at each network node and use histograms for summarizing data distributions. We demonstrate that it is sufficient for a node to create and maintain local statistics, i.e., statistics for its

locally stored data. These local statistics are in fact global statistics needed by the optimization algorithms and can be obtained by other nodes by sending low cost messages. This is a very good property of the indexing scheme we adopt from [32] that has not been pointed out in the literature before.

- Using the above techniques, we have implemented a P2P system, called Atlas, for distributed query processing and reasoning of RDF and RDFS data. Atlas is publicly available as open source under the LGPL license[7]. Although our proposed algorithms and techniques have been implemented in Atlas using the Bamboo DHT [122], they are DHT-agnostic; they can be implemented on top of any DHT.

The results of this thesis have been published in major international conferences [72, 74] of the Semantic Web community, workshops [70, 76] and journals [71, 75] or are under review in major international journals [69].

1.4 Thesis Structure

The remainder of the thesis is structured as follows. Chapter 2 sets the foundations on technologies that enable us to achieve our goals and surveys related work in the areas of RDF data management in P2P networks, RDFS reasoning and RDF query optimization. In Chapter 3 we present concepts, terminology and theoretical results that are used throughout the thesis. Chapter 4 presents Atlas, a system that is able to support full-fledged management of RDF data in a large-scale decentralized environment and some basic protocols that are used in our algorithms described in the next chapters. In Chapter 5 we design and evaluate distributed RDFS reasoning techniques in a DHT environment. Chapter 6 presents a complete query evaluation algorithm which is used to answer arbitrary queries for RDF and describes a distributed mapping dictionary that increases the system's performance. In Chapter 7 we present the query optimization techniques we have developed in our system to increase its performance. Finally, Chapter 8 summarizes the achievements of this thesis and indicates directions for future research.

[7] `http://atlas.di.uoa.gr`

Chapter 2

Background and Related Work

In this chapter we set the foundations on technologies that enable us to achieve our goals and we survey related work in the areas most relevant to the thesis. We first introduce the RDF and RDFS data model and the query language SPARQL. Then, we present a survey of P2P networks emphasizing on structured overlays. In the related work, we start with P2P database systems, and especially elaborate on PIER [63] which is closely related with our work. Additionally, we present proposals for managing RDF data in P2P networks. Then, we survey related work in the area of RDFS reasoning in centralized environments as well as distributed ones of various architectures. Finally, we present optimization techniques that have been proposed recently for RDF query optimization both in centralized and distributed settings.

2.1 Resource Description Framework

In this section we introduce the Resource Description Framework (RDF) together with its accompanying vocabulary RDFS. In addition, we present the corresponding standardized query language, called SPARQL (SPARQL Protocol And RDF Query Language), for querying RDF and RDFS data.

2.1.1 RDF

The Resource Description Framework (RDF) [88] is a framework for representing information about Web resources. It consists of W3C[1] recommendations that enable the encoding, exchange and reuse of structured data, providing means for publishing both human-readable and machine-processable vocabularies. Nowadays the current W3C

[1] `http://www.w3.org/`

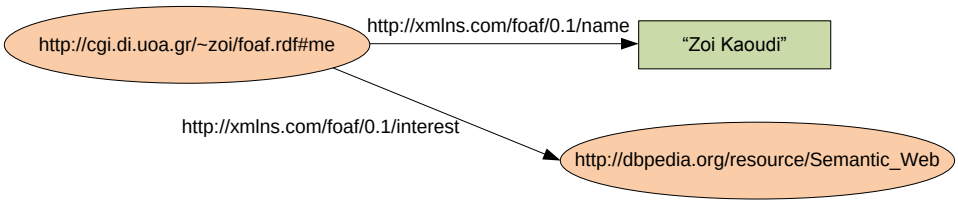


Figure 2.1: An RDF graph

recommendations for RDF are used in a variety of application areas. The Linked Data initiative[2], which aims at connecting data sources on the Web, has already become very popular and has exposed many datasets using RDF and RDFS. DBpedia[3], BBC music information [82], government datasets[4] are only a few examples of the constantly increasing Linked Data cloud[5].

The RDF data model offers the following basic concepts:

- *Resources*: In RDF, a resource is anything that we want to describe in the World Wide Web. A resource may be a Web page, a book, an author, a paper or a computer file. Every resource is uniquely identified by a *Universal Resource Identifier* (URI) [19].

- *Properties*: A property is a characteristic of a resource. For example, "hasName" may be used for describing the name of an author. Properties are also identified by URIs.

- *Literals*: Literals are constant values of any property. For example, "Zoi Kaoudi" may be the value of property "hasName".

- *Statements*: Statements are the constructs offered by RDF for representing information about a domain. A statement has three parts: the resource the statement is about, the property of the resource the statement refers to, and the value of that property. The three parts of a statement are named, respectively, *subject*, *predicate* and *object*. The object of a statement can be another resource or a literal.

Two possible representations of RDF data are *labeled graphs* or *triple sets*. In the first one, a resource of a literal is depicted as a *node* and a property as an *arc*. In

[2] `http://linkeddata.org/`
[3] `http://dbpedia.org`
[4] `http://www.data.gov/`, `http://data.gov.uk/`
[5] `http://www4.wiwiss.fu-berlin.de/lodcloud/state/`

the *triple sets* representation, all statements are represented as "triples" in the form *subject - predicate - object* (resource, property, value). Figure 2.1 shows a simple RDF graph which states that the resource `http://cgi.di.uoa.gr/~zoi/foaf.rdf#me` has name `"Zoi Kaoudi"` and has interest in a resource which is expressed by the URI `http://dbpedia.org/resource/Semantic_Web`. We depict a resource with an oval circle and a literal with a rectangle. The labeled arcs represent the properties of the RDF graph.

To avoid writing long strings of URIs, *namespaces* come into use. Namespaces substitute the first part of a URI with a short prefix. Then, the full URI is abbreviated by the prefix appended with the last part of the URI. For example, for the property `http://xmlns.com/foaf/0.1/name`, the URI `http://xmlns.com/foaf/0.1/` can be abbreviated by the prefix `foaf` and then the property can be written as `foaf:name`.

The same knowledge represented as a graph in Figure 2.1 can be represented by a set of two triples. Using the namespace `foaf` as defined above and the namespace `dbpedia` to abbreviate the URI `http://dbpedia.org/resource/` these triples can be written as follows:

```
http://cgi.di.uoa.gr/~zoi/foaf.rdf#me foaf:name "Zoi Kaoudi" .
http://cgi.di.uoa.gr/~zoi/foaf.rdf#me foaf:interest dbpedia:Semantic_Web .
```

Another feature used in RDF is *blank nodes*. Blank nodes are anonymous resources which are not expressed by a URI but in the form of `_:bnodeID`. The purpose of blank nodes is twofold. First, they can be used to encode n-ary relationships. Another purpose of blank nodes is to make statements about resources that might not have URIs but can be described in terms of their relationship with other resources. For example, imagine that we want to encode the relationship between a resource which represents a person and the address of this person which is composed by the street name, postal code and city. Then, we can use a blank node to represent the notion of the address. This information can be expressed in RDF triples as follows:

```
http://cgi.di.uoa.gr/~zoi/foaf.rdf#me foaf:name "Zoi Kaoudi" .
http://cgi.di.uoa.gr/~zoi/foaf.rdf#me foaf:address :_zoiaddress .
:_zoiaddress foaf:street "Panepistimiopolis" .
:_zoiaddress foaf:postalCode "15784" .
:_zoiaddress foaf:city "Athens" .
```

In this case, `:_zoiaddress` denotes the blank node. Blank node identifiers are used to differentiate one by another and may only appear as subjects or objects in triples but not as predicates.

Although the conceptual model of RDF is a graph, RDF provides an XML syntax, called *RDF/XML*, for writing and exchanging RDF graphs. For example, the RDF graph of Figure 2.1 can be encoded in RDF/XML as follows:

```
<?xml version="1.0"?>
<rdf:RDF
  xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
  xmlns:foaf="http://xmlns.com/foaf/0.1/"
  <rdf:Description rdf:about="http://cgi.di.uoa.gr/~zoi/foaf.rdf#me">
  <foaf:name>Zoi Kaoudi</foaf:name>
  <foaf:interest rdf:resource="http://dbpedia.org/resource/Semantic_Web"/>
<\rdf:RDF>
```

Apart from the RDF/XML serialization, there are other formats commonly used such as N3[6], N-Triples[7] and Turtle[8].

2.1.2 RDF Schema (RDFS)

The RDF data model offers a simple way for describing interrelationships among resources in terms of named properties and values, but does not provide mechanisms for declaring these properties, nor does it provide any ways for defining the relationships between these properties and other resources. This is the role of RDF Schema (RDFS) [27]. RDFS is like a vocabulary of RDF; it provides the means to a user to define terms that will be used in RDF statements and give specific meaning to them. RDFS defines not only the properties of the resource (e.g., title, author, subject etc.) but also the kinds of resources being described (people, paper, Web pages, books etc.). Resources having similar characteristics can be divided into groups called RDFS *classes*. When we want to refer to both RDF and RDFS information we will use the term RDF(S).

Figure 2.2 shows an RDF(S) graph. The upper part of the graph is the RDFS information, while the lower part depicts the RDF data. The namespaces used are shown in the lower left of the figure. The RDFS of this example defines the relationships between the classes. The property `rdfs:subClassOf` is a predefined property in RDFS and denotes a specialization relationship between two classes. For example, `ex:GraduateStudent` is a specialized type of `ex:Student`. Another predefined property in RDF commonly used is `rdf:type`. This property defines the members of a certain class. For example, resource `http://cgi.di.uoa.gr/~zoi/foaf.rdf#me`

[6] `http://www.w3.org/DesignIssues/Notation3`
[7] `http://www.w3.org/TR/rdf-testcases/#ntriples`
[8] `http://www.w3.org/TeamSubmission/turtle/`

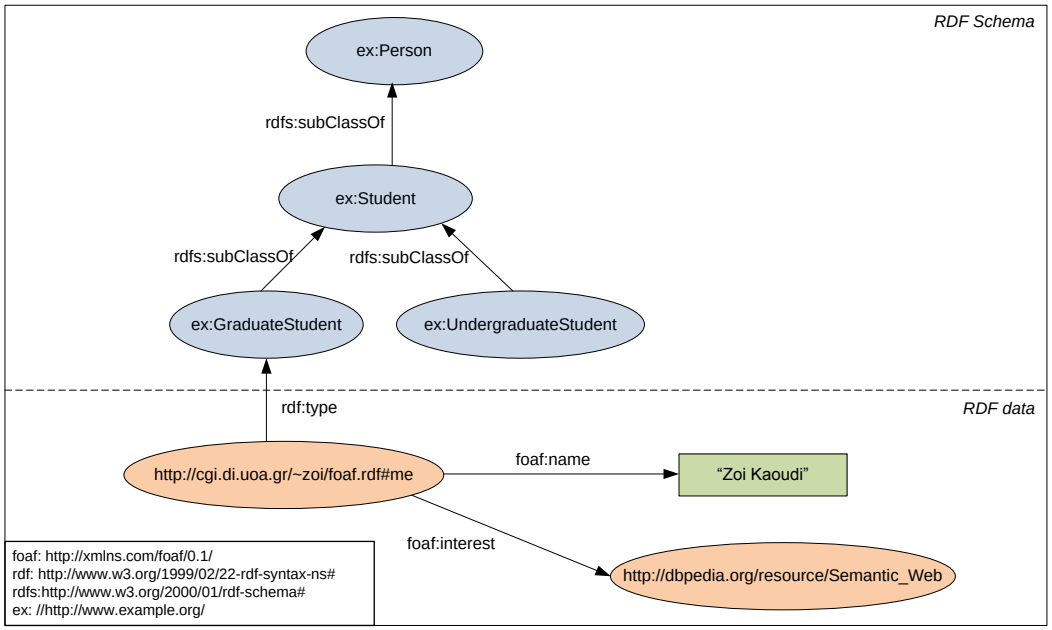


Figure 2.2: An RDF(S) graph

is a member of class `ex:GraduateStudent`. Then, if this class denotes all graduate students, resource `http://cgi.di.uoa.gr/~zoi/foaf.rdf#me` is a graduate student. The members of a class are also known as *instances* of the class. Namespaces `rdf` and `rdfs` are the namespaces of the core RDF and RDFS vocabulary defined by the URIs `http://www.w3.org/1999/02/22-rdf-syntax-ns#` and `http://www.w3.org/2000/01/rdf-schema#` respectively and will be used throughout the thesis.

Another feature of RDFS is that it also provides vocabulary for describing how properties and classes are intended to be connected in an RDF(S) graph. The most important information of this kind is supplied by using the RDF Schema properties `rdfs:domain` and `rdfs:range`. The `rdfs:domain` property is used to indicate that a particular property applies to a specific class. The `rdfs:range` property is used to indicate that the values of a particular property are instances of a specific class or are types of specific literals. For example, in the RDFS of Figure 2.2 we could add the following triples for properties `foaf:name` and `foaf:interest`:

```
foaf:name rdfs:domain ex:Person .
foaf:interest rdfs:domain ex:Student .
foaf:name rdfs:range string .
```

This means that the property `foaf:name` applies to instances of class `ex:Person`,

while the property `foaf:interest` applies to instances of class `Student`. In addition, the last triple states that property `foaf:name` takes values that are strings.

The most important functionality of RDFS is the ability to make inferences using the RDFS entailment rules [56]. For example, the RDFS property `rdfs:subClassOf` is defined as a transitive property in the $rdfs11$ rule of the RDFS Semantics [56]:

$$\frac{\texttt{uuu rdfs:subClassOf vvv}, \texttt{uuu rdfs:subClassOf xxx}}{\texttt{vvv rdfs:subClassOf xxx}}$$

This rules means that if class `uuu` is a subclass of class `vvv` and class `vvv` is a subclass of class `xxx`, then we infer that class `uuu` is a subclass of class `xxx`. Thus, in the example of Figure 2.2, we can infer that class `ex:GraduateStudent` is a subclass of class `ex:Person`. Then, using this inferred triple and some other rule of [56], we can derive extra knowledge from this RDFS graph. For example, rule $rdfs9$ states the following:

$$\frac{\texttt{uuu rdfs:subClassOf xxx}, \texttt{vvv rdf:type uuu}}{\texttt{vvv rdf:type xxx}}$$

Therefore, we can infer that resource `http://cgi.di.uoa.gr/~zoi/foaf.rdf#me` in Figure 2.2 is also an instance of class `ex:Person`. The complete set of the RDFS entailment rules can be found in [56]. In Chapter 3, we define the complete set of rules we use in our work.

2.1.3 SPARQL

During the past years, many query languages have been proposed for the RDF data model. Some of them include RQL [81], RDQL [133], SeRQL [29], and TRIPLE [141]. On 15 January 2008, SPARQL [119] became the official W3C recommendation language for querying RDF data.

SPARQL [119] is a query language for RDF graphs and has the ability to extract information about both the data and the schema. SPARQL is based on the concept of matching graph patterns. The simplest graph patterns are *triple patterns*, which are like an RDF triple but with the possibility of a variable in any of the subject, predicate or object positions. A query that contains a conjunction of triple patterns is called *basic graph pattern*. A basic graph pattern matches a subgraph of the RDF data when RDF terms from the subgraph may be substituted with the variables of the graph pattern.

The syntax of SPARQL follows an SQL-like `select-from-where` paradigm. The `select` clause specifies the variables that should appear in the query results. Each variable in SPARQL is prefixed with character "?". The `from` clause specifies the RDF(S) graph that should be used for answering the query. If not used, the query runs over the whole RDF(S) triples stored in the system. The graph patterns of the query are set in the `where` clause. The following SPARQL query asks for all resources which have a name and an interest.

Listing 2.1: A simple SPARQL query

```
PREFIX foaf: <http://xmlns.com/foaf/0.1/>
SELECT *
WHERE {
    ?x foaf:name ?y .
    ?x foaf:interest ?z
}
```

Keyword `PREFIX` defines the namespaces used in the `where` clause of the query. The answer of a SPARQL query with a `select` clause is variable bindings. If the above SPARQL query is posed over the RDF graph of Figure 2.1, the variable bindings of the answer would be:

?x	?y	?z
http://cgi.di.uoa.gr/∼zoi/foaf.rdf#me	"Zoi Kaoudi"	http://dbpedia.org/resource/Semantic_Web

SPARQL also gives the capability for inserting constraints to the variables included in the graph pattern using the `FILTER` keyword. In this way, answers are restricted to those which result in evaluating the filter expression to true. For example, if we want to ask for the names of those resources which have an interest in the Semantic Web, we can write the following query:

Listing 2.2: A simple SPARQL query with a filter expression

```
PREFIX foaf: <http://xmlns.com/foaf/0.1/>
SELECT ?y
WHERE {
    ?x foaf:name ?y .
    ?x foaf:interest ?z .
    FILTER (?z=http://dbpedia.org/resource/Semantic_Web)
}
```

The answer to this query according to the RDF graph of Figure 2.1 would be {`?y="Zoi Kaoudi"`}. Constraints in filter expressions may include mapping resources to vari-

ables such as the above example, as well as arithmetic constraints to literals or string matching using regular expressions.

Another feature of SPARQL is the *optional* functionality. This functionality provides us with the ability to have patterns that might not match for every part in an RDF graph. In this case, the optional part creates no bindings but does not eliminate the entire solution. Keyword `OPTIONAL` is used with a graph pattern such as the ones in the `WHERE` clause.

For example, assume that we have an RDF graph composed by the following triples:

```
http://cgi.di.uoa.gr/~zoi/foaf.rdf#me foaf:name "Zoi Kaoudi" .
http://cgi.di.uoa.gr/~zoi/foaf.rdf#me foaf:interest dbpedia:Semantic_Web .
http://www.w3.org/People/Berners-Lee/card foaf:name "Tim Berners-Lee" .
```

The query displayed in Listing 2.3 below asks for the name of all resources and optionally for their interest, it there exists one. If it does not, we still want to retrieve their name.

Listing 2.3: A simple SPARQL query with optional

```
PREFIX foaf: <http://xmlns.com/foaf/0.1/>
SELECT ?y ?z
WHERE {
    ?x foaf:name ?y .
    OPTIONAL {
        ?x foaf:interest ?z .
    }
}
```

The answer to this query would be the following variable bindings:

?y	?z
"Zoi Kaoudi"	http://dbpedia.org/resource/Semantic_Web
"Tim Berners-Lee"	

The result set of a query varies according to the main query clause. Apart from the `SELECT` clause, SPARQL also allows `CONSTRUCT, DESCRIBE` and `ASK` clauses. In the case of a `SELECT` query, the result set is a set of variables and their possible bindings. On the other hand, if we have a `CONSTRUCT` query, the result is an RDF graph constructed by substituting variables in a set of triple templates. Finally, a `DESCRIBE` query returns an RDF graph that describes the resources found and an `ASK` query returns yes or no depending on whether a query pattern matches or not.

At this point, we should note that the current recommendation of SPARQL has a lot of shortcomings. For instance, it does not support aggregates, updates, and more importantly, it does not deal with the RDFS entailment. Therefore, RDFS entailment depends on the implementation of each system. The next version of SPARQL, i.e., SPARQL 1.1, will introduce several of these issues. One of these is RDFS regimes which define the evaluation of basic graph patterns using semantic entailment relations instead of explicitly given RDF graphs. This effort is still a W3C working group and will be soon a W3C recommendation. Apart from the W3C working group, several extensions for SRARQL have been proposed in the literature that capture the RDFS entailment semantics. One of them is nSPARQL, a navigational language for RDF which provides the expressivity of RDFS semantics as well [11, 115].

2.2 Peer-to-Peer Systems

In P2P systems a very large number of autonomous computing nodes (the *peers*) pool together their resources and rely on each other for *data* and *services*. P2P networks have emerged as a natural way to share data. Popular systems such as Napster[9] (now in a commercial service), Gnutella[10], Freenet[11], Kazaa[12], Morpheus[13] and others have made this model of interaction popular. Ideas from P2P computing can also be applied to other distributed applications beyond data sharing such as Grid computation (e.g., SETIHome[14] or DataSynapse[15]), collaboration networks (e.g., Groove[16]) and even new ways to design Internet infrastructure that supports sophisticated patterns of communication and mobility [146].

P2P networks are typically distinguished into three different classes according to their topology: unstructured networks, structured networks and hierarchical networks. In unstructured networks all peers are equal and form an overlay network with no restrictions on topology and no centralized source of information. On the contrary, structured networks have a regular topology, e.g., rings or hypercubes, and were devised as a remedy for the routing and object location inefficiencies of unstructured

[9] `http://www.napster.com`
[10] There are various clients implementing the Gnutella protocol or variations. See for example, `http://www.limewire.com`.
[11] `http://freenet.sourceforge.net`
[12] `http://www.kazaa.com`
[13] `http://www.musiccity.com`
[14] `http://www.setiathome.ssl.berkeley.edu`
[15] `http://www.datasynapse.com`
[16] `http://www.groove.net`

networks. Distributed hash tables (DHTs) are a prominent class of structured overlays that attempt to solve the object lookup problem by offering some form of distributed hash table functionality: assuming that data items can be identified using unique numeric keys, DHT nodes cooperate to store keys for each other. Finally, hierarchical networks partition the nodes into two sets: super-peers and clients. In a super-peer system, all super-peers are equal and have the same responsibilities. Each super-peer serves a fraction of the clients and keeps indices on the resources of those clients. In the rest of this section, we briefly survey some well-known DHTs which constitute our chosen paradigm.

2.2.1 Distributed hash tables

The success of P2P protocols and applications such as Napster and Gnutella motivated researchers from the distributed systems, networking and database communities to look more closely into the core mechanisms of these systems and investigate how these could be supported in a principled way. This quest gave rise to a new wave of distributed protocols, collectively called *distributed hash tables* (DHTs), that were aimed primarily at the development of P2P applications [2, 8, 95, 121, 125, 147, 169]. DHTs are *structured* P2P systems. DHTs attempt to solve the following *lookup problem*:

> Let X be some data item stored at some distributed dynamic network of nodes. Find data item X.

The core idea in all DHTs is to solve this lookup problem by offering some form of distributed hash table functionality: assuming that data items can be identified using *unique numeric keys*, DHT nodes cooperate to store keys in a way that can be easily found. Although the DHTs available in the literature differ in their technical details, all of them address the following central questions:

- *How do we map keys to nodes?* Keys and nodes are identified by a binary number. Keys are stored at one or more nodes with identifiers "close" to the key identifier in the identifier space.

- *How do we route queries for keys?* Any node that receives a query for key k, returns the data item X associated with k if it owns k, otherwise it forwards k to a node with identifier "closer" to k using only local information.

- *How do we deal with dynamicity?* DHTs are able to adapt to node joins, leaves and failures and update routing tables with little effort.

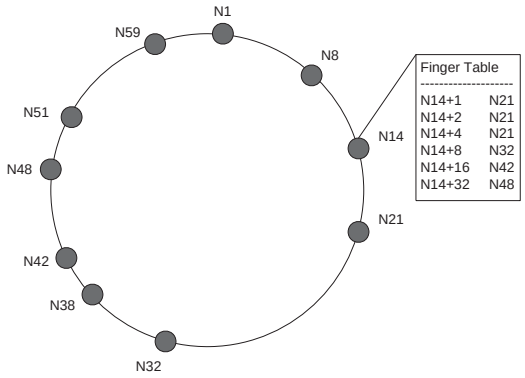


Figure 2.3: An example of a identifier circle with m=6 and 10 nodes

The answers to the above questions can give us a good high-level categorization of existing DHTs [9, 12]. Most implementations of DHTs offer a very simple interface consisting of two functions:

- Put($id, item$): this operation inserts a data item with key id and value $item$ in the DHT.
- Get(id): this operation retrieves the data item with key id from the DHT.

Both functions rely on the Lookup(id) operation, which returns a pointer to the DHT node that is responsible for the key id.

In the rest of this section we focus on Chord [147] and Pastry [125], two early and influential DHTs. Finally, we present the Bamboo DHT [122] a Pastry-based DHT which manages to handle churn more efficiently. Bamboo is our chosen DHT for the implementation of our system. Other DHTs include CAN [121], P-Grid [1, 2], Tapestry [169], Kademlia [97], Viceroy [95] and DKS [8]. The current study of P2P systems and DHTs represents only some of the systems in the area. More detailed surveys include [12, 92, 99, 159].

Chord

Chord uses a variation of *consistent hashing* [77] to map keys to nodes. In the consistent hashing scheme, each node and data item is assigned a m-bit identifier where m should be large enough to avoid the possibility of different items hashing to the same identifier (a cryptographic hashing function such as SHA-1 can be used). The identifier of a node can be computed by hashing its IP address. For data items, we first have to decide a key and then hash this key to obtain an identifier. For example, in a file-sharing

application the name of the file can be the key (this is an application-specific decision). Identifiers are ordered in an *identifier circle (ring)* module 2^m i.e., from 0 to $2^m - 1$. Figure 2.3 shows an example of an identifier circle with 64 identifiers ($m = 6$) but only 10 nodes.

Keys are mapped to nodes in the identifier circle as follows. Let H be the consistent hash function used. Key k is assigned to the the first node which is equal or follows $H(k)$ in the identifier space. For example, in the network shown in Figure 2.3, a key with identifier 8 would be stored at node $N8$. In other words, key k is assigned to the node whose identifier is the first identifier *clockwise* in the identifier circle starting from $H(k)$. This node is called the *successor* node of identifier $H(k)$ and is denoted by $successor(H(k))$. We will often say that this node is *responsible* for key k. In our example, node $N32$ would be responsible for all keys with identifiers in the interval $(21, 32]$.

If each node knows its successor, a query for locating the node responsible for a key k can always be answered in $O(N)$ steps where N is the number of nodes in the network. To improve this bound, Chord maintains at each node a routing table, called the *finger table*, with at most m entries. Each entry i in the finger table of node n, points to the first node s on the identifier circle that succeeds identifier $H(n) + 2^{i-1}$. These nodes (i.e., $successor(H(n) + 2^{i-1})$ for $1 \leq i \leq m$) are called the *fingers* of node n. Since fingers point at repeatedly doubling distances away from n, they can speed-up search for locating the node responsible for a key k. If the finger tables have size $O(\log N)$, then finding a successor of a node n can be done in $O(\log N)$ steps with high probability [147].

To simplify joins and leaves, each node n maintains a pointer to its *predecessor* node i.e., the first node *counter-clockwise* in the identifier circle starting from n. When a node n wants to join a Chord network, it finds a node n' that is already in the network using some out-of-band means, and then asks n' to help n find its position in the network by discovering n's successor [147]. Every node runs a *stabilization* algorithm periodically to learn about nodes that have recently joined the network. When n runs the stabilization algorithm, it asks its successor for the successor's predecessor p. If p has recently joined the network then it might end-up becoming n's successor. Each node n periodically runs two additional algorithms to check that its finger table and predecessor pointer is correct [147]. Stabilization operations may affect queries by rendering them slower (because successor pointers are correct but finger table entries are inaccurate) or even incorrect (when successor pointers are inaccurate). However, assuming that successor pointers are correct and the time it takes to correct finger tables is less than the time it

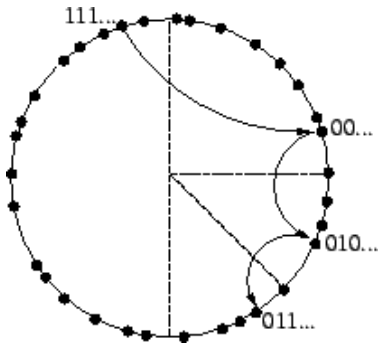


Figure 2.4: Pastry lookup operation [122]

takes for the network to double in size, one can prove that queries can still be answered correctly in $O(\log N)$ steps with high probability [147].

To deal with node failures and increase robustness, each Chord node n maintains a *successor list* of size r which contains n's first r successors. This list is used when the successor of n has failed. In practice, even small values of r are enough to achieve robustness [147]. If a node chooses to leave a Chord network voluntarily then it can inform its successor and predecessor so they can modify their pointers and, additionally, it can transfer its keys to its successor. It can be shown that with high probability, any node joining or leaving a Chord network can use $O(\log^2 N)$ messages to make all successor pointers, predecessor pointers and finger tables correct [147].

Pastry

Nodes and data items in Pastry [125] are also mapped to a circular identifier space modulo 2^m using a hash function such as SHA-1. A node maintains local state in three data structures: the *leaf set*, the *routing table* and the *neighbourhood set*. The leaf set of a node n is the set of $2r$ nodes immediately preceding and following n in the circular identifier space. We will use $L = \{L_i, -r \leq i \leq r\}$ to denote this set where L_0 is node n. Pastry's leaf set corresponds to Chord's set of successors. As in Chord, correctness of the leaf set is enough to guarantee correct routing in Pastry (within $O(N)$ steps where N be the number of nodes in a network).

Let us assume that node identifiers are treated as sequences of digits in base 2^b. The routing table of a node n in Pastry is organized as a two-dimensional table R with $\log_{2b} N$ rows and $2^b - 1$ columns. Let $R[l, i]$ denote the element of the routing table in row l and column i. Node n chooses entry $R[l, i]$ in its routing table to be a node with an identifier that matches n's identifier in the first l digits and its $(l + 1)$-st digit is i.

The choice among the potentially many nodes with this property can be made using the proximity metric.

The neighbourhood set of a node n is a set of nodes that are close to n according to a scalar *proximity metric*, e.g., number of IP routing hops or geographical distance. The use of the neighbourhood set allows Pastry to achieve good locality properties [125].

Routing in Pastry proceeds as follows. If a node n wants to locate the node with key k then it first checks whether k is included in his own leaf set. If this is the case, then n sends a *lookup* message to the member of its leaf set with identifier numerically closer to k (modulo 2^m). If that node is n, then the lookup process terminates. Otherwise, n computes the length l of the largest common prefix of k and its own identifier, and uses its routing table to send a *lookup* message to node $R[l, k[l]]$. If this is not possible because the entry $R[l, k[l]]$ of the routing table is empty, n sends a *lookup* message to the member of its leaf set with identifier numerically closer to k. The node that receives the *lookup* message proceeds in the same way so eventually we reach the node responsible for k. Figure 2.4 shows graphically how lookup is performed in Pastry. To find the node closest to identifier 011, the node whose identifier starts with 111 sends a lookup message to its neighbor whose first digit is 0. This node then forwards the query to its neighbor whose first two digits are 01, and from there the node is forwarded to the neighbour whose first three digits are 011.

We can show that this routing procedure converges in at most $O(\log_{2^b} N)$ steps assuming accurate routing tables and no recent node failures [125]. The leaf sets are used when routing tables are incomplete in exchange for potentially longer paths. But notice that leaf sets add a great deal of *resilience* to Pastry. As shown in [46], if $r = 16$ and $N = 65,536$, even after a random 30% of the network links are broken, there are still connected paths between every pair of nodes.

When a new node n with key x wants to join a Pastry network, it must first find a member node n' close to it in the proximity metric. Then, n asks n' to route a message *join* to node with key x. If n'' is the destination node of the *join* message then n', n'' and all nodes in the path from n' to n'' send their state tables to n. n uses these state tables to create a routing table, a leaf set and a neighbourhood set, and to inform other nodes of its presence. This process is completed in $O(\log^2_{2^b} N)$ steps [125]. Pastry is able to deal with node failures or silent departures (even concurrent ones) using appropriate protocols [125]. For example, if a node n detects that node L_i in its leaf set has failed, it contacts the live node with the largest index on the side of the failed node (i.e., L_r if $i > 0$ and L_{-r} if $i > 0$) and asks it for a copy of its leaf set. This

leaf set is then used by n for finding an appropriate node to substitute for L_i. Then n sends a message to all nodes in its leaf set informing them of the change. Unless r nodes with adjacent identifiers fail concurrently, this procedure guarantees that nodes lazily recover from failed nodes in their leaf set. Due to the diversity of nodes with adjacent identifiers in Pastry, such failure is unlikely to take place even for small values of r [125].

Bamboo

Bamboo is a DHT based on Pastry from where it takes the circular identifier space and the routing algorithm [122]. Bamboo improves on Pastry by using more incremental algorithms for node joins and neighbour management. This allows Bamboo to withstand very dynamic changes in network membership i.e., it is resilient to churn. *Churn* is a term used to characterize the continuous process of node arrival and departure that has been observed in deployed P2P applications such as Napster, Gnutella and Kazaa [47, 127, 135]. [122] tested an implementation of Chord (MIT Chord) and Pastry (Free Pastry v1.3 from Rice University) and demonstrated that under high churn rates, these implementations may fail to complete lookup requests and return inconsistent results for the same lookup launched from different query nodes, or return consistent results but suffer major delays due to network latency. [122] have shown using Bamboo that DHTs can handle high churn rates and identify the following three factors affecting DHTs under churn.

The first factor is *reactive* vs. *periodic* recovery from failures. In reactive recovery, if a node discovers that one of its leaf nodes has failed, it computes a new leaf set and then informs all its members as discussed in 2.2.1. In low churn situations, this approach is scalable and results in very low message costs. But as churn increases (as in the file sharing scenarios of [47, 127, 135]), periodic recovery is a better strategy: a node *periodically* shares its leaf set with each node in this set and this node responds in the same way. The second factor is selection of *message timeouts*. Nodes must make sure that the timeout for a request is judiciously selected. [122] shows that having fixed timeouts is not a good idea and more sophisticated timeout calculation is needed under churn. Finally, the last factor is proximity neighbour selection. [122] propose to run periodically a maintenance algorithm that updates routing tables for a node n to include neighbours that are closer to n in the proximity metric. They show experimentally that *global sampling* using the lookup functionality of the DHT is a technique that gives good results under churn.

2.3 Relational Data Management in P2P Networks

In the last decade, there has been an interest in data management in large-scale heterogeneous P2P systems using various data models such as relational, XML and RDF. In this section, we focus on relational P2P database systems which were the first works on P2P query processing, and more specifically, we focus on PIER [63]. In the next section we elaborate on RDF data management in P2P networks.

The first papers that envisioned a P2P database setting are [22, 45]. Several research directions were then investigated such as data models and architectures for P2P databases in [136] and [43] respectively, and Peer Data Management Systems (PDMSs) were introduced. Piazza [50] is a well-known PDMS that enables sharing and integration of heterogeneous data using schema mappings for both relational and XML data. Hyperion [124] is a PDMS built on top of its own unstructured P2P network, which supports SQL queries over heterogeneous relational data. PeerDB [110] is built on a two layer hierarchical network that provides data sharing without a shared schema and integrates mobile agents to assist in query processing. One system that is closely related with our work is PIER [63], which is a query engine for relational databases built on top of a DHT. In the following, we describe PIER in detail and compare it with our work.

2.3.1 PIER

PIER[17] is a distributed relational query engine built on top of a DHT and has been developed at University of California, Berkeley [51, 62, 63]. The main role of PIER is to enable query processing of relational data in an Internet-scale environment. It tries to apply database techniques in a wide-area, non-structured environment such as the Internet. For this reason, PIER uses "best-effort" semantics for query processing. The PIER architecture assumes that data is located in its physical location, thus wrappers are needed to extract relational data from pre-existing sources. For example, a wrapper in a file-sharing application could extract information such as file name, creation date and author, integrate it in a relational model and store it in the DHT. Finally, PIER keeps no metadata catalogs and supports no schema mappings but it assumes that schemas are known by all nodes.

PIER's architecture is a 3-tier system. On the top level there are the applications which interact with the query processor in the middle, which in turn utilizes the DHT. The query processor is a dataflow engine that supports simultaneous execution of mul-

[17] `http://pier.cs.berkeley.edu/`

tiple operators for selections, projections, distributed joins, grouping and aggregation. The user (programmer) gives a physical query plan as an input to the PIER engine. The engine parses the query plan and creates several operator graphs which can be executed independently. Each operator graph that arrives at a node is evaluated by the local dataflow. The functionality of the underlying DHT that is used by PIER is routing. A storage manager module is used for the temporary storage of tuples. When a node fails, all data stored in memory of the node is lost.

Naming in PIER is done in the following manner. Each tuple in the DHT is identified by three values, a *namespace*, a *resource id* and an *instance id*. The namespace corresponds to the table (relation) name the tuple belongs to, the resource id corresponds to an attribute value of the tuple (usually the value of the primary or any other attribute value) and the instance id corresponds to the identifier of the application. The namespace defines a region of space in the DHT and the multiple nodes that serve this range of keys. The namespace together with the resource id define a single node that serves that key, and the combination of the three defines a single object (tuple) in a single location.

One of the supported operators of PIER that is especially interesting to us is distributed join. PIER supports two implementations of the join operator, *symmetric hash join* and *fetch matches* [63]. In symmetric hash join, relations should be rehashed on the join attribute. To accomplish this, both relations have to be scanned. Each node in the namespace of the two relations performs a local scan and for each tuple a PUT is done using a unique namespace for the result relation and the value of the join attribute. Then, each node in the new namespace registers to receive a callback whenever new data is inserted in the local node. When a tuple arrives, a GET is performed to find matches in the specific namespace. Matches are concatenated to generate output tuples. In the fetch matches algorithm, it is assumed that one relation is already hashed on the join attribute. Then, the other relation is scanned and for each tuple a GET is issued for a tuple of the first relation. Once this tuple arrives, predicates are applied, concatenation is performed and results are created.

At first sight one could think that PIER and our system have various things in common. Both systems use the Bamboo DHT as the underlying P2P infrastructure in their most recent versions. However, there are a lot of differences as well. The main difference is that PIER is only a query processor for widely distributed relational data while our work relies on a full-fledged system for distributed storage and query processing especially built for RDF(S) data. In our system, data are persistently stored and if a node fails at some point in time and later on joins the network again, the node

can provide the data again to the DHT. On the other hand, PIER's data is located in its physical place and relational information (i.e., tuples) is extracted and inserted in the DHT on demand. Tuples are temporarily stored in memory hash tables at each PIER node. When a node fails those tuples are lost. Finally, PIER requires as an input from the user a physical query plan instead of a SQL query since it does not support any kind of optimization. In our work, we have fully implemented a query optimizer that is responsible for transforming a SPARQL query to a query plan.

2.4 Distributed RDF Data Management

The combination of Semantic Web technologies (i.e., RDF, RDFS and ontologies) and P2P systems can provide accurate data retrieval and efficient search in distributed application scenarios, thus, it has been the focus of many research the past few years. A most comprehensive survey of this work can be found in [143]. In the following paragraphs, we present some characteristic works in this research area which utilize either a structured overlay network or some other kind of P2P architecture.

2.4.1 Structured overlay networks

[32, 33] study the problem of evaluating RDF queries over a scalable distributed RDF repository, named RDFPeers. RDFPeers is implemented on top of the self-organized MAAN presented in [31]. MAAN builds on DHT technology by extending the Chord DHT protocol [147] to efficiently answer multi-attribute and range queries. RDFPeers is the first work to consider RDF queries on top of a structured overlay network and has influenced significantly our work.

In RDFPeers, each node uses the RDF data model to create descriptions of resources that it wants to make available to the rest of the network nodes. Each RDF triple in RDF document is indexed to three different networks nodes: it is stored once in the successor node of the identifier that is computed by hashing the subject value of the triple, and two more times by using the predicate and object values of the triple. The SHA-1 hash function [137] is used if the value is a string. If the value is a numeric one then an order preserving hash function is used, which allows efficient evaluation of range queries. An RDFPeers node can pose atomic triple queries, disjunctive and range queries and conjunctive multi-predicate queries. [32] propose a series of algorithms for evaluating these types of queries for the one-time query scenario. The general idea of these algorithms is that they use the constant parts of queries so as to create identifiers that will lead to nodes that store relevant triples. Furthermore, a simple

replication algorithm is used to improve load distribution. Finally, [32] sketches some ideas regarding publish/subscribe scenarios in RDFPeers.

GridVine [3] is a scalable semantic overlay network that supports the creation of local schemas while providing global semantic interoperability. It follows the principle of data independency and separates the logical from the physical layer. The logical layer consists of the semantic overlay for managing and mapping data and metadata schemas, while the physical layer consists of a structured P2P overlay network that efficiently routes messages. The latter is used to implement various functions at the logical layer, like attribute-based search, schema management and schema mapping management.

GridVine uses P-Grid [2], a structured overlay network based on the principles of DHTs. Peers in GridVine are able to publish available resources by creating RDF triples (metadata). An RDF triple is stored three times in the network using three different keys based on its subject, predicate and object values, as in [32]. Thus, the kinds of queries studied in [32] are also are supported by GridVine. In addition, prefix searches, e.g., on the beginning of a string representing an object value, are easily supported using P-Grid routing mechanisms.

GridVine allows peers to derive new schemas from well-known base schemas (using RDFS), providing schema inheritance. Each peer has also the possibility to create a mapping between two schemas, in which case translation links among network peers are created (using OWL). In this way, queries are propagated from one semantic domain to another. There are two approaches used for resolving translation links, the iterative and the recursive resolution. With iterative resolution, the peer issuing an RDF query tries to find and process all translation links by itself, while with recursive resolution more than one peers are involved by delegating the query and its translations.

A more recent DHT-based system for managing RDF is BabelPeers [15, 57, 58]. In BabelPeers, triples are indexed three times in the network as in [33]. The query evaluation process of the system includes two phases. In the first phase, all candidate sets for all triples and variables are retrieved from the various nodes of the network to the node that receives the query request. The second phase of the query evaluation, includes local processing of the candidate sets to find the actual answer to the query. The disadvantage of this approach is that the node that receives the query request has to do all the computation and suffers a lot of query processing load. In addition, the candidate sets transferred through the network might contain results that will never be used in the final answer of the query. This causes unnecessary traffic to the network. The authors of [57, 58] propose some methods to remove the amount of

the useless information from the candidate sets using Bloom filters. The application scenario proposed in BabelPeers is taken from the Semantic Grid research area, where the problem of resource discovery is addressed [15]. Finally, [14] addresses the problem of uneven load among the nodes of BabelPeers due to the skewness of the RDF datasets and proposes several techniques for load balancing.

2.4.2 Other P2P-based architectures

[130] proposes a new graph topology for P2P systems, called HyperCup, that allows for efficient broadcasting and searching. The authors describe a broadcast algorithm that exploits the proposed topology to reach all nodes in the network by achieving the minimum number of messages possible. Also, they show how a globally known taxonomy can be used to organize the peers in the graph topology, allowing for efficient search based on concepts of the taxonomy.

The HyperCuP algorithm is able of organizing peers of a P2P network into a recursive graph structure from the family of Cayley graphs, out of which the hypercube is the most well-known topology. All participant peers are equal (there is no central server nor super-peers) and are organized in a hypercube topology. Peers are able to join and leave the self-organized network at any time. In case that some peers are "missing", some peers in the network will occupy more than one positions in the hypercube so that the hypercube topology is maintained. HyperCuP guarantees that non-redundant messages will be created while broadcasting.

The authors of [130] make the observation that in the case of Semantic Web applications additional knowledge is available that can be exploited to further improve the performance of P2P networks. Thus, in [130] peers with identical or similar interests are grouped in concept clusters, which are organized into a hypercube topology to enable routing to specific clusters in the topology. Concept clusters are hypercubes or star graphs. A query is propagated to the appropriate concept clusters (i.e., the ones that store relevant data according to the schema information) and queries are forwarded to all peers within the cluster.

An early and influential distributed RDF repository is the Edutella system [105–107]. Edutella provides a very general RDF-based metadata infrastructure for P2P applications. [105–107] argue that a super-peer topology is the suitable topology for schema-based P2P networks and thus such a topology is used in Edutella. In an Edutella network, there are two kind of peers: the super-peers and the clients. The super-peers are organized under the HyperCup topology, while clients are connected to super-peers in a star-like fashion. Each client connects to one super-peer only. Super-

peers are used to efficiently handle all the requests of clients. On registration a client provides its super-peer with its metadata information, i.e., a description of the metadata that has been created by this client (supported schema, used values etc.). The actual metadata remains in the client peer. Each super-peer stores information about metadata usage at each client that is directly connected to it. Also, each super-peer stores schema information about its (direct) neighbour super-peers (i.e., this is a description of the metadata used by their clients). This information is used to efficiently route queries only to relevant super-peers and clients.

Edutella researchers have also concentrated on schema integration in super-peer based P2P networks for RDF. The general idea is that super-peers are used as wrappers, i.e., they maintain schema mapping information that they use while routing queries in the network so as to translate queries. One application scenario of the Edutella system is Elena [140], a mediation infrastructure for educational services that are announced and mediated by electronic means. Elena is an EU/IST project that follows the vision of creating and testing Smart Spaces for Learning^{TM}, which are open environments where learners can choose learning services from heterogeneous sources.

The authors of [155] are inspired by the success of ranking algorithms in Web search engines and top-k retrieval algorithms in databases, and they propose a distributed top-k evaluation algorithm for P2P networks that retrieves the k most relevant answers for each query. This research can be very important even for scalability reasons since usually by increasing the network size, also the number of answers in given queries will also be increased since more data are available in the network.

The algorithm that is proposed in [155] delivers the k more relevant results without having to rely on any centralized knowledge and without the need for a complete distributed index. The architecture that is assumed for the P2P network is the super-peer architecture of Edutella that we have already described above. The proposed algorithm allows the optimization of query distribution and routing. It makes use of ranking methods in order to reduce the overall number of answers in the result set, and also to return close matches avoiding empty result sets in case that no exact matches are found. Each peer computes local rankings for a given query that it receives, and sends the results to the super-peer that it is connected to. Each super-peer again merges results from local peers and from neighboring super-peers and forwards only the "best" results towards the super-peer of the peer that posed the query. Then the results are merged and ranked again at the super-peer (where the node that posed the query is attached) and finally from there are routed back to the query originator. While results are routed through the super-peers, the algorithm maintains statistics regarding which

peers/super-peers returned the best results, and uses this information to distribute queries (that have been already posed in the past) only to the most promising peers. The algorithm minimizes the answer set size and thus the network traffic.

SQPeer [83, 84, 139] is a middleware for routing and planning complex queries in P2P database systems, exploiting the schemas of peers. In SQPeer, each peer provides RDF(S) descriptions about information resources that wants to make available in the network. Peers that employ the same schema belong essentially to the same semantic overlay network (SON) [157]. Queries in SQPeer are formulated in RQL [81] according to the RDF schema that the requester peer supports. Also, each peer is able to *advertise* the content of its local base (the actual data values or the actual schema of its base) using RVL *view patterns* [94]. The proposed query routing and query processing algorithm can find the relevant peers that actually answer each query and generate query plans by taking into account statistics on data distributions. SQPeer SONs can be implemented on two different architectural alternatives; a hybrid P2P architecture or a structured P2P based on DHTs. However, no experimental evaluation is presented.

The SWAP project[18] has studied the combination of Semantic Web and P2P systems. The vision of this project is to allow peers to maintain individual views of knowledge, while they will also be able to share their knowledge with the rest of the network. In [39], the authors consider a P2P topology where all peers are equal. They propose an RDFS-based metadata model for encoding semantically rich information regarding peers and their knowledge resources. The metadata model consists of two RDFS classes, the "Swabbi"-class and the "Peer"-class, which contain several properties. The former is used for the annotation of every piece of knowledge is the P2P system, while the latter is used for the description of the peer that originates this knowledge. For example, the "Swabbi"-class contains information in terms of the URI, the location and the label of the described knowledge. Also, it describes how reliable a statement is and indicates its access control by the other peers. The "Peer"-class provides characteristics like the id and the label of each peer, or how reliable a peer is.

Bibster [49] is a system implemented as an instance of the SWAP platform which is based on JXTA[19]. It is a P2P system, for exchanging bibliographic data (BibTex entries). Bibster allows search for bibliographic entries using keyword searches, as well as more advanced semantic searches. It also provides the integration of a query's results into a local knowledge base for further use. Bibster exploits ontologies for importing data, formulating and routing queries and processing answers. In particular,

[18] http://swap.semanticweb.org/
[19] http://www.jxta.org

it uses the Semantic Web Research Community ontology (SWRC[20]) and the ACM topic hierarchy[21]. Each peer manages a local RDF repository with bibliographic data and uses the ACM topic hierarchy to advertise the semantic description (termed *expertise*) of its repository in the P2P network. In addition, it is able to pose queries in SeRQL [29]. During query processing, a peer first evaluates the query against its local node repository and then decides where it should be forwarded. This decision is based on the *subject* of the query, namely an abstraction that specifies the required expertise to answer the query, and therefore the peer forwards the query to the peer with the appropriate expertise.

REMINDIN' (Routing Enabled by Memorizing INformation about Distributed INformation) [153] is an algorithm that exploits social metaphors to find the right peers in a semantic peer-to-peer network to answer a given query. It has been implemented for using the SWAP platform. In REMINDIN', peers observe which queries are successfully answered by other peers, memorize this information and use it when they want to forward future requests. When a peer issues a query (using SeRQL), this query is evaluated locally and across the network. The requester peer selects a set of peers that it is possible to be able to answer its query. If it cannot select any peers to forward its query, it weakens the query conditions (relaxes the query, e.g., it creates a more general query) and repeats the procedure. When the selection of appropriate peers is over, then the original query is sent to the selected peers. When a peer receives a query, it searches for answers. Any answers created are returned directly to the requester peer. The latter stores the relevant answers in its local repository and rates them. Those answers identify the set of the most knowledgeable peers, thus the querying peer can use this information for further requests.

Another paper that studies the searching of the Semantic Web is [152]. [152] presents SERSE (Semantic Routing System), a multi-agent system appropriate for searching the Semantic Web. This system combines different technologies from different areas such as P2P, ontologies and multi-agent technology. In SERSE, the routing agents have the same capabilities and responsibilities and communicate on a P2P basis. The available resources are semantically annotated and agents are responsible to retrieve these based on their descriptions (annotations). Semantic descriptions of resources determine a semantic overlay on the P2P network, where each peer is able to communicate only with those peers that belong to the same *semantic neighborhood*. There is no global knowledge of the network, namely each peer knows just its immediate neighbours, and

[20] http://www.semanticweb.org/ontologies/swrc-onto-2001-12-11.daml
[21] http://www.acm.org/class/1998/

also agents cannot broadcast messages to the whole network.

2.4.3 Other distributed architectures

Lately there is an emerging interest in the area of distributed SPARQL query processing in general. However, none of the existing systems have become very popular yet. Although the architecture of these systems is very different from the one offered by our system, we briefly mention them in this section.

In [114], a clustered triple store for Jena is presented. Two types of nodes are required in this setting; query coordinator nodes and data nodes. A query coordinator node is responsible for receiving queries, transforming URIs and literals to identifiers, and controlling the execution on the data nodes. A data node is responsible for storing the node relation and triple indexes and performing operations such as sorts and joins. Hash partitioning is used for distributing the identifiers index and the triple indexes.

In [55], YARS2 is presented, a federated repository for RDF data. The authors of [55] study distributed indexing methods for graph-structured data and parallel query evaluation methods on a cluster of computers using a hash-based placement. For joins they use index nested loops join. In Virtuoso Cluster Edition [40], RDF data can be stored as RDF quads using bitmap indices. Query evaluation is performed in a pipeline way. Each step of the pipeline takes the input of the previous stage, partitions it and sends an appropriate message to the cluster nodes involved. If intermediate sets are large, they are processed in consecutive chunks. The execution of the pipeline steps may overlap in time and can be divided into separate partitions.

2.5 RDFS Reasoning

In order for an RDF(S) system to support the complete semantics and capabilities of RDF(S), it also needs to support the RDFS entailment rules [56]. In this section, we survey related work in the area of RDFS reasoning in both centralized environments as well as distributed ones of various architectures. We compare these methods and architectures with our work on distributed RDFS reasoning presented in Chapter 5.

2.5.1 Centralized systems

A representative centralized RDF store that supports a forward chaining approach is Sesame [30]. Each time an RDF Schema is uploaded in Sesame, an inference module computes the closure of a given dataset udner the RDFS entailment and asserts the

inferred RDF statements. So, every RDF statement, explicit or implicit, is stored in Sesame's database. Jena [168] has a generic reasoning module designed to allow the usage of any kind of reasoner. Therefore, Jena can eventually support different approaches depending on the underlying reasoner. RDFSuite[22] and specifically RSSDB [7] follows a totally different approach in which the taxonomies are stored using the underlying DBMS inheritance capabilities so that retrieval is more efficient. For example, if a class c is represented as a table (relation) R, a subclass of c would be a subtable of R in the underlying DBMS. Nevertheless, this approach is still an *on demand* approach and resembles the backward chaining evaluation algorithm. 3store [52] follows a hybrid approach in order to gain from the advantages of both approaches and avoid their disadvantages. In [52], the authors have chosen which inference rules will be evaluated a priori using forward chaining when new facts are asserted, and rules which have greater storage load and lower query processing cost will be evaluated on demand with backward chaining and query rewriting.

In the Oracle RDBMS [34], RDFS inference is done at query execution time using appropriate SQL queries to the underlying relations. However, if a rule is used frequently, then the system can determine that inferencing can be done using forward chaining to pre-compute the inferred triples and store them in a separate relation. Virtuoso[23] also provides support for RDF(S) reasoning [41]. Virtuoso's SPARQL implementation supports inference at run time by rewriting the query appropriately to retrieve all inferred answers. Finally, GiaBATA [64] is a prototype system which uses logic programming approaches coupled with a persistent relational database for implementing SPARQL with dynamic rule-based inference. In [65], the authors of GiaBATA employ different optimization techniques like magic set rewriting in an effort to stay competitive as they extend SPARQL to be able to use a custom ruleset for specifying inferences.

2.5.2 Structured overlay networks

A DHT-based RDF store that is closely related with our work is BabelPeers [16] which was the first system to support RDFS reasoning using DHTs. It is implemented on top of Pastry [125] and only a forward chaining approach is supported. RDF(S) triples are distributed in the network using the indexing scheme of [33]. The reasoning process runs in regular intervals at each node and checks for new triples that have arrived to the node. Then, it exhaustively generates new inferred triples based on the RDFS inference rules

[22] `http://139.91.183.30:9090/RDF/`
[23] `http://virtuoso.openlinksw.com/`

and sends them to be stored in the network. [16] presents no experimental evaluation of the forward chaining algorithm. However, the results of the experimental evaluation of our forward chaining algorithm show how expensive such an approach is in terms of storage load, time and bandwidth and could demonstrate a poor performance of the reasoning algorithm of [16].

In [42], the authors present a DHT system called DORS for distributed ontology reasoning. All nodes share the same schema (TBox), while the instances of the data (ABox) are distributed in the network using a DHT partitioning scheme. Each node uses a DL reasoner to infer the complete subsumption relationships among classes and properties of the TBox. Although TBox reasoning is taking place at each node independently, ABox reasoning is performed in a distributed manner iteratively until no new inferences are produced, similarly with our forward chaining approach. A prefetch procedure retrieves the required data before the rule engine starts the reasoning process. Then, the inferred assertions are distributed according to the DHT partitioning scheme until no new assertions are generated. The experiments presented in [42] are performed in a network of up to only 32 nodes and for a small dataset.

Finally, a different research area that is related with our work is declarative networking. In [90, 91], a variation of Datalog is used for expressing routing protocols in a simple and compact way based on the observation that recursive query languages are very suitable for expressing routing protocols. The implementation of the routing protocols in [90, 91] can then be done by evaluating recursive queries in a distributed manner. In [90] a semi-naive bottom-up algorithm is presented for finding paths in a physical network graph and an optimized version of it based on the magic set rewriting techniques [13]. The notion of iteration in the semi-naive algorithm of [90] is kept local to each node.

2.5.3 Other distributed and parallel architectures

Apart from DHTs which were the first infrastructures proposed for distributed RDF(S) query processing and reasoning, other distributed and parallel computing infrastructures have been proposed lately. Such infrastructures are based on distributed computing platforms consisting of powerful clusters and cloud computing platforms using the MapReduce framework [37].

MaRVIN [112, 113] is a parallel and distributed platform for RDFS reasoning over large amount of RDF(S) data. MaRVIN supports a forward chaining approach for

RDFS reasoning and runs on DAS-3 (Distributed ASCI Supercomputer[24]). The creators of MaRVIN point out that a distribution of RDF(S) triples based on a DHT can suffer from load imbalances due to the skewness of RDF data [85]. Therefore, they propose an approach of *divide-conquer-swap* where triples are fairly partitioned to all peers, each peer performs the reasoning, repartitions its triples and swaps it with another peer. Although this method produces sound results, it is not complete. In [113], the authors present an analytical model to prove that their system will eventually reach completeness over time. The swapping phase can be either random or using another algorithm called SpeedDate in [113] which enables data clustering and thus improves the chances of getting completeness earlier. However, the amount of time required to reach completeness remains questionable. In addition, the system uses an in-memory implementation at each peer which speeds up inferencing significantly but if a node fails all triples are lost. Datasets used in the experiments of [113] contain up to 14.9 million triples.

In [163], a different forward chaining approach is proposed based on MapReduce [37]. The system is implemented on top of Hadoop[25] and runs on the DAS-3 distributed supercomputer managing to scale to 865 millions of triples. The authors of [163] show that a naive implementation is inefficient due to load-balancing problems and the generation of many duplicate triples and propose three optimizations to achieve an RDFS closure computation more efficiently. Firstly, the RDFS triples are kept in memory since they are fewer than the RDF data triples. Secondly, data is grouped in a way which prevents the generation of duplicate triples and avoids load balancing problems. This is performed by using as keys more than one part of the triples in some cases. Finally, the rules are executed in a specific order so that the number of iterations required is limited. Load-balancing issues are also handled by the Hadoop framework which dynamically assigns tasks to optimize the workload of each node. In [163], the authors also utilize a distributed dictionary encoding in MapReduce. However the time required to encode the triples to dictionary identifiers is not taken into consideration in the experiments presented.

[166] considers the problem of producing the full RDFS closure using parallel computing techniques. The authors show that RDFS rules have certain properties that allow for an embarrassingly parallel algorithm. This means that the RDFS reasoning task can be divided into a number of completely independent tasks that can be executed in parallel by separate processes. Similarly with [42], a distinction is made

[24] `http://www.cs.vu.nl/das3/`
[25] `http://hadoop.apache.org/`

between triples that describe RDFS information (referred as ontological triples) and triples that encode RDF data information (referred as assertional triples). The partitioning scheme requires each process to have all ontological triples, while assertional triples are split equally to the processes. Each process iterates over the RDFS rules in the appropriate order until no more inferences are found. The inferred triples produced from a processor are added to the the set of triples of the same processor. The authors show that this algorithm is sound and complete with respect to the RDFS rules supported. A disadvantage of this approach is that each process outputs the set of triples to a separate file. This introduces the problem of having different processes producing the same triples and thus the resulting data set contains many duplicate triples. As one would expect, removing duplicates would require much time and would sacrifice the scalability of the algorithm. Experiments were conducted in an Opteron blade cluster using machines with 16GB memory each and the datasets used contained up to ~ 346 million triples from the LUBM benchmark.

In [126], the authors present the reasoning engine of 4store [53] which runs in a backward chaining fashion. 4store is a clustered RDF store which uses the subject of each triple to decide to which cluster node the triple should be stored. Then a query processing node is responsible for retrieving the required data and compute the answer to a query. The reasoning engine of 4store works in a backward chaining fashion but keeps all RDFS information at one cluster node. The experiments presented in [126] were conducted in 5 Dell PowerEdge R410 machines, each of them with 4 dual core processors at 2.27GHz, 48GB memory and 15k rpm disks managing to scale to 500 million triples.

Other related approaches consider OWL reasoning in parallel platforms. One such approach is presented in [142], where two partitioning approaches are studied. The first one partitions the data which are then processed independently. The second one partitions the rules and each process applies its rules to the complete data set. The authors of [163] go one step further by using MapReduce to compute the closure of RDF graphs under the OWL Horst semantics in [162]. In [103], a MapReduce algorithm is presented for classifying EL^+ ontologies, following the paradigm of [162, 163] for RDFS ontologies.

2.6 RDF Query Optimization

RDF query optimization has received a lot of attention only recently. In this section we survey query optimization techniques that have been proposed for RDF in both

centralized and distributed settings.

2.6.1 Centralized approaches

Recent attention has been given to SPARQL query optimization in centralized environments [108, 109, 145]. The first proposal is [145], where the authors present a selectivity-based framework for optimizing SPARQL BGP queries. The optimization algorithm proposed is based on the minimum selectivity heuristic of [144]. The join selectivity is precomputed by executing the actual SPARQL queries on every pair of predicates that is related through the RDFS schema and keeping the size of all the result sets. In the absence of an RDFS schema all combinations of distinct predicates should be considered, an approach that can become very expensive. For that reason, the query optimizer can be set to either fully support the estimation of the selectivity of both triple patterns and joined triple patterns or support the estimation of the selectivity of triple patterns only.

RDF-3X [108] is a RISC-style RDF engine designed for the management and querying of RDF data. RDF-3X exhaustively builds aggregated indexes on every combination of triple components and uses compression techniques to decrease the size of these indexes. In [108], the join ordering is determined using a bottom-up dynamic programming algorithm [100]. For the selectivity estimation of joins of triple patterns the authors propose two kinds of statistics. The first one uses specialized histograms which can handle both triple patterns and joins by leveraging the aggregated indexes built. Since histograms assume the independence between predicates, they also consider the computation of frequent join paths in the RDF graphs. The exact join statistics of the most frequent paths are precomputed and kept in main memory. During query optimization, the exact join statistics are used if available, otherwise independence is assumed and the appropriate histograms are used. In [109], the authors of RDF-3X go one step further to propose more accurate selectivity estimations by precomputing exact join cardinalities for all possible choices of one or two constants in a triple pattern and materializing the results in additional indexes. Certainly a good estimation of the join cardinality using exact statistics comes with the cost of a large amount of time for precomputing these statistics and the occupation of a large amount of space. For example, for a total database size of 41GB, the computation of the statistics took 30 minutes and they occupied 3GB on disk.

In order to enable the query optimization process, the cardinality estimation of RDF graphs is crucial. A method for the cardinality estimation of SPARQL queries using a probability distribution is presented in [138]. The algorithm relies on the decomposition

of queries into query pattern paths, where each path produces a set of values for each variable within the result form of the query. In order to estimate the total number of result set parameters for each path, a set of statistics is created on the properties of the ontology. In [93] the authors propose graph summaries for the estimation of the frequency of subgraph patterns using techniques from graph databases. They propose two tree-based structures which encode the frequency of a graph and the dependencies between parts of the graph.

2.6.2 Distributed approaches

Earlier works that consider SPARQL query processing on top of DHTs such as [32, 57, 70, 89] lack optimization techniques resulting in handling very small datasets (only thousands of triples). The only DHT-based work that deals with query optimization for a triple-based model is Unistore [79, 80]. Unistore is built on top of P-Grid DHT [2] and supports a SPARQL-like query language. Each query is transformed into a logical query plan which is in turn transformed into a physical query plan using operators defined in [79]. In Unistore, a cost-based optimizer is implemented which estimates the cost of physical operators in terms of the number of hops and messages required for each operator. Query planning is performed dynamically at each peer involved in the query evaluation. The experimental evaluation presented in [79, 80] is conducted in PlanetLab and hence only small datasets are used. The work of [79] is complementary to ours and the two approaches could actually be combined by an appropriate cost model.

Earlier works that studied RDF query optimization in a distributed environment, although not a DHT, include [84, 120, 150]. In [150], several RDF repositories are combined using a mediator and can be accessed through a single SeRQL Parser. The authors of [150] address the problem of query optimization by defining a cost model which takes into consideration the local query processing cost (i.e., the cost for the join operation) as well as the communication cost. For determining the join order, the authors adopt randomized algorithms as proposed in the database literature [144]. The join selectivity of two triple patterns is assumed to be estimated but no such method is clearly proposed.

In [120], the authors present an engine for federated SPARQL databases and make use of query rewriting and cost-based optimization techniques. The query rewriting technique lies in the simple substitution of variables appearing in filter expressions with an equal operator and in the widely used strategy from relational systems where selections are applied as early as possible. For the cost-based optimization, they use

an iterative dynamic programming but fail to estimate the selectivity of conjunctions of triple patterns and set it to a fixed value instead.

As mentioned in Section 2.4.2, SQPeer [84] is a P2P middleware for routing and planning queries for RDF(S) data based on semantic overlay networks. SQPeer supports two kinds of query optimization for query planning, i.e., compile-time and run-time optimization. Optimization relies on heuristics as well as a cost-based model. Cost-based optimization utilizes statistics about the communication cost between peers and the size of expected intermediate query results. These statistics can be used to decide which peer and in what order will execute each query operator. In addition, local processing load of peers is also crucial and can be kept in the statistics. Compile-time optimization uses heuristics and statistics to determine at compile time between data, query or hybrid shipping execution strategies. On the other hand, run-time optimization includes deciding at execution time on altering the data or query shipping decision or discovering alternative peers for answering a certain part of a query plan. However, in [84] the cost model about the expected size of the intermediate results is not given and a detailed way of how to measure the communication cost between the nodes is missing. Finally, [84] presents no experimental evaluation.

A more recent work on data summaries over Linked Data is presented in [54]. The authors of [54] describe a distributed approach for answering SPARQL queries over Linked Data sources. Their proposal is based on data summaries for finding quickly which are the sources that contain relevant information to answer a query. For summarizing data sources they use an index, called QTree, which combines histograms with R-trees.

2.7 Summary

In this chapter we introduced the RDF and RDFS data model and the SPARQL query language. In addition, we presented a brief survey of P2P networks focusing on the technology of DHTs. We surveyed work most relevant to our work including P2P database systems and proposals for managing RDF data in P2P networks. Then, we surveyed related work in the area of RDFS reasoning in centralized and distributed environments of various architectures. Finally, we presented optimization techniques that have been proposed recently for RDF query optimization both in centralized and distributed settings.

In the next chapters we present in detail the contributions of our work. We start by introducing the data model and query language supported by our system.

Chapter 3

Data Model and Query Language

In this chapter we present concepts, terminology and results that are used throughout the thesis. We start with formal definitions for the RDF and RDFS data model. We choose to rely on the minimal deductive system of [101] for our RDFS data model. Therefore, we present useful results from [101] that we use for our theoretical results in Chapter 5. Finally, we present the fragment of the SPARQL query language and its graph representation that we use in our system.

3.1 RDF and RDFS Data Model

In our work we constantly make use of the notion of an RDF triple. More formally, we define an RDF triple as follows.

Definition 3.1.1. *Let U, L and B denote the sets of URIs, literals, and blank nodes, respectively. These sets are pairwise disjoint. An* RDF triple *is a tuple (s, p, o) from $(U \cup B) \times U \times (U \cup L \cup B)$, where s is the subject, p is the predicate and o is the object of the triple.*

URIs, literals and blank nodes were introduced in Section 2.1. We will call a triple *ground* if it contains no blank nodes.

RDF data as well as RDFS descriptions can be written as RDF triples of the above form. We will call a set of RDF(S) triples an *RDF(S) database* or an *RDF(S) graph*. An RDF(S) graph is *ground* if the set of triples of the graph contains only ground triples. In the theoretical part of our work we deal with ground RDF(S) graphs. Blank nodes, however, do not affect our implementation.

Table 3.1: *ρdf* inference rules [101]

1 (simple)	(a) $\frac{G}{G'}$ for a map $\mu : G' \to G$	(b) $\frac{G}{G'}$ for $G' \subseteq G$
2 (subproperty)	(a) $\frac{(A,sp,B)(B,sp,C)}{(A,sp,C)}$	(b) $\frac{(A,sp,B)(X,A,Y)}{(A,B,Y)}$
3 (subclass)	(a) $\frac{(A,sc,B)(B,sc,C)}{(A,sc,C)}$	(b) $\frac{(A,sc,B)(X,type,A)}{(X,type,B)}$
4 (typing)	(a) $\frac{(A,dom,B)(X,A,Y)}{(X,type,B)}$	(b) $\frac{(A,range,B)(X,A,Y)}{(Y,type,B)}$
5 (implicit typing)	(a) $\frac{(A,dom,B)(C,sp,A)(X,C,Y)}{(X,type,B)}$	(b) $\frac{(A,range,B)(C,sp,A)(X,C,Y)}{(Y,type,B)}$
6 (subproprerty reflexivity)	(a) $\frac{(X,A,Y)}{(A,sp,A)}$ (c) $\frac{(A,sp,B)}{(A,sp,A)(B,sp,B)}$	(b) $\frac{}{(p,sp,p)}$ for $p \in \rho df$ (d) $\frac{(A,p,X)}{(A,sp,A)}$ for $p \in \{dom, range\}$
7 (subclass reflexivity)	(a) $\frac{(A,sc,B)}{(A,sc,A)(B,sc,B)}$	(b) $\frac{(X,p,A)}{(A,sc,A)}$ for $p \in \{dom, range\}$

3.2 RDFS Entailment

As we have already discussed in Section 2.1, RDFS entailment plays an important role in RDF(S) graphs. Inferences are made using the RDFS entailment rules of RDF Semantics [56]. In our work, we deal with a subset of the RDFS entailment rules of [56] and give a different notation using Datalog. This subset of rules is also proposed in [101] as a minimal model for RDFS and is proved to be sound and complete. First, we provide details about the minimal deductive system for RDFS of [101] and then, we present our Datalog-like approach.

3.2.1 A minimal deductive system for RDFS

To support RDFS reasoning one could use the sound RDFS inference rules presented in RDF Semantics [56] or their extension given in [154] which is also complete. We choose to base our work on the minimal set of inference rules of [101] which we briefly present here. This set of inference rules covers only the subset of the RDFS vocabulary which is truly useful for modeling an application domain, and leaves out vocabulary and inferences that capture the (possibly complicated) internals of RDF and RDFS.

In [101, 102], the authors start with a subset of RDF and RDFS vocabulary which they call *ρdf*. The only RDFS predefined terms allowed in the *ρdf* vocabulary are `rdfs:subPropertyOf`, `rdfs:subClassOf`, `rdfs:domain`, `rdfs:range` and `rdf:type`. [101, 102] give a deductive system over a minimal set of inference rules over *ρdf* and prove that it is sound and complete. For ease of the reader, we present below some of the

definitions and results of [101] that we use throughout the thesis. Table 3.1 shows the inference rules used for the *ρdf* fragment. The following abbreviations are used in Table 3.1 and the rest of the thesis: `sc` for `rdfs:subClassOf`, `sp` for `rdfs:subPropertyOf`, `type` for `rdf:type`, `dom` for `rdfs:domain` and `range` for `rdfs:range`.

In the following, we give the definition of a *ρdf* proof. First, let us define the concept of a map. A *map* is a function $\mu : U \cup B \cup L \rightarrow U \cup B \cup L$ preserving URIs and literals, i.e., $\mu(u) = u$ for all $u \in U \cup L$. Given an RDF(S) graph G, $\mu(G)$ is defined as the set of all triples $(\mu(s), \mu(p), \mu(o))$ such that $(s, p, o) \in G$. In [101], the authors overload the meaning of a map and speak of a map μ from G_1 to G_2 ($\mu : G_1 \rightarrow G_2$), if the map μ is such that $\mu(G_1)$ is a subgraph of G_2.

Definition 3.2.1. *Let G and H be RDFS graphs. We will say that there is a ρdf* proof *of H from G (denoted by $G \vdash_{\rho df} H$) if and only if there exists a sequence of graphs $P_0, P_1, ..., P_k$, with $P_0 = G$ and $P_k = H$, and for each j ($1 \leq j \leq k$) one of the following cases hold:*

- *there exists a map $\mu : P_j \rightarrow P_{j-1}$ (rule (1a)),*
- *$P_j \subseteq P_{j-1}$ (rule (1b)),*
- *there is an instantiation $\frac{R}{R'}$ of one of the rules (2)-(7), such that $R \subseteq P_{j-1}$ and $Pj = P_{j-1} \cup R'$.*

[101] constrain the *ρdf* subset further by disallowing *ρdf* vocabulary as subject or object of a triple. The new subset of RDFS is called minimal RDFS and denoted by *mrdf*. This is the subset of RDFS we use in our work. A *minimal RDFS triple (mrdf-triple)* is a ground *ρdf* triple having no *ρdf* vocabulary as subject or object [101]. A *mrdf-graph* is a set of mrdf-triples. Based on this notion and the Definition 3.2.1 of a *ρdf* proof, the authors of [101] define a *mrdf* proof as follows.

Definition 3.2.2. *Let G, H be mrdf-graphs. Then, $G \vdash_{mrdf} H$ if and only if there is a ρdf proof of H from G involving solely the rules (1b), (2), (3) and (4).*

Each pair (P_{j-1}, P_j), $1 \leq j \leq k$ is called a *mrdf step* of the proof or *mrdf proof step*, which is labeled by the respective instantiation of the rule applied. Whenever $G \vdash_{mrdf} H$, we can also say that the graph H is derived from the graph G or H is *mrdf*-entailed from G.

Let $\models$ be the RDFS entailment relation defined in RDF Semantics [56]. The following result presented in [101] shows that the normative semantics of RDFS are preserved

Table 3.2: RDFS Entailment Rules

Rule	Head	Body
1 (1b)	`newTriple(?X, ?P, ?Y)`	`triple(?X, ?P, ?Y)`
2 (2a)	`newTriple(?X, sp, ?Y)`	`triple(?X, sp, ?Z), newTriple(?Z, sp, ?Y)`
3 (2b)	`newTriple(?X, ?P, ?Y)`	`triple(?X, ?P1, ?Y), newTriple(?P1, sp, ?P)`
4 (3a)	`newTriple(?X, sc, ?Y)`	`triple(?X, sc, ?Z), newTriple(?Z, sc, ?Y)`
5 (3b)	`newTriple(?X, type, ?Y)`	`triple(?X, type, ?Z), newTriple(?Z, sc, ?Y)`
6 (4a)	`newTriple(?X, type, ?Y)`	`newTriple(?X, ?P, ?Z), triple(?P, dom, ?Y)`
7 (4b)	`newTriple(?X, type, ?Y)`	`newTriple(?Z, ?P, ?X), triple(?P, range, ?Y)`

by the deductive system $\vdash_{mrdf}$ for those mrdf-graphs that do not contain triples of the form (x, sp, x) or (x, sc, x) for $x \in U \cup L$.

Theorem 1. *Let G and H be mrdf-graphs. Assume that H does not contain triples of the form (x, sp, x) nor (x, sc, x) for $x \in U \cup L$. Then $G \models H \Leftrightarrow G \vdash_{mrdf} H$.*

In Chapter 5, we use the deductive system for *mrdf* to prove that our RDFS reasoning algorithms are sound and complete.

3.2.2 A data model for RDFS based on Datalog

We base our RDFS data model on the minimal RDFS fragment *mrdf* of [101]. Following Theorem 1 and by assuming ground graphs, our algorithms focus on the *ρdf* inference rules 1(b) and (2)-(4) of Table 3.1. In addition, we do not consider reflexive triples such as (x, sc, x) or (x, sp, x).

Following a datalog-like notation with the extensional database relation (edb) `triple` and the intensional database relation (idb) `newTriple`, the RDFS entailment rules can be written as shown in Table 3.2. Each rule is indexed by a number that we use to refer to it. Rules are also indexed with their number on the deductive rules for the *ρdf* fragment for co-reference reasons.

In our notation, arguments beginning with "?" (such as $?X$ and $?Y$) denote variables, and arguments starting with a lowercase letter denote constants. Predicate names always start with a lowercase letter. To avoid confusion with the double meaning of the word *predicate*, we will refer to the predicate p of an RDF triple (s, p, o) with the word *property* and to a term of a datalog rule with the word *predicate*.

In comparison with the rules of the deductive system *ρdf* of [101], rule 1 is actually rule (1b) (simple) and is responsible for inferencing all triples that are initially stored in the network, rules 2 and 3 represent rules (2) (subproperty), rules 4 and 5 represent rules (3) (subclass), and rules 6, 7 represent rules (4) (typing).

We note that all recursive rules above are linear and safe. *Linear* rules are rules with at most one recursive predicate in their body. *Safe* rules are rules where all variables in the head of the rule appear as an argument in the rule bodies. The order in which the predicates appear in a rule body is not semantically important. However, as it is explained later in Section 5.2, we have adopted a specific ordering pattern for the case of the backward chaining algorithm.

3.3 Query Language

In this section we present the query language supported by our system. First, we introduce some formal definitions and then we describe how we represent the queries internally in our system.

3.3.1 Definitions

As already discussed in Section 2.1.3, the core construct of SPARQL is a basic graph pattern, i.e., a conjunction of triple patterns. A triple pattern is a subject-predicate-object tuple where the components can be either constants or variables. More formally, we define a triple pattern as follows.

Definition 3.3.1. *Let U, L and V denote the pairwise disjoint sets of URIs, literals and variables respectively. A* triple pattern *is a tuple (s, p, o) from $(U \cup V) \times (U \cup V) \times (U \cup L \cup V)$.*

We find useful to represent a triple pattern query using a *single* idb relation `newTriple`. A triple pattern `(s, p, o)` is then written as `newTriple(s, p, o)`. We navigate freely between these representations throughout the thesis.

A *basic graph pattern* (BGP) of a SPARQL query [119] is a conjunction of triple patterns and we express it as follows:

$$?x_1, ..., ?x_k : (s_1, p_1, o_1) \wedge (s_2, p_2, o_2) \wedge ... \wedge (s_n, p_n, o_n)$$

where $?x_1, ..., ?x_k \in V$ are variables and (s_i, p_i, o_i) is a triple pattern as defined above. Variables $?x_1, \ldots, ?x_k$ are called *answer variables* and each variable $?x_i$ appears in at

least one triple pattern. Alternatively, we could express a basic graph pattern query using the Datalog form presented above as follows:

$$\texttt{ans}(?x_1, ..., ?x_k) \text{ :- } \texttt{newTriple}(s_1, p_1, o_1), \texttt{newTriple}(s_2, p_2, o_2), ..., \texttt{newTriple}(s_n, p_n, o_n)$$

The following definition enables us to define the answer to a conjunctive triple pattern query.

Definition 3.3.2. *Let θ be a substitution $\{?x_1/c_1, ..., ?x_k/c_k\}$ and ϕ a formula $(s_1, p_1, o_1) \wedge (s_2, p_2, o_2) \wedge ... \wedge (s_n, p_n, o_n)$. The* application *$\theta(\phi)$ of θ to ϕ is the formula obtained by replacing each occurrence of the variable $?x_i$ in the formula ϕ by the constant c_i $(i = 1, ..., k)$.*

We formally define the answer to a SPARQL query of the above form as follows.

Definition 3.3.3. *Let DB be an RDF(S) database and q a conjunctive query $q_1 \wedge ... \wedge q_n$ where q_i is a triple pattern. The* answer *to q over database DB consists of all substitutions $\{\theta_1, ..., \theta_k\}$ such that the application of any of these substitutions to the query q should result in a conjunction of triples H that are mrdf-entailed from DB (i.e., $DB \vdash_{mrdf} \theta_j(q)$ for each $j = 1, ..., k$).*

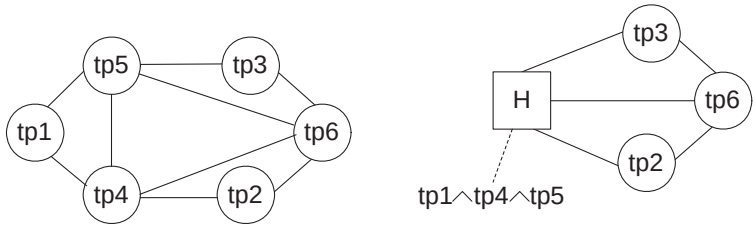
We note that the above definition does not use mappings as in the standard semantics of SPARQL [116, 119]. Since we do not deal with blank nodes in this thesis, the above simpler definition is adequate.

3.3.2 Graph representation of SPARQL

In the following, we define an internal representation of a SPARQL query adopting and extending the graph-based approach used in [108, 145]. This representation aims at avoiding the computation of Cartesian products during the query evaluation.

Definition 3.3.4. *A* query graph *g is a tuple (N, H, E), where N is the set of nodes in g, H is the set of hypernodes in g and E is the set of undirected edges in g. Each node in N denotes a single triple pattern and each node in H denotes a conjunction of triple patterns. Two nodes from $N \cup H$ are connected with an edge in E if and only if the triple pattern or the conjunction of triple patterns represented by these two nodes share at least one variable.*

Initially, the query graph of a query consists only of simple nodes. During query processing, evaluated triple patterns are merged into hypernodes. In the rest of the



(a) Initial query graph (b) Intermediate query graph

Figure 3.1: Query graph examples for query Q9

thesis, we focus only on connected graphs. The evaluation of unconnected graphs is straightforward since each connected subgraph can be evaluated independently, and then the union of the results can be created at the peer that posed the query.

Listing 3.1: SPARQL query Q9 of LUBM

```
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX ub:<http://www.lehigh.edu/~zhp2/2004/0401/univ-bench.owl#>
SELECT ?x ?y ?z
WHERE {
    ?x rdf:type ub:Student . (tp1)
    ?y rdf:type ub:Faculty . (tp2)
    ?z rdf:type ub:Course .  (tp3)
    ?x ub:advisor  ?y.       (tp4)
    ?x ub:takesCourse  ?z .  (tp5)
    ?y ub:teacherOf  ?z   .  (tp6)
}
```

In Listing 3.1, we present a SPARQL query. The query is query Q9 from the LUBM benchmark [48] and asks for the students who take courses taught by their advisor. The initial query graph is shown in Figure 3.1(a). Figure 3.1(b) shows an intermediate query graph where the hypernode H represents the conjunction of triple patterns $tp1 \wedge tp4 \wedge tp5$ (i.e., triple patterns $tp1$, $tp4$ and $tp5$ have already been evaluated).

3.4 Summary

In this chapter we presented formal definitions for the RDF and RDFS data model and useful results from [101] that we use for our theoretical results in Chapter 5. Additionally, we described the part of the SPARQL query language we use in our system and a way to model queries during query evaluation using a graph representation.

In the next chapter we give a full description of our system, Atlas. We describe its

architecture, its API, as well as an application scenario that demonstrates its role.

Chapter 4

A Peer-to-Peer System for RDF Data Management

In this chapter we present a system that is able to support full-fledged management of RDF data in a large-scale decentralized environment. We assume a setting where several heterogeneous sources of data are geographically distributed and expose their contents in RDF(S). Our system's architecture relies on the P2P paradigm but uses a DHT to ensure *complete* and *efficient* query answering from all these heterogeneous sources. The system is able to store RDF(S) data and provides the ability to retrieve information using the standard query language for RDF, i.e., SPARQL. In addition, RDFS reasoning is supported following the RDFS rules given by our data model.

The envisioned architecture has been realized in a real system, called *Atlas*[1]. Atlas is a full-blown open source P2P system for the distributed storage and querying of RDF(S) data. We distinguish two kinds of users in our system architecture; resource/metadata *providers* and resource/metadata *consumers*. Providers expose resource descriptions to the network by inserting RDF(S) information to the system. Consumers want to discover resources by submitting SPARQL queries to the system. Any node in the network can accept a request from a provider or a consumer. All nodes in the system are equal and there is neither centralized control nor global knowledge.

The implementation of Atlas started in project OntoGrid[2] where it has been used for implementing a Semantic Grid service registry [75, 87]. The Atlas prototype developed in OntoGrid has been further refined in the Greek project "Peer- to-Peer Techniques for Semantic Web Services". In this project Atlas formed the core of a distributed service

[1] `http://atlas.di.uoa.gr`
[2] `http://www.ontogrid.net`

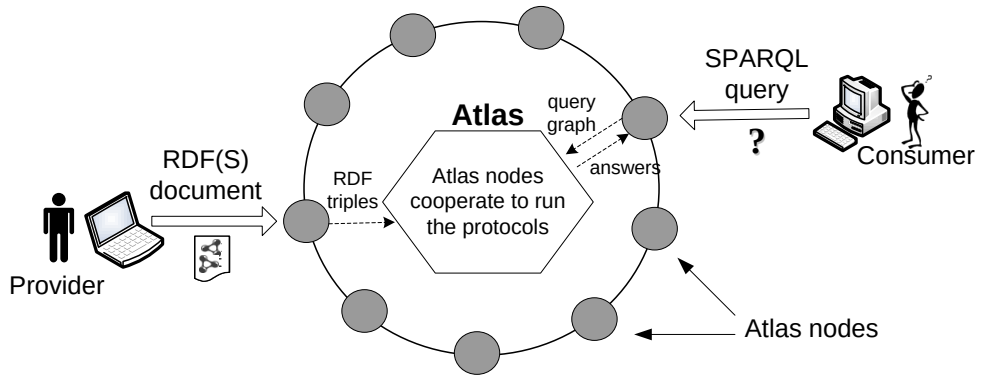


Figure 4.1: Atlas architecture

matchmaker developed for a subset of OWL-S [73]. Atlas was also used in project SemsorGrid4Env[3] where it was extended with the capability to store spatial metadata expressed in an appropriate extension of RDF that our group has defined [86]. Atlas is publicly available as open source under the LGPL license.

In the next sections we elaborate on the design of Atlas providing the architecture and API of the system. Additionally, we present an application scenario in a distributed digital library setting where the role of Atlas is demonstrated. Finally, we describe the basic protocols for indexing and retrieving RDF(S) data which are used in the algorithms we describe in the next chapters. Results of this chapter have been published in [70, 71].

4.1 The Atlas System

In this section we describe in detail Atlas, a P2P system for the distributed management of RDF(S) data. More specifically, we present the general architecture of Atlas, the architectural design of each Atlas node, the API supported by Atlas, and finally, an application scenario that demonstrates how Atlas can be used as a distributed repository of RDF(S) data.

4.1.1 Atlas architecture

A high level view of the Atlas architecture is shown in Figure 4.1. Nodes in Atlas are organized according to the Bamboo DHT [122] protocol. All nodes in the network are equal and there is neither centralized control nor global knowledge. Any node can

[3] `http://www.semsorgrid4env.eu/`

accept either a request for storing RDF(S) data in the system from a provider or a request for evaluating a SPARQL query from a consumer.

The provider can insert both RDF data and RDFS ontologies in the form of RDF documents which can be in an RDF/XML or N-Triple format. Each time a node receives a request for storing an RDF(S) document, it first decomposes it in RDF triples and then stores the RDF triples based on the indexing scheme we describe in Section 4.2.1. We should point out that in our architecture we adopt a generic approach where no global knowledge about the schema is required. We handle data and schema triples in a uniformly. In this way, our system can also handle scenarios with very big ontologies as well as scenarios with many different ontologies.

Alternatively, a user can be a consumer and pose a SPARQL query. Each time a node receives a SPARQL query request, it transforms it to a query graph as we defined in Section 3.3 and then, all Atlas nodes cooperate to find RDF(S) data that form the answer to the query. Query processing algorithms are described later in Chapter 6.

In the remainder of the thesis, we assume that the network is stable with no churn, i.e., network nodes do not join, leave voluntarily or fail in order to have clear semantics for the query processing results. Node failures and network dynamicity pose extra difficulties and require additional recovery algorithms. Our chosen underlying DHT, Bamboo DHT, offers many recovery mechanisms and can handle churn using various methods [122], but handling churn is out of the scope of our work.

4.1.2 Atlas node architecture

A higher level view of each node's architecture as implemented in Atlas is shown in Figure 4.2. On the top layer of the architecture, the API enables a user to provide resource descriptions by storing RDF(S) data in the network or to consume resource descriptions by posing a SPARQL query. The basic components of Atlas are the store and query processors.

Upon receiving a store request, the store processor of the node is responsible for decomposing the document into RDF(S) triples and initiating the protocol that distributes the triples in the network. The triples are indexed in various nodes of the network according to a specific indexing scheme explained later. If forward chaining is chosen as the reasoning scheme, the RDFS engine is responsible for making the appropriate inferences according to the ρdf inference rules. Then, the store processor distributes the inferred triples in the network as well. Every triple that arrives at the responsible node is stored in the local database of this node.

When a node receives a query request, the SPARQL translator transforms the

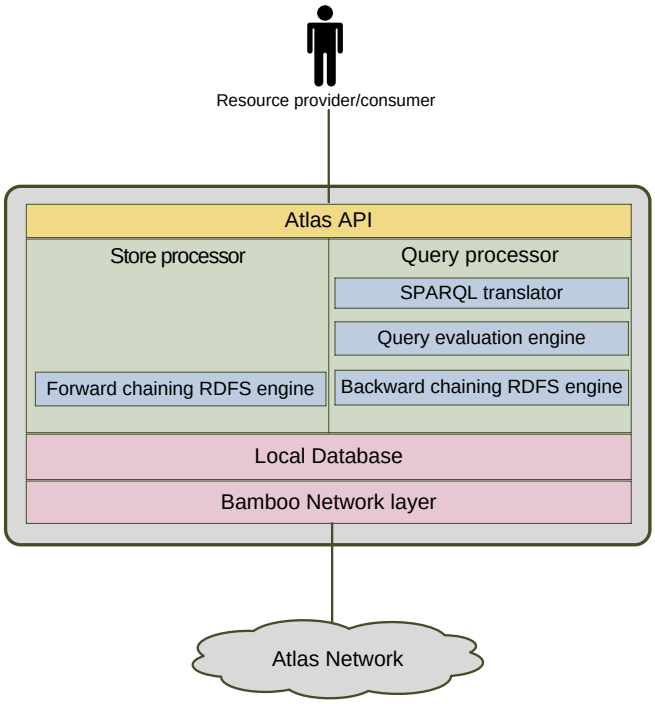


Figure 4.2: Atlas node architecture

query to an equivalent query graph. Then, the query evaluation engine is responsible for the distributed evaluation of the query. If backward chaining is the chosen reasoning scheme, the RDFS engine is also in charge of carrying out the distributed evaluation taking into account the *ρdf* inference rules.

Each node has a local database where both RDF data and RDFS descriptions are stored permanently on the disk. Both the store processor and the query processor are able to communicate with the local database of each node. In earlier versions of Atlas, the Berkeley DB [18] was used which is included in the Bamboo implementation and used by Bamboo for various database tasks. However, we found that this implementation is inefficient and have moved to SQLite[4], a lightweight relational database instead. However, Atlas could use any database as a permanent storage at each node as long as there is a jdbc driver to be able port it to our implementation. At the lower levels of the architecture lies the Bamboo network layer which is responsible for the communication among the nodes.

[4] `http://www.sqlite.org/`

4.1.3 Atlas API

In this section we present the API offered by Atlas so that a user is able to communicate with any node in an Atlas network to store RDF(S) data or pose SPARQL queries. A user can communicate to any available node in the network using the following functions provided by the Atlas API.

- `store(file, format)`: stores an RDF `file` given in a specific format in the Atlas network. The format of the file can be either RDF/XML or RDF/NTriple. Each RDF(S) document is decomposed into a collection of RDF(S) triples and these triples are indexed in various nodes of the Atlas network according to the storing protocol. When all triples are successfully indexed in the network, a confirmation message is returned to the user. Otherwise an error message is returned.

- `store(url, format)`: stores RDF(S) data in the Atlas network specified by a URL. Again the format of the data can be either RDF/XML or RDF/NTriple. When all triples are successfully indexed in the network, a confirmation message is returned to the user. Otherwise an error message is returned.

- `query(q)`: poses a SPARQL query `q` to the Atlas network. Query `q` should be a well-formed SPARQL query containing basic graph patterns as we defined in Chapter 3. Otherwise, an exception error is returned to the user. Each time a node poses a query, the network nodes cooperate to find RDF(S) data that form the answer to the query using algorithms we describe in the next chapters.

4.1.4 Application scenario

We can think of several application scenarios which feature Atlas as a distributed repository for storing and retrieving metadata describing resources: open linked data, content-sharing, Web/Grid service registries, distributed digital libraries and social networks. To demonstrate how Atlas can be used in such application scenarios, we consider the case of distributed digital libraries (DDLs). The use of Atlas to implement a distributed Grid service registry in the context of project OntoGrid has been discussed in [75]. In the distributed digital libraries scenario, a user poses a query (e.g., "I am interested in papers on the Semantic Web") and the distributed digital library returns matching resources.

Figure 4.3 gives a use case of Atlas using a distributed digital library application scenario. We assume a university with three geographically distributed campuses (e.g., Arts, Sciences and Medicine) and a local digital library in each campus (only the Science

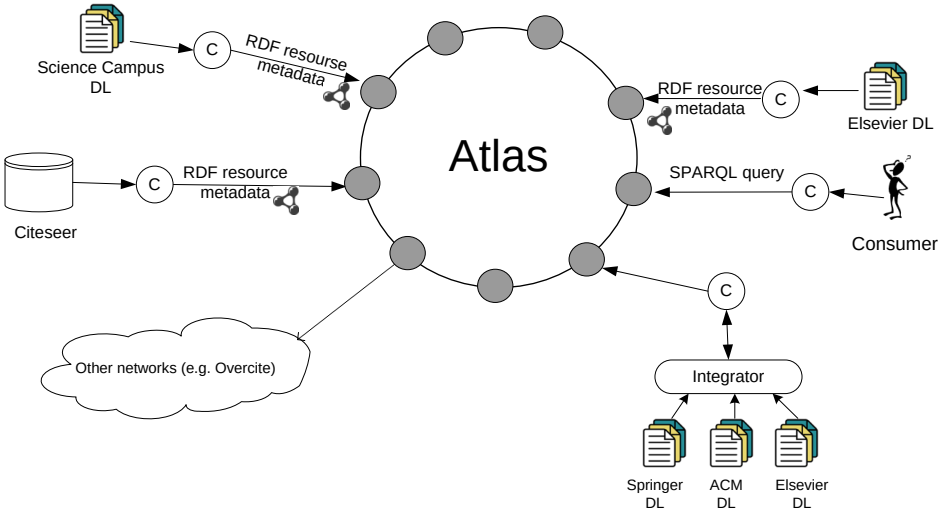


Figure 4.3: Using Atlas to implement a distributed digital library

DL is shown graphically in the figure). Each local digital library is a provider in terms of our architecture. Providers maintain an Atlas client which offers them an access point to the Atlas network. Providers use this client to publish RDF metadata about the resources they hold in the Atlas network. Consumers (e.g., users) can also deploy an Atlas client to pose queries about available resources. Various interesting extensions to this basic setting are possible as shown in Figure 4.3. For example, the university might be interested in making available to its students and staff, in a timely way, the content provided by other publishers (e.g., CiteSeer, ACM, Springer, Elsevier). Figure 4.3 shows how our architecture can be used to fulfill this requirement. A client is used to publish in the overlay network metadata about resources available in CiteSeer. The content of the ACM, Springer and Elsevier digital libraries is similarly made available. In this case, an integration layer is used to unify different digital libraries. We omit discussion of the relevant interoperability issues since they can be solved at the level of providers with well-known integration techniques. We also leave open the question of how to interoperate with other relevant P2P networks e.g., the DHT-based digital library OverCite [148].

The main components of the distributed digital library architecture shown in Figure 4.3 are: *Atlas nodes*, *Atlas clients* (denoted by C), *providers* and *consumers*. Providers own information sources that are part of the DDL, while consumers are interested in the DDL content. Atlas nodes form an overlay network that offers a robust, fault-tolerant

and scalable means for routing messages and managing DDL resource metadata encoded in RDF. Atlas clients are software components that enable providers and consumers to communicate with the Atlas network. Clients can connect to any Atlas node of the network through the Atlas API described previously.

4.2 Basic Protocols for Storing and Retrieving RDF Data in Atlas

In this section we describe the basic protocols for indexing and retrieving RDF data in a DHT network. These protocols form the basis of our system and is the key to the completeness of our algorithms. The algorithms that are based on these protocols are presented in the next chapters (Chapters 5, 6 and 7).

4.2.1 Indexing protocol

Nodes of the network can insert both RDF data and RDF Schema information in the form of RDF documents. Documents can be in an RDF/XML or N-Triple format. Whenever a node receives a request to store RDF(S) data through the `store` function offered by the API, it decomposes the document into RDF(S) triples and creates a request StoreRequest(*triples*). We call this node *store requester node*. Then, the triples are indexed in various nodes of the network.

In the following, we describe the basic protocol used for indexing RDF(S) triples in the network. We have adopted the triple indexing algorithm originally presented in [33] where each triple is indexed in the DHT *three times*. This algorithm is by now standard in DHT-based systems for RDF(S) stores. The hash values of the subject, property and object of each triple are used to compute the identifiers that indicate the nodes responsible for storing the triple. Whenever a node receives a request to store a set of triples, i.e., a request StoreRequest(*triples*), it sends three DHT Put(id, t) messages for each triple $t \in$ *triples*, using as key the subject, property and object respectively, and the triple itself as the item. The store requester node hashes each key using hash function SHA-1 [137] to create the identifier *id* that leads to the responsible node where the triple is stored. We call that node the *responsible node* for this key or identifier. Notice that when a node is responsible for a key which is an RDFS class name C (*responsible node for class* C), it will have in its local database all triples that contain class C either as a subject or as an object (class C cannot be a property).

Each node keeps its triples in its local database consisting of a single relation with

four columns (*triple* relation). The first three columns correspond to the three components of the triples, i.e., subject, predicate, object, while the fourth column indicates which of the three components is the key that led the triple to this node using a simple number.

Since an RDF(S) database is actually a graph, we exploit the fact that many of the triples share a common key (i.e., they have the same subject, property or object) and end up to be stored in the same node. So, instead of sending different Put messages for each triple, we group them in several lists $triples_i$ $(1 \leq i \leq k)$ based on the k distinguished keys that exist in the whole set of triples, hash these keys to obtain identifiers and send a multiPut$(id_i, triples_i)$ message for each identifier. The node responsible for the identifier id_i, which receives this message, stores in its local database all triples included in the list $triples_i$.

We handle data and schema triples in a uniform way. While other approaches, such as [42, 163, 166], require that every node in the system keeps all the RDFS triples, and in some cases in main memory, in our system we adopt a more generic approach where no global knowledge about the schema is required. RDFS triples are distributed in the same way that RDF triples are and both are stored in the local SQLite database of the relevant node. In this way, Atlas can also handle scenarios with very big ontologies as well as scenarios with many different ontologies, where other systems such as the above might not scale since their scalability will depend on the number of schema triples that can be stored locally at each node's main memory.

4.2.2 Querying protocol

In this section we describe the basic protocol used in Atlas to answer an atomic SPARQL query consisting of a *single* triple pattern `(s,p,o)`. Whenever a node receives a request for a query through the `query` function offered by the API, it decomposes the query to a triple pattern *tp* and creates a request QueryRequest(*tp*). We call this node *query requester node*.

When a node receives a query request for a triple pattern *tp*, i.e., a request QueryRequest(*tp*), it should decide to which node it should route the request to evaluate the query. The query requester node chooses a *key* from the triple pattern *tp* and hashes it to create the identifier that will lead to the appropriate node. The key is the constant part of the triple pattern. When there is more than one constant parts, the query requester node selects the keys in the order "subject, object, property" based on the fact that we prefer keys with lower selectivity and the reasonable assumption that subjects or objects have more distinct values than properties. This simple scheme suffices

for the purposes of simplicity. In Chapter 7, we study relevant issues in the context of conjunctive triple pattern queries using selectivities. At the destination node, all triples that contain this key will be found in the local database due to our indexing scheme. The triple pattern will be matched with these triples and the bindings of the triple pattern's variables will be returned to the query requester node. A more detailed algorithm for evaluating more complex SPARQL queries as they were defined in Section 3.3 is presented in Chapter 6.

Note that we do not deal with answering queries that are triple patterns with no constant parts. The answer to such queries is the whole database of triples stored in all the nodes of the network. These queries can be computed by a broadcasting algorithm but this is out of the scope of this work.

4.3 Summary

In this chapter we presented the design of Atlas, a full-blown open source system for the distributed storage and querying of RDF and RDFS data. We described the architecture of the system as well as the architecture of each Atlas node and provided the offered API. We demonstrated how Atlas can be used in a distributed digital library application scenario. In addition, we provided the basic protocols for indexing and retrieving RDF(S) data which are used to the algorithms we describe in the next chapters.

In the next chapters we describe the algorithms we have developed in Atlas for distributed RDFS reasoning, SPARQL query processing and RDF query optimization. The next chapter starts with the RDFS reasoning techniques we have developed in Atlas. We study both theoretically and experimentally our algorithms and showcase the results.

Chapter 5

Distributed RDFS Reasoning

With the Semantic Web brought into being the last couple of years, there is an urgent need only for dealing with a huge amount of distributed data expressed in RDF, but also for being able to reason with it. Reasoning algorithms have been widely studied in the past in the areas of logic and artificial intelligence. Two important reasoning techniques are *forward chaining* and *backward chaining*. Forward chaining works in a bottom-up fashion. It starts from a given dataset and using the inference rules produces new data that is entailed by the given dataset. Backward chaining works in a top-down manner. It starts from a goal and finds a proof of this goal using the given dataset and the inference rules. In this chapter we study such reasoning algorithms for RDFS.

Previous work on *centralized* RDF stores has considered forward chaining, backward chaining and hybrid approaches to implement RDFS reasoning and query processing [7, 30, 52, 168]. In the forward chaining approach, new RDF statements are exhaustively generated from the asserted ones. In contrast, a backward chaining approach only evaluates RDFS entailments on demand, i.e., at query execution time. Intuitively, we expect that a forward chaining approach has minimal requirements during query answering, but needs a significant amount of storage for all the inferred data. In contrast, the backward chaining approach has minimal storage requirements, at the cost of an increase in query response time. There is a time-space trade-off between these two approaches [149], and only by knowing the query and update workload of an application, we can determine which approach would suit it better. This trade-off has never been studied in detail in a *distributed Internet-scale* scenario, and this is one of the challenges that we undertake in this chapter using the technology of DHTs.

The organization of this chapter is as follows. In Section 5.1 we present how the forward chaining algorithm can be implemented in a DHT and give formal proofs for the

correctness of the algorithm. In addition, in Sections 5.1.3 and 5.1.4 we discuss the issue of redundant RDF triples generation in forward chaining algorithms and the role of the semi-naive evaluation in the distributed environment of a DHT, respectively. Section 5.2 presents the backward chaining algorithm with its correctness proofs, while Section 5.3 describes the algorithm based on the magic sets transformation technique [17]. In Section 5.4, we give an analytical cost model of our algorithms, while in Section 5.5 we present the results of our experimental evaluation. Finally, Section 5.6 summarizes the chapter. Results of this chapter have been published in [74] and are under review in [69].

5.1 Distributed Forward Chaining

In this section, we describe our forward chaining algorithm and its implementation in a DHT. The general idea of forward chaining (FC) is that all inferred triples are precomputed and stored in the network a priori. Each time a node receives a triple to be stored in its local database, it computes all inferred triples and sends them to the network to be stored. The algorithm uses the Datalog rules presented in Table 3.2.

5.1.1 Algorithm description

Let us now introduce the notation that will be used in the algorithms description. Keyword **event** precedes every event handler for handling messages, while keyword **procedure** declares a procedure. In both cases, the name of the handler or the procedure is prefixed by the node identifier in which the handler or the procedure is executed. Local procedure calls are not prefixed with the node identifier. Keywords **sendto** and **receive** declare the message that we want to send to a node with known either its identifier (thus DHT routing will be used) or the IP address (thus the node is immediately contacted), and the message we receive from a node respectively.

Our forward chaining algorithm, called FC*, is activated every time an RDF(S) triple is inserted in the network. As explained in Section 4.2.1, whenever a node receives a request StoreRequest to store a set of triples G, it sends three DHT Put messages for each triple, using as key the subject, property and object respectively, and the triple itself as the item. In the case of FC*, instead of using a Put message, the store requester node sends a message StoreMsg and the algorithm is activated. Algorithm 1 shows in pseudocode how FC* works. Suppose a StoreMsg(id, t, k, inf) request arrives at node n which is responsible for the identifier id and a new triple t should be stored in the local database of n. k is a byte that indicates the key that led triple t to this node

Algorithm 1: FC*: Forward chaining algorithm

```
1  event n.StoreMsg(id, t, k, inf)
2      localTriples={t} ∪ getTriplesFromDB (t.key);
3      newtriples=Infer (localTriples);
4      forall the t' of newtriples ∉ infTriples && ∉ localTriples do
5          id1 =Hash (t'.subject);
6          id2 =Hash (t'.property);
7          id3 =Hash (t'.object);
8          sendto id1.StoreMsg(id1, t', 1, true);
9          sendto id2.StoreMsg(id2, t', 2, true);
10         sendto id3.StoreMsg(id3, t', 3, true);
11         infTriples.add(t');
12     end
13     InsertToDB (t, inf);
14 end
```

(we use $k = 1$ for subject, $k = 2$ for property and $k = 3$ for object). Keeping the key in one byte variable reduces the size of the message; the actual string value which can be found from the triple itself. inf is a boolean value that indicates whether t is an inferred triple or not. First, node n retrieves from the local database all triples that contain the key $t.key$ of triple t either as a subject, property or object and puts them in list $localTriples$ together with t. Then, it computes the triples inferred from this list according to the Datalog rules of Table 3.2 using local function Infer($localTriples$). Function Infer($localTriples$) assigns the triples which have originated from the initial RDF(S) graph to the edb relation `triple` and the triples which have been inferred from an inference rule to the idb relation `newTriple`.[1] It outputs the idb relation `newTriple` by matching the triples with the antecedent of the rules of Table 3.2 in a data-driven manner. The new facts of relation `newTriple` generated by function Infer form the inferred triples. Each node holds a list $infTriples$ with all inferred triples that it has computed, so that it can check when it reaches a fixpoint where no new triples can be generated. For all newly inferred triples, three identifiers are created based on the subject, property and object of each triple, and three StoreMsg requests are sent to the network. The initial triple t is stored in node's n local database using local function InsertToDB. The algorithm terminates when all nodes have reached a fixpoint.

Invoking FC* every time new triples are stored in the network allows us to compute the closure of the stored triples under the *mrdf* inference rules (we prove this formally

[1] In the local database of each node there is information of whether a triple is inferred or not.

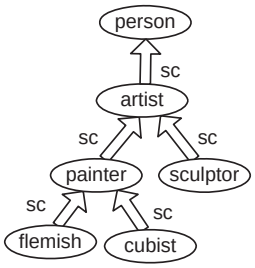


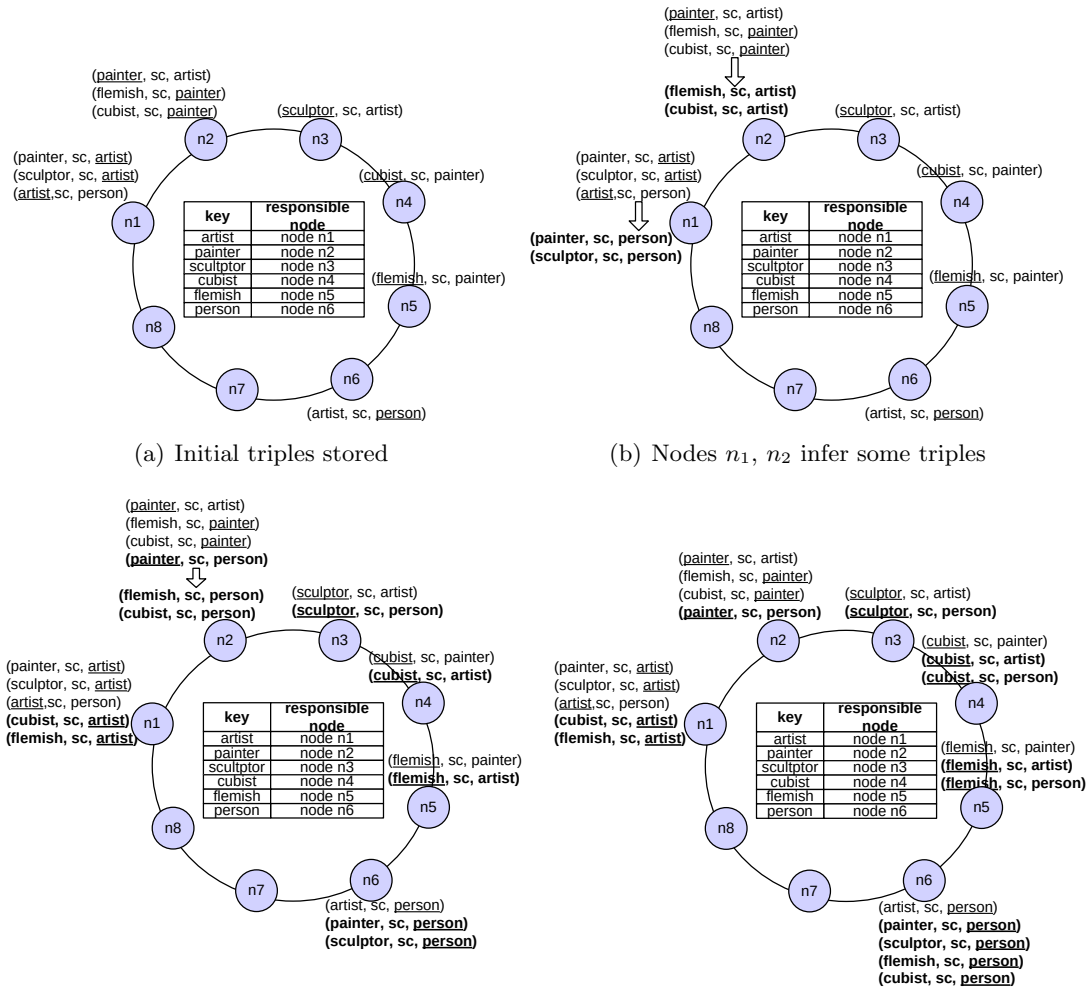
Figure 5.1: Example RDF(S) class hierarchy

below). After FC* has terminated, query evaluation of one triple pattern can be performed exactly as described in Section 4.2.2. The query request is routed to the node that is responsible for the key of the triple pattern and all triples matching this triple pattern are found locally at its database.

FC* runs in the same way when messages MultiPut are used to group triples. Instead of sending a message MultiPut, the store requester node creates a message MultiStoreMsg$(id, triples, k, inf)$ which takes as an input a list of triples. All triples in list *triples* share the same key whose position is kept in variable k. The only difference is that k now refers to the position of the key in the first triple of the list.

Figure 5.1 depicts a small RDFS class hierarchy of the cultural domain [81]. In Figure 5.2, we demonstrate an example of how FC* works based on this RDFS hierarchy. Triples that are not in bold are initially inserted in the network. The key of each triple that led to a specific node is underlined. Figure 5.2(a) shows the initial triples that are indexed in the nodes of the network. The key of each triple that led to a specific node is underlined. Nodes n_1 and n_2 infer two triples each using rules 1 and 4 of Table 3.2 (Figure 5.2(b)). Inferred triples are shown in bold. These triples are sent to be stored to the corresponding nodes of the network. Then, node n_2 will infer two more triples by considering the already inferred triple `(painter, sc, person)` and the two initial triples stored in its local database using rule 4 of Table 3.2 (Figure 5.2(c)). These two triples are finally sent to be stored at nodes n_4, n_5 and n_6. The final state of the nodes' databases is depicted in Figure 5.2(d).

Suppose now that a user submits the query "Find all the subclasses of class artist" to a node of the network. This query can be expressed as the triple pattern `(?X, sc, artist)` and will be routed to node n_1, which is responsible for the key `artist`. The answer will be formed at this node since all triples will be found at its local database.



(a) Initial triples stored

(b) Nodes n_1, n_2 infer some triples

(c) Node n_2 infers some more triples

(d) Final triples stored

Figure 5.2: FC* in operation

A simpler forward chaining algorithm. Initially, we had designed a simple forward chaining algorithm, which we called FC and presented in [74]. The algorithm FC runs similarly with FC*. It is activated every time an RDF(S) triple is indexed in a node of the network and then, new triples are inferred from the RDFS rules and sent to the network. The difference with FC* is that FC uses a slightly different set of rules, i.e., rules of Table 3.2 but without any recursion. The body of all rules contain only the edb relation `triple`. Triples originated from the initial RDF(S) graph and inferred triples are not distinguished by any parameter in the message StoreMsg. All triples that arrive at a node are assigned to the edb relation `triple`. In this way,

many redundant triples are generated in different nodes of the network. Notice that, if FC was used in the previous example, node n_1 would have also inferred the same triples that node n_2 produced in Figure 5.2(c), causing unnecessary traffic and more processing load to nodes n_4, n_5 and n_6. Experimental results in Section 5.5.2 showcase the magnitude of this phenomenon.

5.1.2 Termination, soundness and completeness

First, we prove that FC* terminates.

Theorem 2 (Termination)**.** *Algorithm FC* terminates.*

Sketch. Nodes in the network execute the for loop of lines 4-12 of FC* a finite number of times. The loop is executed a finite number of times since there is a finite upper bound on the number of triples entailed by a given finite graph G. The loop propagates inferred triples in other nodes of the network hence FC* terminates. □

In the following, we prove that algorithm FC* is sound and complete. At this point, we make the following simplifying assumptions regarding the overlay network. We assume a stable network with no churn, i.e., network nodes do not join, leave voluntarily or fail, where messages are always delivered after a finite amount of time. Although the Bamboo DHT has many recovery mechanisms and can handle churn using various methods [122], handling churn is out of the scope of our work.

By *sound* we mean that if H is the RDF(S) graph produced by the FC* algorithm and stored in the network, then $G \models H$ where G is the initial graph. By *complete* we mean that if G is the RDF(S) graph initially stored in the network, H any graph and $G \models H$, then all triples of H will be stored in the network after the completion of FC*.

In the proofs of soundness for the forward and backward chaining algorithms, we use the notion depth of an appropriate kind of finite tree that captures the relevant computation. We define this notion as usual. The *depth of a node* in a tree is the length of the path from the root to this node. The *depth of a tree* is the maximum depth of a node in the tree.

An execution of FC* on top of a DHT can be modeled using the following notion of computation tree.

Definition 5.1.1. *A* computation tree *is a finite tree with the following properties:*

(i) Every node is of the form $a_i : (H^i_{old}, H^i_{new})$ where a_i is a DHT node and H^i_{old}, H^i_{new} are non-empty sets of triples.

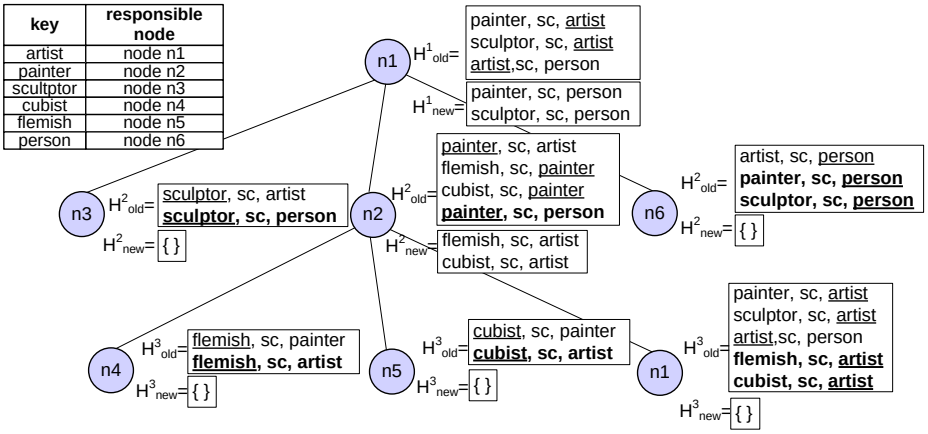


Figure 5.3: Computation tree of the DHT node n1

(ii) For every node $a_i : (H^i_{old}, H^i_{new})$ *we have the following:*

- H^i_{old} *is a set of triples stored at the local database of node* a_i*.*
- H^i_{new} *is the set of triples computed at node* a_i *during some step of FC* by applying some ρdf rules in parallel.*

(iii) For every child $a_{i+1} : (H^{i+1}_{old}, H^{i+1}_{new})$ *of node* a_i*, we have that* $H^i_{new} \cap H^{i+1}_{old}$ *is non-empty and is the set of new triples produced at node* a_i *during some step of FC* and sent to node* a_{i+1} *where FC* continues.*

A computation tree offers a nice pictorial representation of the computation of FC*. Figure 5.3 shows the computation tree for the example shown in Figure 5.2. The root of the tree corresponds to the activation of FC* when a message StoreMsg arrives at a DHT node. The nodes of the tree at increasing depths allow us to understand FC* proceeding in rounds although no such strong assumption is used by the algorithm.

Theorem 3 (Soundness)**.** *Let* G *be the initial graph stored in the network and* H *the set of triples in the local databases of nodes after FC* has terminated. Then it holds that* $G \models H$*.*

Proof. The proof is by induction on d, the depth of the computation tree representing the execution of FC*.

Base case: $d = 1$ (the root is $d = 0$). In this case, FC* runs at the network node represented by the root of the tree, produces k new sets of triples and sends them to k nodes. FC* runs again at these k nodes but no new triples are computed. A sequence of graphs (*mrdf* proof according to Definition 3.2.2) that shows $G \vdash_{mrdf} H$ can easily

be constructed. The first element of the sequence is G followed by k subsequences representing the k sets of triples mentioned above (the order of the subsequences does not matter). Each of the k sets of triples can be represented by as many proof steps as the number of triples it contains. Each step is the result of the application of a *ρdf* rule to the set of triples produced in the immediately previous proof step.

Inductive step: We assume that the theorem holds for executions of FC* with computation trees of depth d and we will show that the theorem holds for executions of FC* with computation trees of depth $d+1$.

We take the computation tree T of depth $d+1$ corresponding to the execution of FC* on the stored graph G and prune all the nodes at depth $d+1$. The resulting tree T' has depth d and the theorem holds for the corresponding execution of FC*. Hence, if H' is the set of triples in the local databases of the DHT nodes as represented by the tree T' then $G \vdash_{mrdf} H'$. Now we can continue the proof of H' from G by adding the steps corresponding to the nodes we pruned from T to arrive at a proof H. This is done as in the base case.

Since we have proved $G \vdash_{mrdf} H$, Theorem 1 of Section 3.2.1 gives us that $G \models H$. □

Theorem 4 (Completeness)**.** *Let G and H be mrdf graphs and graph G is stored in the network. If $G \models H$, then the triples of graph H will be also stored in the network when algorithm FC* terminates.*

Proof. Using Theorem 1 and the fact that $G \models H$, we have that $G \vdash_{mrdf} H$. We will prove the result using induction on k, the number of *mrdf* proof steps that show $G \vdash_{mrdf} H$. Using the notation of Definition 3.2.1, we assume that $P_0 = G$ and $P_k = H$.

Base case ($k = 1$): H is derived from G by the application of a single rule r of the deductive system of Table 3.1.

- If r is rule (1b) then $H \subseteq G$ and thus, all triples of H will be stored in the network.
- If r is one of the rules (2)-(4), then there is an instantiation $\frac{R}{R'}$ of rule r such that $R \subseteq G$ and $H = G \cup R'$. G is stored in the network and hence we need to show that triple R' will also be stored. The instantiation of r will happen at the node where triples which match the antecedent of rules (2)-(4) (i.e., R) will meet. If we check Table 3.2, we observe that the antecedent triples of rules 2-7 always have a common element. Triples are indexed three times based on three identifiers, namely the hash values of their subject, property and object. Therefore, triples

with a common element will meet at the node responsible for the identifier of this common element where the antecedent R will be matched. Then the triple R' will be generated by FC* using local function Infer and will be stored in the network. Thus, all triples included in H will be stored in the network.

Inductive step: We assume that the result holds for $k-1$.We will show that it holds for k.

Let us consider G and H such that $G \vdash_{mrdf} H$ in k proof steps. Using the notation of Definition 3.2.1, H can be proved from P_{k-1} by the application of a single rule r of the deductive system of Table 3.1. The rest of the proof is similar to the one for the base case.

- If r is rule (1b), then $H \subseteq P_{k-1}$ and following the hypothesis of the induction all triples of P_{k-1} have been stored in the network. Consequently, all triples of H are stored in the network.

- If r is one of the rules (2)-(4), there will be an instantiation $\frac{R}{R'}$ such that $R \subseteq P_{k-1}$ and $H = P_{k-1} \cup R'$. According to the hypothesis of the induction all triples of P_{k-1} and therefore all triples of R were stored in the network according to our indexing scheme. Hence, if the rule is one of the rules (2)-(4), the instantiation of the rule will happen at the node where the triples of R will meet. Again since triples in the antecedent of rules (2)-(4) always have a common element, these triples will meet at the node responsible for this common element. Then, triple R' will be generated using local procedure Infer and will be stored in the network. Consequently, all triples of H will be stored in the network.

□

5.1.3 Redundant triple generation in FC*

Several authors have pointed out recently the generation of many redundant triples in forward chaining algorithms for RDFS reasoning, and especially in distributed ones [113, 163, 166]. The drawback of duplicate triple generation can be greater and more harmful in a distributed system. Duplicate triples are generated at different nodes and sent through the network causing an enormous amount of traffic as well as unnecessary load to the nodes. [113] treats the reasoning process at each node as a black box and does not elaborate on the rule set used. However, the authors do observe a big rate of duplicate triples and offer an elimination process to remove the duplicate triples that

have been generated. In [166], the authors present results showing that as the number of processes grows the number of duplicate triples also increases and they especially refer to rule $rdfs9$ (i.e., rule (3b) of Table 3.1) for generating duplicate triples. [163] also provides solutions in the MapReduce framework to avoid duplicate triples and eliminate them if necessary. However, none of these works elaborate on the cause of redundant triples.

An important source of redundant triples which, to the best of our knowledge, has not been discussed in the literature before, is the recursive rules of Table 3.1. When the rules of Table 3.1 are used as they are or, in our case, translated into Datalog in the obvious way, they give rise to bilinear recursion. *Bilinear rules* are rules which have two occurrences of a recursive idb relation in their body and hence contain double recursion. Rules with double recursion can produce a large number of duplicate triples even in a centralized environment [6]. Let us demonstrate this with two simple examples.

Example 5.1.1. *Assume that our initial graph G contains three triples:*
$t_1 =$`(a, sc, b)`
$t_2 =$`(b, sc, c)`
$t_3 =$`(c, sc, d)`
These triples form a small RDFS class hierarchy. Using rule (3a) of the minimal deductive system of [101], we infer the following triples:
$t_1, t_2 \Rightarrow t_4 =$`(a, sc, c)`
$t_2, t_3 \Rightarrow t_5 =$`(b, sc, d)`
$t_3, t_4 \Rightarrow t_6 =$`(a, sc, d)`
$t_1, t_5 \Rightarrow t_6 =$`(a, sc, d)`
Therefore, triple t_6 is inferred twice.

In the following example, we show how the generation of duplicate triples can be prevented if we use the linear rules we presented in Table 3.2.

Example 5.1.2. *Assume that we have the same initial graph G of Example 5.1.1, but now we use rules 1 and 4 of Table 3.2. From applying rule 1, we add 3 assertions with idb relation* `newTriple`*:*
$t'_1 =$`newTriple(a, sc, b)`
$t'_2 =$`newTriple(b, sc, c)`
$t'_3 =$`newTriple(c, sc, d)`
Using rule 4 of Table 3.2, we infer the following triples:
$t_1, t'_2 \Rightarrow t'_4 =$`newTriple(a, sc, c)`
$t_2, t'_3 \Rightarrow t'_5 =$`newTriple(b, sc, d)`

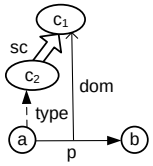


Figure 5.4: Redundancy in RDF(S) graphs

$t_1, t'_5 \Rightarrow t'_6 =$*newTriple(a, sc, d)*
Notice that the pair of triples t_3 and t'_4 cannot satisfy any of the bodies of rules of Table 3.2 and hence triple t'_6 is inferred only once.

In one of our previous implementations of forward chaining, called FC in [74], we used bilinear rules and made no distinction between explicit and inferred triples. Every time a new triple t was generated, it was sent to be stored at the responsible nodes without specifying that t is an inferred triple. The nodes that received t were producing new triples from all triples stored locally without taking into consideration whether they were previously inferred or not. For instance, in the example of Figure 5.2, if we had used our previous algorithm FC [74], node n_1 in Figure 5.2(c) would have also inferred the same triples that node n_2 produced causing unnecessary traffic and more processing load to nodes n_4, n_5 and n_6.

Another source of triple generation redundancy is when RDF data is derived using different entailment rules. In this case, techniques such as the above cannot prevent the generation of redundant triples. For example, domain and range statements can interact with other statements in many ways to produce redundant triples. The following example shows one such case where a triple generation redundancy can occur.

Example 5.1.3. *In the RDF(S) graph G of Figure 5.4, triple (a, type, c_1) can be inferred from triples (a, p, b), (p, dom, c_1) using rules 1 and 6 of Table 3.2. The same triple can also be derived from triples (a, type, c_2), (c_2, sc, c_1) using rules 1 and 5.*

5.1.4 Semi-naive evaluation in FC*

In centralized systems, semi-naive bottom-up evaluation [160] avoids inferring redundant facts in recursive rule execution. This can be accomplished by computing new facts using as input only base facts and new facts derived in the last iteration. Base case rules such as rule 1 of Table 3.2 are used only at the first iteration of the semi-naive evaluation. We could easily rewrite the rules of Table 3.2 so that semi-naive evaluation

can be performed. For example, rules 1 and 2 of Table 3.2 would be rewritten as follows:
Δ`newTriple`^{1}`(?X, ?P, ?Y) :- triple(?X, ?P, ?Y).`
Δ`newTriple`$^{i+1}$`(?X, sp, ?Y) :- triple(?X, sp, ?Z),` Δ`newTriple`^{i}`(?Z, sp, ?Y).`

In this way, new triples generated in iteration i (Δ`newTriple`^{i}) are used as input together with base triples to generate new triples in iteration $i+1$ (Δ`newTriple`$^{i+1}$) which, in turn, will be used in the next iteration. The rest of the rules can be rewritten similarly.

Our FC* algorithm can benefit from a semi-naive evaluation to avoid the inference of redundant triples *locally* at each node. In addition, list $infTriples$ in FC*, which holds all inferred triples to avoid sending redundant triples in the network and to determine fixpoint, is not required in this case. Local fixpoint can be reached when a node does not produce any more triples using the rewritten rules. Therefore, a semi-naive evaluation in FC* can reduce local processing load at each node.

In a distributed environment, checking completion of an iteration is a very difficult task and can be realized by either the coordination among the nodes of the network or by keeping a global iteration counter at one node and assume that nodes work in synchronization. However, the first approach is very expensive in terms of communication, while the second one violates the principle of P2P systems about global control. One solution followed in in [90] is to make the notion of iteration local at each node.

Using this technique, distributed semi-naive evaluation in our case can be realized as follows. Each inferred triple that arrives at a node is assumed to have been computed in the previous round and is inserted in the idb relation Δ`newTriple`^{i}. Therefore, rewritten rules are applied by assigning the local triples which originate from the initial RDF(S) graph to the edb relation `triple`, and inferred triples that arrive at the node to Δ`newTriple`^{i} so that they will be used at the next iteration. Rule 1 is applied only when a node receives a triple that originates from the initial RDF(S) graph.

However, the above way of performing distributed semi-naive evaluation applies only when bottom-up execution of the rules of Table 3.2 starts *after* all triples of the RDF(S) graph have been stored in the system. This is typically the case when the computation of the RDFS closure of a given set of triples is required. The way we defined FC* in Section 5.1 is more flexible since the rules are fired and RDFS closure is computed every time a new triple is sent to the network. Thus, if we introduce semi-naive evaluation in FC*, there is the possibility that some inferences are missed due to network delays. For example, assume that G is the initial graph and the message that contains triple `(a, sc, b)`$\in G$ has been delayed and arrives at the node responsible for key `a` after the inferred triple `(i, type, a)`$\notin G$. Then, the node will retrieve from

its local database the triples which have originated from the initial RDF(S) graph to assign them to the edb relation `triple`. Hence, it will not consider the inferred triple `(i, type, a)` and triple `(i, type, b)` will not be inferred. On the other hand, the original FC* would have retrieved from its local database all triples that contain value `a` regardless of whether they are inferred or not. We have implemented a version of FC* that uses semi-naive evaluation and we discuss its performance in Section 7.6.

5.2 Distributed Backward Chaining

In this section we describe how a backward chaining algorithm can be implemented in the distributed environment of a DHT. In contrast to the data driven nature of forward chaining, backward chaining starts from the given query and tries to find rules that can be used to derive answers.

5.2.1 Adorned rules

Given that we are using a Datalog version of the *ρdf* inference rules, the challenge here is to construct an algorithm that can process recursive Datalog rules in a distributed environment such as DHTs. To achieve this, it is helpful to transform the Datalog rules into a set of *adorned rules* that indicate which variables are bound and which are free. This is useful for finding a good order in which the predicates of the rule bodies should be evaluated. First, we rewrite the Datalog rules of Table 3.2 as shown in Table 5.1. This allows us to compare our backward chaining algorithm (BC) with the entailment algorithm presented in [101] (see Section 5.2.2 below). In addition, we will use this set of rules for the magic sets transformation technique described later in Section 5.3. Using the rules of Table 3.2 would also be possible in BC without changing the algorithm.

In Table 5.1 we use the edb relation `triple` and the idb relations `subClass`, `subProperty`, `type` and `newTriple`. Edb relation `triple` denotes all triples that are locally stored, while the idb relations denote the various kinds of triples that can be inferred. The difference with Table 3.2 is that now `newTriple` is not the only idb relation used to store inferred triples. When the property of a triple or triple pattern is equal to `rdfs:subPropertyOf` then relation `subProperty` is used, when the property is equal to `rdfs:subClassOf` then relation `subClass` is used, and when the property is equal to `rdf:type` then relation `type` is used. In any other case, the predicate `newTriple` is used. Rules 1, 3, 5 and 7 assign the triples found locally to the idb relations `subProperty`, `newTriple`, `subClass` and `type` respectively depending on the

Table 5.1: *ρdf* inference rules in Datalog (2nd version)

Rule	Head	Body
1	`subProperty(?X, ?Y)`	`triple(?X, sp, ?Y)`
2	`subProperty(?X, ?Y)`	`triple(?Z, sp, ?Y), subProperty(?X, ?Z)`
3	`newTriple(?X, ?P, ?Y)`	`triple(?X, ?P, ?Y)`
4	`newTriple(?X, ?P, ?Y)`	`triple(?X, ?P1, ?Y), subProperty(?P1, ?P)`
5	`subClass(?X, ?Y)`	`triple(?X, sc, ?Y)`
6	`subClass(?X, ?Y)`	`triple(?Z, sc, ?Y), subClass(?X, ?Z)`
7	`type(?X, ?Y)`	`triple(?X, type, ?Y)`
8	`type(?X, ?Y)`	`type(?X, ?Z), triple(?Z, sc, ?Y)`
9	`type(?X, ?Y)`	`newTriple(?X, ?P, ?Z), triple(?P, dom, ?Y)`
10	`type(?X, ?Y)`	`newTriple(?Z, ?P, ?X), triple(?P, range, ?Y)`

property of the triple. These rules together are equivalent to rule 1 of Table 3.2. Rules 2, 4 ,6, 8, 9, and 10 of Table 5.1 are equivalent with rules 2, 3, 4, 5, 6 and 7 of Table 3.2 respectively.

We extend the concept of rule adornment from Datalog query processing [161] in order to exploit the distributed philosophy of DHTs. As already mentioned, in order to evaluate a triple pattern, a *key* has to be computed and then hashed to create the identifier that will lead to the responsible node. We compute this key by choosing a constant part of the triple pattern in the order subject, object, property. Therefore, the corresponding predicate of the triple pattern has an argument that not only is bound, but it is also the *key* that led to the responsible node.

Definition 5.2.1. *An* adornment *of a predicate p with n arguments is an ordered string a of k's, b's and f's of length n, where k indicates a bound argument which is also a key, b indicates a bound argument which is not the key, and f a free argument.*

Following this definition, an adorned predicate p^a indicates which argument of p is the key, which ones are bound and which are free. Table 5.2 shows all possible adornments of the rules presented in Table 5.1 based on the indexing scheme of a triple pattern explained before.

Each adorned rule of Table 5.2 covers one possibility of using the corresponding rule of Table 5.1 in our backward chaining algorithm by additionally encoding run-time information about key, bound and free arguments. All possibilities that can actually take place are covered in Table 5.2. Since we do not allow queries where the triple pattern does not contain any constant values, note that there is no adorned predicate

Table 5.2: Adorned *ρdf* inference rules

Rule	Head	Body
1a	$\texttt{subProperty}^{kf}\texttt{(?X, ?Y)}$	$\texttt{triple}^{kbf}\texttt{(?X, sp, ?Y)}$
1b	$\texttt{subProperty}^{fk}\texttt{(?X, ?Y)}$	$\texttt{triple}^{fbk}\texttt{(?X, sp, ?Y)}$
1c	$\texttt{subProperty}^{kb}\texttt{(?X, ?Y)}$	$\texttt{triple}^{kbb}\texttt{(?X, sp, ?Y)}$
1d	$\texttt{subProperty}^{ff}\texttt{(?X, ?Y)}$	$\texttt{triple}^{fbf}\texttt{(?X, sp, ?Y)}$
2a	$\texttt{subProperty}^{kf}\texttt{(?X, ?Y)}$	$\texttt{triple}^{kbf}\texttt{(?X, sp, ?Z)}$, $\texttt{subProperty}^{ff}\texttt{(?Z, ?Y)}$
2b	$\texttt{subProperty}^{fk}\texttt{(?X, ?Y)}$	$\texttt{subProperty}^{ff}\texttt{(?X, ?Z)}$, $\texttt{triple}^{fbk}\texttt{(?Z, sp, ?Y)}$
2c	$\texttt{subProperty}^{kb}\texttt{(?X, ?Y)}$	$\texttt{triple}^{kbf}\texttt{(?X, sp, ?Z)}$, $\texttt{subProperty}^{bf}\texttt{(?Z, ?Y)}$
2d	$\texttt{subProperty}^{ff}\texttt{(?X, ?Y)}$	$\texttt{subProperty}^{ff}\texttt{(?X, ?Z)}$, $\texttt{triple}^{fbf}\texttt{(?Z, sp, ?Y)}$
3a	$\texttt{newTriple}^{kff}\texttt{(?X, ?P, ?Y)}$	$\texttt{triple}^{kff}\texttt{(?X, ?P, ?Y)}$
3b	$\texttt{newTriple}^{kbf}\texttt{(?X, ?P, ?Y)}$	$\texttt{triple}^{kbf}\texttt{(?X, ?P, ?Y)}$
3c	$\texttt{newTriple}^{kbb}\texttt{(?X, ?P, ?Y)}$	$\texttt{triple}^{kbb}\texttt{(?X, ?P, ?Y)}$
3d	$\texttt{newTriple}^{ffk}\texttt{(?X, ?P, ?Y)}$	$\texttt{triple}^{ffk}\texttt{(?X, ?P, ?Y)}$
3e	$\texttt{newTriple}^{fbk}\texttt{(?X, ?P, ?Y)}$	$\texttt{triple}^{fbk}\texttt{(?X, ?P, ?Y)}$
3f	$\texttt{newTriple}^{fkf}\texttt{(?X, ?P, ?Y)}$	$\texttt{triple}^{fkf}\texttt{(?X, ?P, ?Y)}$
4a	$\texttt{newTriple}^{kff}\texttt{(?X, ?P, ?Y)}$	$\texttt{triple}^{kff}\texttt{(?X, ?P1, ?Y)}$, $\texttt{subProperty}^{ff}\texttt{(?P1, ?P)}$
4b	$\texttt{newTriple}^{kbf}\texttt{(?X, ?P, ?Y)}$	$\texttt{triple}^{kff}\texttt{(?X, ?P1, ?Y)}$, $\texttt{subProperty}^{fb}\texttt{(?P1, ?P)}$
4c	$\texttt{newTriple}^{kbb}\texttt{(?X, ?P, ?Y)}$	$\texttt{triple}^{kfb}\texttt{(?X, ?P1, ?Y)}$, $\texttt{subProperty}^{fb}\texttt{(?P1, ?P)}$
4d	$\texttt{newTriple}^{ffk}\texttt{(?X, ?P, ?Y)}$	$\texttt{triple}^{ffk}\texttt{(?X, ?P1, ?Y)}$, $\texttt{subProperty}^{ff}\texttt{(?P1, ?P)}$
4e	$\texttt{newTriple}^{fbk}\texttt{(?X, ?P, ?Y)}$	$\texttt{triple}^{ffk}\texttt{(?X, ?P1, ?Y)}$, $\texttt{subProperty}^{fb}\texttt{(?P1, ?P)}$
4f	$\texttt{newTriple}^{fkf}\texttt{(?X, ?P, ?Y)}$	$\texttt{newTriple}^{fff}\texttt{(?X, ?P1, ?Y)}$, $\texttt{triple}^{fbk}\texttt{(?P1, sp, ?P)}$
5a	$\texttt{subClass}^{kf}\texttt{(?X, ?Y)}$	$\texttt{triple}^{kbf}\texttt{(?X, sc, ?Y)}$
5b	$\texttt{subClass}^{fk}\texttt{(?X, ?Y)}$	$\texttt{triple}^{fbk}\texttt{(?X, sc, ?Y)}$
5c	$\texttt{subClass}^{kb}\texttt{(?X, ?Y)}$	$\texttt{triple}^{kbb}\texttt{(?X, sc, ?Y)}$
5d	$\texttt{subClass}^{ff}\texttt{(?X, ?Y)}$	$\texttt{triple}^{fbf}\texttt{(?X, sc, ?Y)}$
6a	$\texttt{subClass}^{kf}\texttt{(?X, ?Y)}$	$\texttt{triple}^{kbf}\texttt{(?X, sc, ?Z)}$, $\texttt{subClass}^{ff}\texttt{(?Z, ?Y)}$
6b	$\texttt{subClass}^{fk}\texttt{(?X, ?Y)}$	$\texttt{subClass}^{ff}\texttt{(?X, ?Z)}$, $\texttt{triple}^{fbk}\texttt{(?Z, sc, ?Y)}$
6c	$\texttt{subClass}^{kb}\texttt{(?X, ?Y)}$	$\texttt{triple}^{kbf}\texttt{(?X, sc, ?Z)}$, $\texttt{subClass}^{bf}\texttt{(?Z, ?Y)}$
6d	$\texttt{subClass}^{ff}\texttt{(?X, ?Y)}$	$\texttt{subClass}^{ff}\texttt{(?X, ?Z)}$, $\texttt{triple}^{fbf}\texttt{(?Z, sc, ?Y)}$
7a	$\texttt{type}^{kf}\texttt{(?X, ?Y)}$	$\texttt{triple}^{kbf}\texttt{(?X, type, ?Y)}$
7b	$\texttt{type}^{fk}\texttt{(?X, ?Y)}$	$\texttt{triple}^{fbk}\texttt{(?X, type, ?Y)}$
7c	$\texttt{type}^{kb}\texttt{(?X, ?Y)}$	$\texttt{triple}^{kbb}\texttt{(?X, type, ?Y)}$
7d	$\texttt{type}^{ff}\texttt{(?X, ?Y)}$	$\texttt{triple}^{fbf}\texttt{(?X, type, ?Y)}$
8a	$\texttt{type}^{kf}\texttt{(?X, ?Y)}$	$\texttt{triple}^{kbf}\texttt{(?X, rdf:type, ?Z)}$, $\texttt{subClass}^{ff}\texttt{(?Z, ?Y)}$
8b	$\texttt{type}^{fk}\texttt{(?X, ?Y)}$	$\texttt{type}^{ff}\texttt{(?X, ?Z)}$, $\texttt{triple}^{fbk}\texttt{(?Z, subClassOf, ?Y)}$
8c	$\texttt{type}^{kb}\texttt{(?X, ?Y)}$	$\texttt{triple}^{kbf}\texttt{(?X, rdf:type, ?Z)}$, $\texttt{subClass}^{fb}\texttt{(?Z, ?Y)}$
8d	$\texttt{type}^{ff}\texttt{(?X, ?Y)}$	$\texttt{type}^{ff}\texttt{(?X, ?Z)}$, $\texttt{triple}^{fbf}\texttt{(?Z, sc, ?Y)}$
9a	$\texttt{type}^{kf}\texttt{(?X, ?Y)}$	$\texttt{newTriple}^{kff}\texttt{(?X, ?P, ?Z)}$, $\texttt{triple}^{fbf}\texttt{(?P, dom, ?Y)}$
9b	$\texttt{type}^{fk}\texttt{(?X, ?Y)}$	$\texttt{newTriple}^{fff}\texttt{(?X, ?P, ?Z)}$, $\texttt{triple}^{fbk}\texttt{(?P, dom, ?Y)}$
9c	$\texttt{type}^{kb}\texttt{(?X, ?Y)}$	$\texttt{newTriple}^{kff}\texttt{(?X, ?P, ?Z)}$, $\texttt{triple}^{fbb}\texttt{(?P, dom, ?Y)}$
9d	$\texttt{type}^{ff}\texttt{(?X, ?Y)}$	$\texttt{newTriple}^{fff}\texttt{(?X, ?P, ?Z)}$, $\texttt{triple}^{fbf}\texttt{(?P, dom, ?Y)}$
10a	$\texttt{type}^{kf}\texttt{(?X, ?Y)}$	$\texttt{newTriple}^{ffk}\texttt{(?Z, ?P, ?X)}$, $\texttt{triple}^{fbf}\texttt{(?P, range, ?Y)}$
10b	$\texttt{type}^{fk}\texttt{(?X, ?Y)}$	$\texttt{newTriple}^{fff}\texttt{(?Z, ?P, ?X)}$, $\texttt{triple}^{fbk}\texttt{(?P, range, ?Y)}$
10c	$\texttt{type}^{kb}\texttt{(?X, ?Y)}$	$\texttt{newtriple}^{ffk}\texttt{(?Z, ?P, ?X)}$, $\texttt{triple}^{fbb}\texttt{(?P, range, ?Y)}$
10d	$\texttt{type}^{ff}\texttt{(?X, ?Y)}$	$\texttt{newTriple}^{fff}\texttt{(?Z, ?P, ?X)}$, $\texttt{triple}^{fbf}\texttt{(?P, range, ?Y)}$

`newTriple`^{fff}. In addition, based on the order of the triple pattern's terms which we choose to use as an identifier (i.e., subject, object, predicate), not all combinations of the adornments are required. For example, the adorned predicate `subProperty`^{bk}`(?X, ?Y)` will never appear in our algorithm, since such a request would require to use as a key the object of the corresponding triple pattern although the subject is also bound.

5.2.2 Algorithm description

Let us now describe our backward chaining algorithm BC which is shown in pseudocode in Algorithm 2. The query requester node that receives request QueryRequest to pose a query composes a QueryMsg(id, tp, k, rid) message and sends it to the network as described in Section 4.2.2. id is the identifier obtained by hashing the key k of the triple pattern tp. Suppose that a QueryMsg request with unique identifier rid arrives at node n which is responsible for the identifier id. Node n firstly transforms the triple pattern tp to an adorned predicate p^a and calls local procedure `BwdRDFS`. Local procedure `BwdRDFS` takes as an input the adorned predicate p^a and the request identifier rid and outputs a relation R which contains the tuples of the bindings of the free arguments (i.e., the variables) of the predicate. These tuples of bindings form the answer to the query.

When `BwdRDFS` is called, the input predicate p^a is checked against the head of the rules of Table 5.2 using local function ApplyRule. Rules that can be applied to the predicate are added to the list *adornedRules*. Each rule can have one or two predicates in its body. Rules that have one predicate in their body (i.e., rules 1, 3, 5, 7) can always be evaluated locally since this predicate is always the edb relation `triple` and one of its bound arguments is the key that led to this node. In this case, node n calls local procedure MatchPredicate(p^a) and assigns to relation R the bindings of the predicate's variables that match the triples locally stored in its database.

For rules with two predicates in their body, we have to decide which predicate should be evaluated first. We select to evaluate first the predicate that can be processed locally. There is one such predicate always since one of the arguments of the head predicate is the key that led to the specific node. Therefore, there will be a body predicate (let us call it q_1) which has an adornment containing the letter k (this is always possible as seen in Table 5.2) and can be processed locally. If predicate q_1 is the edb relation `triple`, it is checked against the local database to find matching triples using local function MatchPredicate. The variable bindings are returned in relation R'. In case predicate q_1 is an idb relation, then procedure `BwdRDFS` is called recursively with input the new adorned predicate. By evaluating the first predicate locally, we have values

Algorithm 2: BC: Backward chaining algorithm

```
1  event n.QueryMsg (id, tp, k, rid) from m
2      Let p^a be the adorned predicate of tp;
3      R=BwdRDFS(p^a);
4      sendto m.QueryResp(R);
5  end

1  procedure n.BwdRDFS (p^a, rid)
2      if rid + p^a ∈ processedRequests then return {} ;
3      add rid + p^a to processedRequests;
4      adornedRules =ApplyRule(p^a);
5      forall the rules ∈ adornedRules do
6          r ← RemoveFirst(adornedRules);
7          if r.body has one predicate then
8              R=MatchPredicate (p^a);
9          else
10             Let q1 be the adorned predicate of r.body with a k element in its adornment and q2
               the other predicate of r.body;
11             if q1 is edb relation triple then
12                 R' = MatchPredicate (p_k);
13             else
14                 R' = BwdRDFS(q1, rid)
15             end
16             if R' = ∅ then return R;
17             foreach value v_i of the common variable ?Z in R' do
18                 id_i =Hash (v_i);
19                 rewrite q2 to q2';
20                 sendto id_i.BwdRDFSMsg(q2');
21                 receive BwdRDFSResp(R_i) from id_i;
22                 R = R ∪ R_i;
23             end
24         end
25     end
26     return R;
27 end

1  event n.BwdRDFSMsg (p^a, rid) from m
2      R=BwdRDFS(p^a);
3      sendto m.BwdRDFSResp(R);
4  end
```

that can be passed to the second predicate which is sent to be evaluated remotely at different nodes. Notice in Table 5.2 that all rule bodies with two predicates have a single common variable, let it be $?Z$. Therefore, each tuple in relation R' will include a binding for this common variable $?Z$. For each of these bindings $(?Z/v_i)$, node n rewrites the second predicate q_2 to a new predicate q_2' where it has substituted the variable $?Z$ with its value v_i and made the corresponding letter of the adornment equal to k. Then, it sends a BwdRDFSMsg message to the node responsible for the hashed value of key v_i. This part of the procedure is executed *in parallel* for each value v_i since the messages are sent to different nodes. Node n sends $|R'|$ number of messages (equal to the number of bindings found) and receives the responses asynchronously. When node n has collected all responses BwdRDFSResp(R_i), it adds the tuples of each R_i to relation R and returns R. In case of a Boolean query, i.e., a query without any variables, relation R actually holds a true or false answer. In this case, the union operator of line 26 is actually an OR operator meaning that procedure `BwdRDFS` requires at least one answer to be true in order to return true. Otherwise, it returns false.

This procedure is recursive in two ways. Recursion appears locally at a node when predicate q_1 is an idb predicate and among the nodes participating in the query evaluation when a new BwdRDFSMsg message is sent. The procedure terminates when the node that received the initial query has collected all responses. A recursion path ends when the predicate which is evaluated first returns no bindings and therefore there are no values to pass to the second predicate. Cyclic hierarchies are handled by keeping a list of all processed requests (lines 2-3 of `BwdRDFS` procedure) so that an infinite loop is avoided.

Let us now show how BC works using the example RDF(S) hierarchy of Figure 5.1. Suppose that all triples of the RDF(S) hierarchy are stored in the network and a user poses the query "Find all the subclasses of class artist" to a node in the network. Figure 5.5 shows how the rules will be distributed in the various nodes of the network. The query is expressed as the triple pattern `(?X, sc, artist)` and is routed to node n_1, which is responsible for key `artist`. Node n_1 transforms the triple pattern to the adorned predicate $\texttt{subClass}^{fk}\texttt{(?X, artist)}$ which matches the head of rules 5b and 6b of Table 5.2. Using rule 5b, node n_1 finds the local matches of the query, i.e., it retrieves all the immediate subclasses of class `artist` (`painter` and `sculptor`). Using rule 6b, it finds matches locally for the predicate `triple` and rewrites the second predicate `subClass` to other adorned predicates that are sent to other nodes to be evaluated. More specifically, node n_1 sends messages to nodes n_2 and n_3 to retrieve the subclasses of classes `painter` and `sculptor`, respectively. Node n_3 finds no matches for

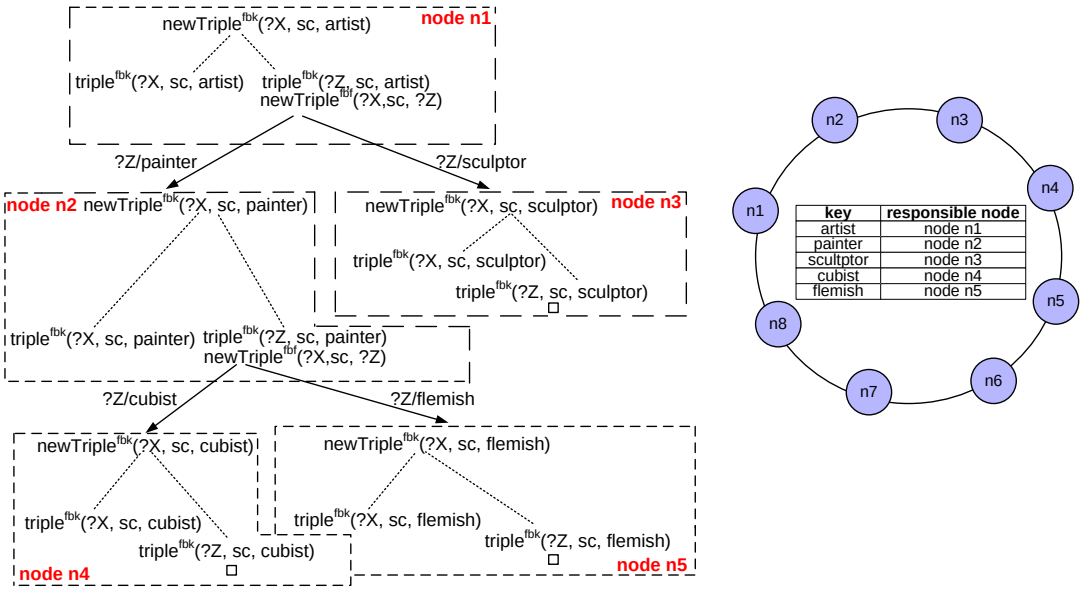


Figure 5.5: Distribution of rules for backward chaining

class `sculptor` and returns an empty result to node n_1. Node n_2 finds locally classes `cubist` and `flemish` using rule 5b and sends two other requests to nodes n_4 and n_5 using rule 6b. Nodes n_4, n_5 find no matches and return an empty result to node n_2. Node n_2 that collects the answers of nodes n_4 and n_5, adds to them the answers found locally and returns the answer to node n_1. Finally, as soon as node n_1 has the answers of both nodes n_2, n_3, it composes the final answer by adding its local answers with the answers of these nodes and returns the result set to the node that posed the query.

Comparison with the entailment algorithm of [101]. It is interesting to examine how our backward chaining algorithm is related with the entailment algorithm proposed in [101]. The entailment algorithm of [101] checks whether a triple t=`(a,p,b)` is entailed from a graph G. Comparing our algorithm with the entailment algorithm of [101], we observe certain similarities. The algorithm of [101] takes certain actions depending on the property `p` of the triple. Similarly, we trigger a rule depending on the property of the triple. The first point of the algorithm deals with the case where `p` is equal to `dom` or `range`. In this case, it just checks if t exists in G. Similarly, BC would trigger rules 3c and 4c from Table 5.2 and since no triple will be found to satisfy the first predicate of rule 4c, rule 3c will actually retrieve triples locally stored and therefore, triples that are in G. The second step of the algorithm deals with triples that have as property the `sp` value. In this case, the algorithm checks if there exists a path from `a` to `b` in the graph through `sp` links. Similarly, BC triggers rules 1c and 2c which traverse the transitive

closure of the graph consisting of `subProperty` relations. The same holds if `p` equals to `sc`. The 4th step of the algorithm, where property `p` does not belong to the *ρdf* vocabulary, is actually covered in our case with rules 3c and 4c. The difference is that the algorithm of [101] builds a graph to check if there is a path from a vertex marked with `(a,b)` that reaches `p` in a bottom-up fashion. On the contrary, BC works in a top-down fashion which searches if there is another property `p'` in a triple `(a,p',b)` and checks if there is a path from `p'` to `p` in the transitive closure of the subproperty relation. Finally, the algorithm deals with the case where the property of t is equal to `type`. The difference here, is that the algorithm of [101] needs to preprocess the whole graph G in order to mark certain nodes that are relevant to the triple in question. This case is covered in BC by rules 7c, 8c, 9c, 10c, 11c, 12c and traverses only the part of graph that is relevant to the query in a top-down manner.

5.2.3 Termination, soundness and completeness

First, we prove that BC terminates.

Theorem 5 (Termination)**.** *Algorithm BC terminates.*

Sketch. BC is a recursive algorithm in two ways. First, recursion occurs for a query q when one of the predicates of the body of a rule is an idb relation. If the predicate evaluated first is the edb relation `triple`, the second predicate to be evaluated is an idb relation and recursion occurs by sending a message BwdRDFSMsg to another node (line 30 of BC). A recursion path starts from the query q and traverses the RDF(S) graph in a top-down fashion until there is no other node to follow. In this case, the recursion path terminates since no tuples of bindings are found at a node for the first predicate and therefore there are no values to pass to the second predicate. In case the RDF(S) graph contain cycles an infinite loop can occur. To avoid infinite loops caused by graphs with cycles, the unique query request identifier together with the adorned predicate is inserted in a list *processedRequests*. Each time procedure `BwdRDFS` is called, the list of all processed requests is checked and if a request has been processed already, BC terminates by returning an empty set of tuples (lines 7-8 of BC).

Second, if the predicate that should be evaluated first (predicate with k in its adornment) is an idb relation, then we have local recursion at one node (line 19 of BC). This recursion can happen only once since the only rules which contain an adorned predicate with k in its adornment which is not the edb relation `triple` are rules 9a, 9c, 10a, and 10c of Table 5.2. In this case, local procedure `BwdRDFS` is called again but

this time the first predicate to be evaluated is the edb relation `triple` which results to a recursion of the previous case.

□

In the following we prove that algorithm BC is sound and complete. By *sound* we mean the following. Let G be a graph stored in the network and q be a query answered by BC. Let R be the relation-answer to q which has as attributes the variables of q and as tuples the tuples of bindings for these variables. If H is a set of triples obtained from q by replacing the variables of q by all the corresponding values in R, then $G \models H$. By *complete* we mean that if G is the graph stored in the network, H any graph and $G \models H$, then for each triple t in H, BC will return true for the query $q = t$. We make the same assumptions as in FC* regarding the network stability and message delivery.

We define the concept of a proof tree for BC, which models the execution of BC on top of a DHT, as follows.

Definition 5.2.2. *A* proof tree *is a finite tree with the following properties:*

(i) Every node is of the form $a_i : (q_i, R_i, LR_i, RR_i)$, where a_i is a DHT node, q_i is the query that the node should evaluate, R_i, LR_i are relations with tuples of bindings for the variables of query q_i and $RR_i = \{RR_i^1, RR_i^2, ..., RR_i^{k_i}\}$ is a set of relations with tuples of bindings for the variables of query q_i. Relations R_i and LR_i as well as the set RR_i can be empty.

(ii) For every node $a_i : (q_i, R_i, LR_i, RR_i)$ we have the following:

- *q_i is the query (triple pattern) with a key for which the node is responsible.*
- *LR_i is a relation that contains the tuples of bindings for q_i found from the local database of the node by applying one of the rules 1, 3, 5 or 7 depending on the property of q_i.*
- *$RR_i = \{RR_i^1, RR_i^2, ..., RR_i^{k_i}\}$ is a set of relations that node a_i received from its k_i children. If node a_i does not have any children, then the set RR_i is empty.*
- *If RR_i is non-empty then $R_i = LR_i \cup \bigcup_{j=1}^{k} RR_i^j$. Otherwise, $R_i = LR_i$.*

(iii) If node a_i has k_i children, then for each child $a_j : (q_j, R_j, LR_j, RR_j = \{RR_j^1, RR_j^2, ..., RR_j^{k_j}\})$, we have that $R_j = RR_i^j$ $(1 \leq j \leq k_i)$.

Such a proof tree offers us a nice representation of the computation of BC. Figure 5.6 shows the proof tree as defined above for the example shown in Figure 5.5. The root

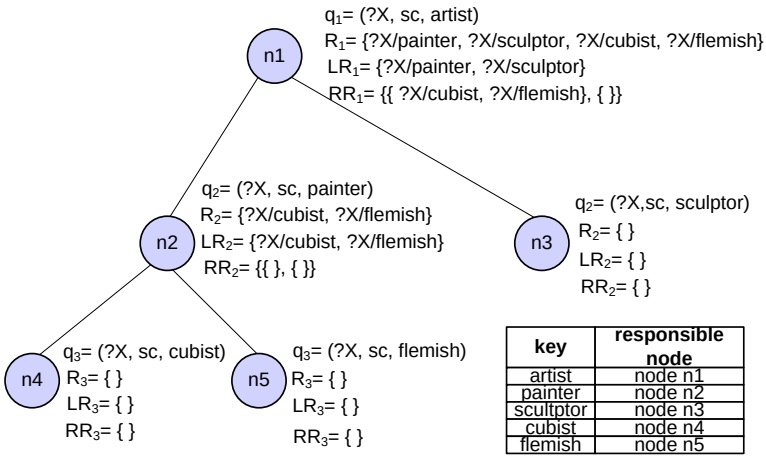


Figure 5.6: Proof tree for backward chaining

of the tree corresponds to the activation of BC when a message QueryMsg arrives at a DHT node. The nodes of the tree at increasing depths also show the communication between the nodes as BC proceeds. Each edge in the proof tree shows that a message BwdRDFSMsg was sent from the parent node to the child and one BwdRDFSResp message was sent from the child to its parent node.

Theorem 6 (Soundness)**.** *Let q be a query and R the relation produced by BC which has as attributes the variables of q. If G is the set of triples stored in the network and H is the set of triples obtained from q by replacing its variables by all their values in the relation R then $G \vdash_{mrdf} H$ and $G \models H$.*

Proof. The proof is by induction on the depth d of the proof tree of BC.

Base case: $d = 1$ (the root is $d = 0$). In this case, BC starts at the DHT node n_1 represented by the root of the tree and transforms the query q to an idb relation. If the predicate of q is equal to `sp` then the corresponding predicate is `subProperty` and rules 1 and 2 of Table 5.2 are applied, depending on which arguments of the query are bound or not. Node n_1 finds locally l tuples of bindings for the variables of q by applying rule 1, which has a single predicate in its body i.e., the edb predicate `triple`, and assigns them to relation LR. Then, the node applies rule 2, which has two predicates in its body. The predicate that is evaluated first is the one that has k in its adornment (predicate `triple` in this case). Since the first predicate is the edb predicate `triple`, k values are found locally and are passed to the second predicate (idb predicate `subProperty`) forming k new queries that will be evaluated at k different nodes. Therefore, node n_1 sends k BwdRDFSMsg messages to its k nodes (children nodes in the proof tree) for

evaluating a query with predicate `sp`. BC runs again at these k nodes and finds local results from rule 1. Since the depth of the tree is 1, rule 2 is not satisfied and no other messages are sent. Node n_1 receives k BwdRDFSResp response messages from the k nodes, each one containing a relation R^j (for $1 \leq j \leq k$) with the data the k nodes found locally. Then, n_1 composes the answer to query q by assigning to relation R the union of the tuples of all relations (i.e., $R = LR \cup \bigcup_{j=1}^{k} R^j$).

The sequence of graphs (*mrdf* proof according to Definition 3.2.2) that shows $G \vdash_{mrdf} H$ can easily be constructed. The first element of the sequence is G followed by k subsequences representing the sets of triples obtained from replacing the variables of q by their values in relations R^j for $1 \leq j \leq k$ (the order of the subsequences does not matter). Each of the k sets of triples can be represented by as many proof steps as the number of tuples of bindings contained in R^j. Each step is the result of the application of rule (2a) of the *ρdf* rules of Table 3.1 to the set of triples produced in the immediately previous proof step. Then, the sequence of graphs continues with l subsequences representing the triples obtained from replacing the variables of q by their values in relation LR. This can be done by applying rule (1b).

Similarly, if the property of the query is equal to `sc`, then rules 5 and 6 of Table 5.2 will be fired. To compose a sequence of graphs, *ρdf* rules (3a) and (1b) of Table 3.1 will be used. If the property of the query is anything but `sp`, `sc` and `type`, rules 3 and 4 of Table 5.2 will be fired and *ρdf* rules (2b) and (1b) would be used to construct a sequence of of graphs. Finally, if the property of the query is equal to `type`, then rules 7, 8, 9 and 10 of Table 5.2 will be fired at the DHT nodes. In case of rules 9 and 10, the idb predicate `newTriple` is the first predicate that will be evaluated. In this case, rules 3 and 4 will also be fired at node n_1. Messages are now sent to other nodes for the evaluation of the first predicate, while the values returned are passed to the second predicate which is the edb relation `triple` and will be evaluated locally. A *mrdf* proof for a query with property equal to `type` would then be constructed by rules (3b), (4), (2b) and (1b) of Table 3.1.

Inductive step: We assume the theorem holds for the execution of BC with a proof tree of depth d. We will show the result for a proof tree T of depth $d + 1$.

We take the proof tree T of depth $d + 1$ and prune the root node. The resulting k subtrees T_j ($1 \leq j \leq k$) have depth d and the theorem holds for the queries q_j of their root nodes. Hence, if H' is the union of the sets of triples obtained by replacing the variables of queries q_j by their value in the relations R_j, then $G \vdash_{mrdf} H'$. Now we can continue the proof of H' from G by adding the steps corresponding to the root node of tree T to arrive at graph H. This is done as in the base case.

Since we have proved $G \vdash_{mrdf} H$, Theorem 1 of Section 3.2.1 gives us that $G \models H$ as well. □

Theorem 7 (Completeness)**.** *Let G, H be mrdf graphs with G the graph stored in the network. If $G \models H$, then for each triple $t \in H$, BC will return true to the query $q = t$.*

Proof. Using Theorem 1 and the fact that $G \models H$, we have that $G \vdash_{mrdf} H$. If $G \vdash_{mrdf} H$, we will show that for each triple $t \in H$, BC will return true to query $q = t$. We will prove this using induction on k, the number of *mrdf* proof steps required for H to be derived from G.

For a Boolean query q, let it be $q = t =$`(a,p,c)`, the subject `a` is used as the identifier of the triple pattern and algorithm BC is instantiated at node n_1 which is responsible for key `a`. As soon as node n_1 receives a message QueryMsg, it transforms the triple pattern (t in this case) to an adorned predicate p^a depending on the property of the triple pattern t (i.e., t is transformed to one of the adorned predicates `subProperty`^{kb}, `newTriple`^{kbb}, `subClass`^{kb}, `type`^{kb}).

Base case ($k = 1$): All triples of H are derived from G in 1 proof step by the application of a single rule r of the minimal deductive system of Table 3.1.

- If r is rule (1b) then $H \subseteq G$. In this case, all triples of H are stored in the network according to our indexing scheme. At node n_1, BC will fire one of the rules 1c, 3c, 5c, 7c of Table 5.2 depending on the property of q. These rules have one predicate in their body which is the edb predicate `triple`. Local function MatchPredicate will retrieve matching triples from the node's local database. According to the indexing scheme, the triple `(a,p,c)` will be located at node's n_1 local database and the answer for query q will be true.

- If r is rule (2a), then there is an instantiation $\frac{R}{R'}$ of r such that $R \subseteq G$ and $H = G \cup R'$ and $t = R' =$`(a, sp, c)`. Let $t \in H$. R consists of two triples, $t_1 =$`(a, sp, b)` and $t_2 =$`(b, sp, c)`. Since this is the first *mrdf* proof step, $t_1, t_2 \in G$. The property of the triple t is equal to `sp` and hence the adorned predicate p^a at node n_1 is equal to `subProperty`^{kb}`(a, c)` and will be matched with the head of rules 1c and 2c of Table 5.2. If $t \in G$, then rule 1c will match with t and BC will return true. Otherwise, rule 2c will match locally predicate `triple(a, sp, Z)`. Since all triples of G have been stored using the triple indexing scheme and triple t_1 shares the same subject with t, triple t_1 (which has subject `a`) will be located at node n_1. Variable Z will be bound to the object of triple t_1, i.e., Z will be bound to value `b`. According to the algorithm, for each value v_i of

the bindings the second predicate of rule 2c is rewritten into a new one with a `subProperty` predicate without any variable. For the value `b` found from triple t_1, a new rewritten predicate will be `subProperty`^{kb}`(b, c)` and a BwdRDFSMsg message is sent to node n_2 which is responsible for key `b`. Node n_2 that receives this message matches the predicate with rule 1c. Triple t_2 has as subject the same value with triple's t_1 object (i.e., `b`). Therefore, node n_2 retrieves locally triple t_2 =`(b, sp, c)` and returns true to node n_1. Then, node n_1 returns true for query q.

- If r is one of the rules (2b),(3) or (4) then there is an instantiation $\frac{R}{R'}$ of r such that $R \subseteq G$ and $H = G \cup R'$. The proof is similar with the one for rule (2a).

Inductive step: We assume that the result holds for H that can be proved from G with number of proof steps less than k. We will prove that it holds for H that can be proved from G with number of proof steps equal to k.

So let us assume that $G \vdash_{mrdf} H$ in k proof steps. Using the notation of Definition 3.2.1, $P_k = H$. Let r be the rule of Table 3.1 that was used in the proof to go from P_{k-1} to P_k.

- If r is rule (1b), then $H \subseteq P_{k-1}$ and for each triple $t \in H$, $G \vdash_{mrdf} t$ in $k-1$ or less steps. Then, from the induction hypothesis, BC returns true to query $q = t$.

- If r is rule (2a), there will be an instantiation $\frac{R}{R'}$ of r such that $R \subseteq P_{k-1}$ and $H = P_{k-1} \cup R'$. Let $t \in H$. If $t \in P_{k-1}$, then $G \vdash_{mrdf} t$ in less than k proof steps, and therefore BC returns true for the query $q = t$ by the induction step. If $t \in R'$, then it is equal to `(a, sp, c)`. In BC, node n_1 matches the adorned predicate `subProperty`^{kb}`(a, c)` with the head of rules 1c and 2c of Table 5.2.

- If r is one of the rules (2b),(3) or (4) then there is an instantiation $\frac{R}{R'}$ of r such that $R \subseteq P_{k-1}$ and $H = P_{k-1} \cup R'$. The proof is similar with the one for rule (2a).

□

5.3 Forward Chaining for Magic Rules

Comparing a backward chaining algorithm with a forward chaining one is not always a fair comparison since forward chaining always computes all possible inferences while backward chaining focuses on a specific goal. In this section, we present a bottom-up

method that benefits from the top-down technique of backward chaining. This method is well-known from the database literature as *magic sets* transformation technique [17]. The basic idea is that, given a specific query, rules are rewritten using information from the query so that a bottom-up evaluation is able to generate only the appropriate inferences. The benefit of using the new ruleset in the bottom-up evaluation is that it focuses only on data which is associated with the query and hence no unnecessary information is generated. In our case, we use the same ideas and rewrite the Datalog version of the *ρdf* inference rules using the magic sets transformation. When the rewritten rules are executed in a forward chaining fashion, only triples that are related to the query are involved.

5.3.1 Magic rules

Imagine that we need to answer the query "Find all instances of class `a`". This query is a very commonly used query which often requires RDFS inference so that derived instances of `a` are also computed. In this section, we focus only on queries of the form `(?X, type, a)` in order to show how the magic sets algorithm can be implemented on top of a DHT and to compare the bottom-up approach of this algorithm with the top-down approach of backward chaining. Although we rewrite the *ρdf* inference rules using the magic set transformation technique based on this type of query, the algorithm we present in the following can be applied for any type of query if the rules are rewritten accordingly.

In order to transform a set of rules using the magic sets technique, we require the *unique binding property* [161]. This requirement means that each idb relation should appear with a unique adornment when a backward chaining algorithm is used. The rules of Table 3.2 do not satisfy the unique binding property for this type of query [161]. The idb relation `newTriple` appears with different adornments if we apply the rules using the backward chaining algorithm. For example, consider the execution of query `newTriple(?X, type, a)`. Using rule 5, the idb relation `newTriple` appears with bound the second and third argument, while using rules 6 or 7 and after passing values from the edb relation `triple`, it appears with bound only the second argument. Hence, the rules cannot be transformed into a set of magic rules, unless the predicate `newTriple` is split in two different predicates. However, this would introduce twice the number of rules imposing unnecessary overhead. For this reason, we prefer to use the *ρdf* inference rules as we presented them in Table 5.1. We transform the above rules into a set of magic rules for the query q using the techniques of [17]. The rules are shown in Table 5.3.

Table 5.3: Magic Rules

Rule	Head	Body
1	m_newTriple(?P)	sup_{61}(?P, ?Y)
2	m_newTriple(?P)	sup_{71}(?P, ?Y)
3	m_subProperty(?Z)	sup_{21}(?Z,?Y)
4	m_subProperty(?P)	m_newTriple(?P)
5	m_subClass(?Z)	sup_{41}(?Z,?Y)
6	m_type(?Z)	sup_{51}(?Z,?Y)
7	sup_{21}(?Z, ?Y)	m_subProperty(?Y), triple(?Z, sp, ?Y)
8	sup_{31}(?P1, ?P)	m_newTriple(?P), subProperty(?P1, ?P)
9	sup_{41}(?Z, ?Y)	m_subClass (?Y), triple(?Z, sc, ?Y)
10	sup_{51}(?Z, ?Y)	m_type(?Y), triple(?Z, sc, ?Y)
11	sup_{61}(?P, ?Y)	m_type(?Y), triple(?P, dom, ?Y)
12	sup_{71}(?P, ?Y)	m_type(?Y), triple(?P, range, ?Y)
13	newTriple(?X, ?P ,?Y)	m_newTriple(?P), triple(?X, ?P, ?Y)
14	subProperty(?X, ?Y)	m_subProperty(?Y), triple(?X, sp, ?Y)
15	subProperty(?X, ?Y)	sup_{21}(?Z,?Y), subProperty(?X, ?Z)
16	newTriple(?X, ?P, ?Y)	sup_{31}(?P1,?P), triple(?X,?P1,?Y)
17	subClass(?X, ?Y)	m_subClass(?Y), triple(?X, sc, ?Y)
18	subClass(?X, ?Y)	sup_{41}(?Z,?Y), subClass(?X, ?Z)
19	type(?X, ?Y)	m_type(?Y), triple(?X, type, ?Y)
20	type(?X, ?Y)	sup_{51}(?Z, ?Y), type(?X, ?Z)
21	type(?X, ?Y)	sup_{61}(?P, ?Y), newTriple(?X, ?P, ?Z)
22	type(?X, ?Y)	sup_{71}(?P, ?Y), newTriple(?Z, ?P, ?X)
23	m_type(a)	

For each idb predicate p, we create a magic predicate m_p which has as arguments the bound arguments of the unique binding appearing in p (i.e., the unique adornment). For each rule we introduce a number of magic supplementary predicates associated with this rule. A supplementary predicate sup_{ij} denotes that it is the supplementary predicate for rule i and the predicate j of the body of rule i with $j = 0, .., n$. For example, sup_{21} denotes the supplementary predicate for predicate subProperty of the body of rule 2 in Table 5.1. Rules 1-6 define the magic predicates, rules 7-12 show the supplementary predicates and rules 13-22 show the initial rules modified to include the supplementary predicates. All these rules together with the fact m_type(a) (rule 23), which is used as the initialization rule, form the complete rule set after the magic set transformation.

This set of rules ensures that only inferences related with class a will be generated. The role of magic predicates is that m_p(v) should be true if and only if in the top-down evaluation value v is passed as a binding to predicate p. Similarly, supplementary

Algorithm 3: MS: Magic sets algorithm

```
1  event n.MSReq (id, pred) from m
2  |  Magic(id, pred);
3  end

1  procedure n.Magic (id, pred)
2  |  localTriples = getTriplesFromDB (pred.argument1);
3  |  Infer (localTriples, pred);
4  |  forall the t' of newtriples with t' ∉ infTriples && t' ∉ localTriples do
5  |  |  if t'.property ∈ ρdf then
6  |  |  |  id =Hash (t'.object);
7  |  |  |  sendto id.StoreMsg(id_1, t', 3, true);
8  |  |  else
9  |  |  |  id =Hash (t'.property);
10 |  |  |  sendto id.StoreMsg(id_1, t', 2, true);
11 |  |  end
12 |  |  infTriples.add(t');
13 |  |  forall the pred' ∈ supPredicates && ∉ infPreds do
14 |  |  |  id' =Hash (pred'.argument1);
15 |  |  |  sendto id.MSReq(id', pred');
16 |  |  end
17 |  |  infPreds.add(pred');
18 |  end
19 end

1  event n.StoreMsg (id, t, k, inf) from m
2  |  if inf == true then Magic(id, pred);
3  |  InsertToDB (t, inf);
4  end
```

predicates represent the bound variables that have either been bound by the rule's head or by predicates evaluated at a previous step. Note here that we present an optimized version of the magic set transformation where zeroth supplementary predicates have been replaced by the magic predicates so that extra computations are avoided. For more information on the magic sets transformation the interested reader might refer to [161].

5.3.2 Algorithm description

Let us now describe how our new algorithm, which we call MS, works. Generally, MS works as the forward chaining algorithm presented in Section 5.1 with the only difference that more predicates are inferred from the rules. In MS, apart from the edb relation `triple` and the idb relations `subProperty`, `newTriple`, `subClass` and `type`, we also have the idb relations of the magic and the supplementary predicates. The

relations of the supplementary predicates are also indexed in the network based on the values of their arguments so that appropriate values can be found locally at the corresponding nodes. Magic predicates are not indexed since they can be reproduced by the supplementary predicates and rules 1-6 of Table 5.3.

Assume that a node needs to find the instances of class `a`. It sends a message containing the magic predicate `m_type(a)` to the node responsible for value `a` using the hash value of `a` as an identifier. The node that receives the magic predicate `m_type(a)` starts a bottom-up evaluation of the rules of Table 5.3 and sends the new inferred predicates to the network. Each time a node receives a new predicate, it computes the closure locally according to the rules of Table 5.3 and distributes the newly inferred facts in the network.

The pseudocode of the algorithm is presented in Algorithm 3. The node that receives the query transforms it to the corresponding magic predicate *pred*. Then, it creates an identifier *id* by hashing the value of the only argument of the magic predicate *pred* and sends a MSReq($id, pred$) message to the network. When a node n receives such a message it calls local procedure `Magic`. This procedure works as FC* with two differences. First, newly inferred triples are only indexed once in the network. For the idb predicates `subProperty`, `subClass` and `type`, only the object of the corresponding triple is used for indexing the triple. For the idb predicate `newTriple`, the property of the corresponding triple is used as an identifier instead. This indexing scheme ensures that triples will be sent to the appropriate nodes so that the reasoning process is complete. Second, inferred supplementary predicates are also sent in the network to the corresponding nodes. For each newly inferred supplementary predicate $pred'$, an identifier id' is created by hashing the value of the first argument of $pred'$. Then a message MSReq(id, $pred'$) is sent to the responsible node in the network. We only use the first argument of $pred'$ since we are already at the node responsible for the value of the second argument and the required reasoning has been already completed. Procedure `Magic` is called when a node receives either a newly inferred triple or a new supplementary predicate value. The algorithm terminates when all nodes have reached a fixpoint and neither new triples nor new supplementary predicate values are generated.

The soundness and completeness of this algorithm follows from the equivalence of the original rules to the rewritten ones given the query [161] and the soundness and completeness results from FC*.

In Figure 5.7, we demonstrate an example of the algorithm MS using the computation tree as defined in Section 5.1, where we have the edb and idb relations instead

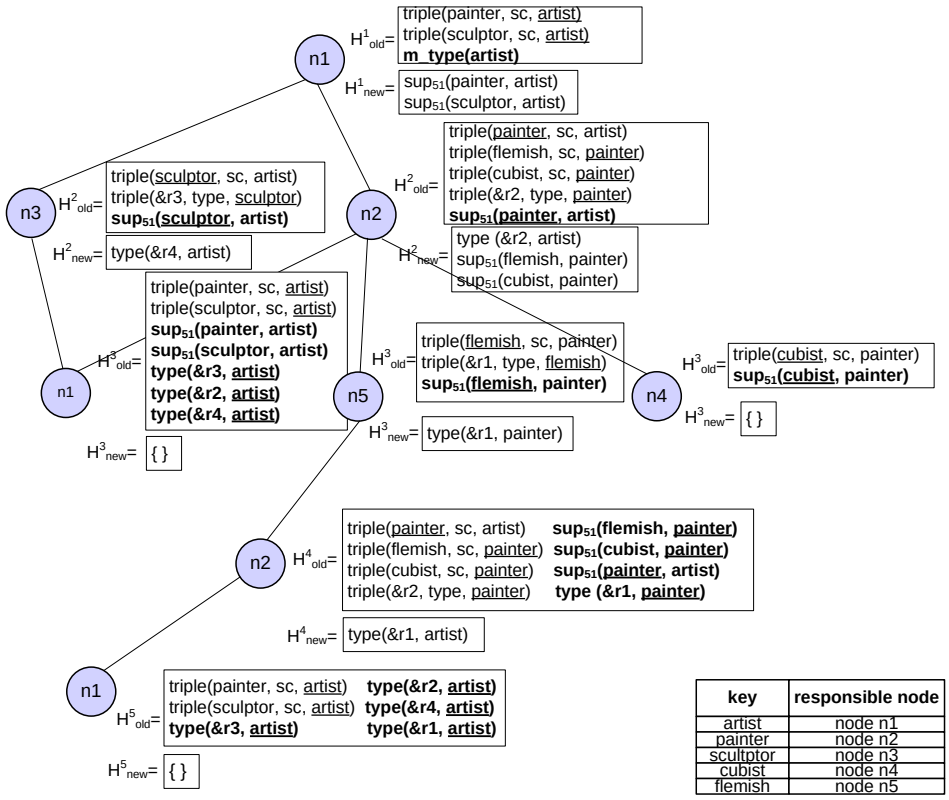


Figure 5.7: Example for MS algorithm

of triples. We use the RDF(S) example of Figure 5.1 and we want to compute all instances of class `artist`. Node n_1 receives a MSReq message with the predicate `m_type(artist)`. Using the magic set rules of Table 5.3, the supplementary predicates shown in H^1_{new} of node n_1 are generated and sent to nodes n_2 and n_3 which are responsible for keys `painter` and `sculptor` respectively. These nodes apply the magic sets rules using as input data the data appearing in H^2_{old} and generate new values for the idb relations. The procedure continues until all nodes have reached a fixpoint (i.e., H^i_{new} is empty). In this case, node n_1 has received all three instances of class `artist`. Then, the answer to the query can be found locally at node n_1.

5.4 An Analytical Cost Model

In this section, we present an analytical cost model for algorithms FC, FC*, BC and MS presented earlier. We show in the experimental evaluation of Section 5.5 that our implementation follows the predictions of this cost model. We focus on the

frequently used query `(?X, type, a)` which asks for all the instances of class `a` in an RDFS hierarchy. As we have already seen, the algorithms are able to answer any type of queries considered in the thesis. Our results for class hierarchies trivially hold for property hierarchies too.

We assume a complete tree-shaped RDFS class hierarchy of depth d and branching factor b as our schema (i.e., $(b^{d+1}-1)/(b-1)$ classes) with instances distributed under classes following either a uniform or a Zipfian distribution. Some triples are of the form `(`c_k`, sc, `c_l`)` (RDF Schema triples) while the rest are of the form `(`r_j`, type, `c_k`)` (RDF data triples). In the analytical calculations presented below we start with an RDF(S) database. Then, we apply the FC/FC* and BC algorithms in order to be able to answer a query of the type mentioned above, and estimate its cost. We denote the number of triples of the form `(`r_j`, type, `c_k`)` in our initial database by T_d (i.e., also total number of instances). Similarly, we use T_s to denote the number of triples of the form `(`c_k`, sc, `c_l`)` in the initial database. For a uniform distribution, given the total number of instances (i.e., T_d), the number of instances under *each* class is $I_u = (T_d*(b-1))/(b^{d+1}-1)$. Considering a Zipfian distribution of instances with a skew parameter of 1, a class with rank r has $I_r = T_d/(r*h)$ instances where $h = \sum_{j=1}^{N} 1/j$ for N classes ($N = (b^{d+1}-1)/(b-1)$). Leaf classes are given a lower rank.

In the following, we constantly use the result that the total number of subclasses of class c including itself is $(b^{d-\ell+1}-1)/(b-1)$ where ℓ is the level of the class. The proof is straightforward and is omitted. Furthermore, we utilize the fact that the reasoning and query answering algorithms for the type of queries and rules we consider are essentially transitive closure computations [68].

5.4.1 Storage cost model

We first estimate analytically the costs associated with the storage of triples (given and inferred triples) in all algorithms. These costs are captured by two parameters that we define below: database storage load and number of store messages. Both parameters are measured using a uniform as well as a Zipfian distribution of instances under classes.

Storage load

We define as *database storage load* the total number of triples that are stored in the network. In BC, the storage load (SL_b) is three times the number of RDF(S) triples that were inserted in the network based on our indexing scheme. In FC and FC*, triples initially inserted in the network, as well as inferred triples, are stored three times.

Therefore, it is sufficient to compute the total number of triples that results from the transitive closure computations of the hierarchy (triples initially in the database plus inferred ones). Then, the database storage load incurred in FC and FC* is three times this total number of triples. Note that the database storage load of FC and FC* is equal and hence we denote it with SL_f.

Lemma 5.4.1. *The total number of triples generated after the computation of the transitive closure of the assumed RDFS class hierarchy (given triples plus inferred RDFS triples) is* $T_s + \sum_{i=1}^{d} b^i \times (i-1)$.

Proof. For each level i of the tree, we have b^i classes, and for each class we infer $i-1$ triples of the form (C$_k$, sc, C$_l$) for the upper levels of the tree. □

Lemma 5.4.2. *The total number of triples generated after the computation of the inferred instances of the assumed RDFS class hierarchy (given triples plus inferred RDF triples) is* $T_d + I_u \times \sum_{i=1}^{d} b^i \times i$.

Proof. Each class has I_u direct instances. For each level i of the tree, we have b^i classes, and for each class we infer i triples of the form (r$_j$, type, C$_k$) for its i superclasses. □

Based on these two lemmas, the total number of triples stored in the nodes of the network after FC/FC* has terminated is at most three times the sum of the formulas computed above. Depending on the distribution of instances, the storage load of FC changes based on the number of instances per class (i.e., I_u and I_r). For the Zipfian distribution we also made use of the following proposition.

Proposition 5.4.1. *Given a class with rank* r *in a Zipfian distribution of instances of a hierarchy with depth* d*, the level of the class in the hierarchy is* $\ell_r = \lfloor \log_b((b-1) \times [(b^{d+1}-1)/(b-1) - r + 1]) \rfloor$.

The database storage load for the MS algorithm (SL_m) depends on the level of the class that is queried and the number of inferred instances for this class.

Lemma 5.4.3. *The total number of inferred instances of a class at level* ℓ *of an RDFS class hierarchy is* $I_u \times \sum_{i=1}^{d-\ell} b^i \times i$.

Proof. We can think of a class hierarchy under a class at level ℓ as a complete class hierarchy with depth $d-\ell$. Therefore, the result follows directly from Lemma 5.4.2. □

Based on the above Lemma, the database storage load for MS for a uniform distribution is $3 \times (T_s + T_d) + \sum_{i=1}^{d-\ell} b^i \times i$. For a Zipfian distribution, we would have $\sum_{r=1}^{N_\ell} I_r \times \ell_r$ inferred instances, where $(N_\ell = b^{d-\ell+1} - 1)/(b-1)$.

Table 5.4 summarizes the database storage load of FC/FC*, BC and MS for both kinds of instance distributions.

Store messages

We define as *store messages* the number of DHT messages sent for storing triples. In BC, the number of store messages sent (SM_b) is three times the number of triples stored and therefore it is equal to the database storage load incurred, i.e., $SM_b = 3*(T_s+T_d)$. It is also independent from the instance distribution among the RDFS classes.

Because of the nature of the RDF(S) graph, where we have only triples of the form (`r`$_j$`, type, c`$_k$) and (`c`$_k$`, sc, c`$_l$), FC* does not generate any redundant triples. Therefore, the number of store messages sent by FC* is equal to the database storage load of FC*. However, the total number of messages sent by FC (SM_f) is much larger than the database storage load incurred. This is caused by the redundant triples that are sent in the network. Each node responsible for a certain class c that is in the i level of the class hierarchy generates for each instance i triples of the form (`r`$_j$`, type, c`$_k$), where c_k is a superclass of class c, and $i-1$ triples of the form (`c, sc, c`$_l$), where c_l is an ancestor class of class c. Note that messages are sent not only for each direct instance of each class but also for the instances of its subclasses. The number of messages sent by FC is depicted in Table 5.4 for both kinds of instance distribution. We do not depict the number of messages sent by FC* in the Table.

The number of messages sent by MS (SM_m) is equal to the database load incurred and depends on the level of the class whose the instances are asked.

5.4.2 Querying cost model

In this section, we calculate the cost of answering the query (`?X, type, c`) where class c is at level ℓ of the hierarchy.

Query processing load

We define as *query processing load* of a node the number of triples that this node retrieves from its local database in order to answer a query.

The FC/FC* and MS algorithms generates query processing load (QL_f) only at the node that is responsible for class c. This load is equal to the total number of instances of class c: $QL_f = ((b^{d-\ell+1}-1)*I_u)/(b-1)$. On the other hand, BC will generate load at $(b^{d-\ell+1}-1)/(b-1)$ nodes, namely the nodes that are responsible for the subclasses

Table 5.4: Storage cost summary table

Storage cost	**Uniform**	**Zipfian**
SL_b	$3 \times (T_s + T_d)$	$3 \times (T_s + T_d)$
SL_f	$3 \times [T_s + T_d + \sum_{i=1}^{d} b^i(i-1) + I_u \times \sum_{i=1}^{d} b^i \times i]$	$3 \times [T_s + T_d + \sum_{i=1}^{d} b^i(i-1) + \sum_{r=1}^{N} I_r \times \ell_r]$
SL_m	$3 \times (T_s + T_d) + \sum_{i=1}^{d-\ell} b^i \times i$	$3 \times (T_s + T_d) + \sum_{r=1}^{N_\ell} I_r \times \ell_r$
SM_b	$3 \times (T_s + T_d)$	$3 \times (T_s + T_d)$
SM_f	$3 \times [T_s + T_d + \sum_{i=1}^{d} b^i(i-1) \times ((b^{d-i+1} - 1)/(b-1)) + I_u \times \sum_{i=1}^{d} b^i \times (i+1) \times ((b^{d-i+1} - 1)/(b-1))]$	$3 \times [T_s + T_d + \sum_{i=1}^{d} b^i(i-1) \times ((b^{d-i+1} - 1)/(b-1)) + \sum_{r=1}^{N} I_r \times \ell_r \times (\ell_r + 1)/(b-1)]$
SM_m	$3 \times (T_s + T_d) + \sum_{i=1}^{d-\ell} b^i \times i$	$3 \times (T_s + T_d) + \sum_{r=1}^{N_\ell} I_r \times \ell_r$

of class c including c. The load of each of these nodes (QL_b) is simply I_u. Note that the total query load is the same for both approaches.

For a Zipfian distribution, the number of instances of the class would be $I_r = T_d/(h * r)$, where the rank of the class is in the interval $[(b^d - 1)/(b-1) + 1, (b^{d+1} - 1)/(b-1)]$.

Query messages

We define as *query messages* the messages sent while answering a query. The cases of FC/FC* and MS is straightforward since just one message is sent to the node responsible for class c. In BC, the number of messages sent (QM_b) is as many as the number of the subclasses of class c. Therefore, we have: $QM_b = [(b^{d-\ell+1} - 1)/(b-1)] - 1$. The distribution of the instances does not affect the number of messages sent for the query answering.

5.5 Experimental Evaluation

We present a detailed experimental evaluation of the algorithms described in this chapter. All algorithms have been implemented in our system Atlas, which is built on

top of the Bamboo DHT [122]. In our algorithms, we have also utilized the dictionary encoding implemented in Atlas, where URIs and literals are mapped to integer identifiers. More details about the implementation of a mapping dictionary in a DHT are given in Chapter 6. Our goal in this section is to evaluate and compare the performance of the RDFS reasoning algorithms we discussed before in a real distributed system.

5.5.1 Experimental setup and datasets

For our experiments, we first used as a testbed the PlanetLab network[2]. PlanetLab is a global network used for testing in various research fields. At the time of our experiments, it consisted of 1132 nodes at 513 sites worldwide. Resources of PlanetLab nodes (such as memory, disk and bandwidth) can be very limited at the time of experiments since PlanetLab nodes are used simultaneously by many users which share the nodes' resources. This very often results in overloaded nodes and bandwidth congestion. In this experimental evaluation, we used about 210 nodes that were available at the time of the experiments.

Although PlanetLab offers real-world testing of distributed systems, due to its reliability and performance issues[3], we also tested our system in a local shared cluster[4] so that we could scale to bigger datasets. The cluster consisted of 41 commodity machines with two processors at 2.6GHz and 4GB memory. We used 39 of these machines where we run multiple instances of Atlas on each machine (i.e., up to 4 Atlas nodes per machine). This allowed us to build networks of up to 156 Atlas nodes in total. The experiments to be presented in Sections 5.5.2 and 5.5.3 were conducted in PlanetLab and thus use smaller datasets. In Sections 5.5.4 and 5.5.5, we demonstrate results from the local cluster where we were able to scale to bigger datasets.

The datasets we used are taken from two different benchmarks. The first one is the RBench generator[5] [156] which produces RDF(S) data synthetically. The generator produces binary-tree-shaped RDFS class hierarchies parameterized on three different aspects: the depth of the tree, the total number of instances under the tree, and the distribution of the instances under the nodes of the tree. The queries we measure are queries that ask for all the transitive instances of the root class of the RDFS hierarchy. We used class hierarchies of depth $2-6$ (corresponding to $7-127$ RDFS classes). We used both uniform and Zipfian distribution of instances under the RDFS class hierarchy.

[2] `http://www.planet-lab.org/`

[3] These issues are well-known in the PlanetLab community. They have been discussed by other researchers as well e.g., by the group that implemented PIER [61].

[4] `http://www.grid.tuc.gr/`

[5] `http://athena.ics.forth.gr:9090/RDF/RBench/`

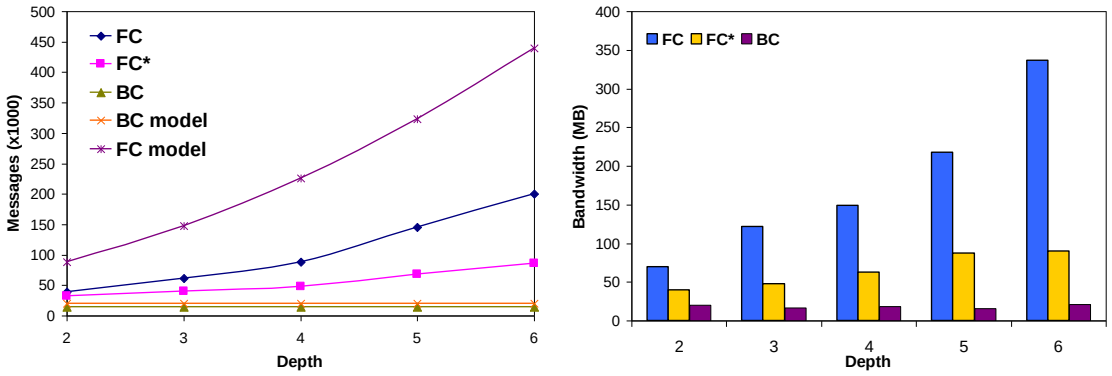
In the Zipfian distribution, we used a skew parameter of value 1. Leaf classes were given a lower rank and therefore more instances of the lower level classes were generated.

The second benchmark is the Lehigh University benchmark (LUBM) [48] that provides synthetic RDF datasets of arbitrary sizes. LUBM consists of a university domain ontology modeling an academic setting and is widely used for testing RDF stores. Each dataset can be defined by the number of universities generated and is described in RDF. For example, the dataset LUBM-1 involves one university, while the dataset LUBM-10 involves 10 universities. The more universities are generated the more triples are produced. Since our intention in this chapter is to compare the reasoning algorithms, we did not want to use the LUBM queries and involve algorithms for conjunctive queries. Therefore, we used as a query workload queries that ask for instances of certain classes included in the LUBM ontology. In the next chapters we discuss algorithms that can process conjunctive queries such as the ones provided by LUBM.

5.5.2 Storing RDF(S) data

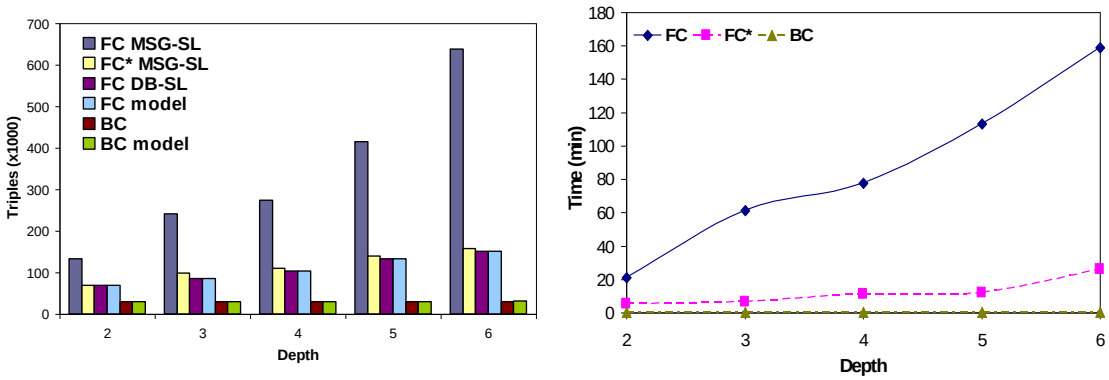
In this section, we compare the performance of the forward chaining algorithm (FC) with the backward chaining algorithm (BC) when storing RDF(S) data in the network. Alongside the results of the forward chaining algorithm we have presented in [74], we also present results from the forward chaining algorithm described in Section 5.1 (FC*). For this set of experiments, we have generated $10,000$ instances uniformly distributed under an RDFS class hierarchy of varying depth using the RBench generator. We inserted the data in the network together with the corresponding RDFS class hierarchy and measured the following metrics: network traffic, the load incurred at the nodes and the time required for all triples to be stored in the network (i.e., for backward chaining just the given triples and for forward chaining the given and the inferred triples).

Initially, we present results concerning the network traffic that is generated by the system measured in terms of both total number of messages sent in the network and bandwidth usage. As shown in Figure 5.8(a), the number of messages sent by FC increases exponentially with the tree-depth while it remains constant in BC. In this experiment, we use the multiPut functionality described in Section 5.5.2 for both approaches, which groups triples, and thus the number of messages measured in the experiment is less than the number of messages computed by the analytical cost model for both algorithms (lines *BC model* and *FC model* in the graph). multiPut decreases significantly the number of messages sent by FC. In the same figure, we depict the number of messages sent by algorithm FC*. We observe that the number of messages sent by FC is greater than the number of messages sent by FC* for every tree depth



(a) messages (b) bandwidth

Figure 5.8: Network traffic

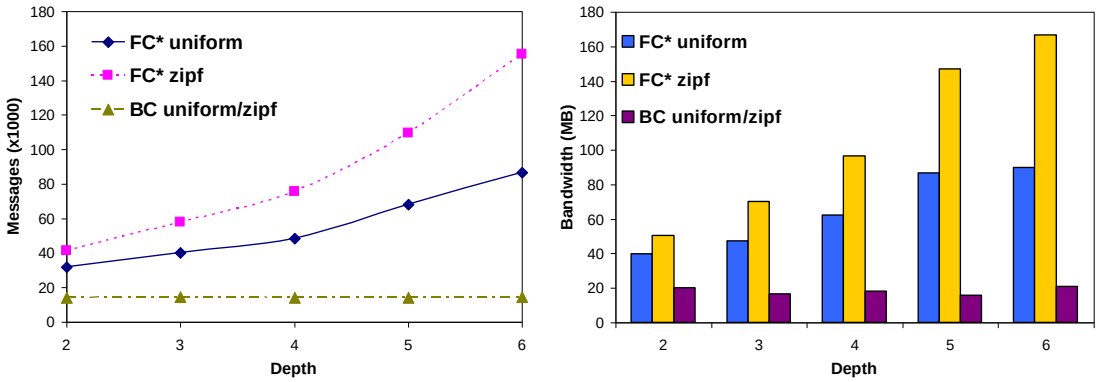


(a) storage load (b) storage time

Figure 5.9: Storage load and time

and the difference becomes greater as the tree-depth of the class hierarchy grows. This results from the fact that deeper RDFS class hierarchies cause more redundant triples to be generated. The graph of Figure 5.8(b) shows the total bandwidth consumption of the algorithms in MBytes. We observe that the bandwidth is increasing exponentially with the tree-depth in FC while remaining constant in BC. As the number of messages sent affects the bandwidth consumption in the network, FC consumes dramatically more bandwidth than FC*.

Figure 5.9(a) shows the total load incurred at all nodes of the network when storing RDF(S) data. We define as *database storage load* of a node n the number of triples that n stores locally in its database. We also define as *store message load* of a node n the number of triples that n receives to store in its local database. If the triples



(a) messages (b) bandwidth

Figure 5.10: Network traffic for uniform and Zipfian distribution

that are sent to node n to be stored are not already stored in its local database, then the database storage load is equal to the store message load. If at least one triple is already stored at n, then the store message load is greater than the database storage load. The difference of these two metrics allow us to estimate the redundant triples that are sent to node n. The total database storage load (DB-SL) is the total number of triples stored in the network. The total store message load (MSG-SL) is the sum of the store message load incurred at all nodes of the network. Figure 5.9(a) shows DB-SL and MSG-SL for algorithms FC, FC*, and BC. DB-SL and MSG-SL of BC are equal and we only depict them by the single bar *BC*. BC's storage load is significantly lower than FC and is independent of the tree-depth. Both FC and FC* cause the same database storage load and therefore we depict it with bar FC DB-SL. However, the message storage load of FC and FC* have a significant difference. While MSG-SL of FC grows exponentially with the tree depth, MSG-SL of FC* increases more gently and is almost equal with the database storage load incurred in the network. This means that FC* does not produce the amount of redundant triples generated by FC. Figure 5.9(a) also shows that our cost model regarding the database storage load (BC model and FC model in the graph) precisely predicts the results obtained in the experiments.

In Figure 5.9(b), we show the time needed by each approach to complete the insertion of RDF(S) data. In BC, this time represents the time needed until all given triples are stored at the respective nodes. In FC and FC*, this time represents the time required for the algorithm to terminate, i.e., to reach a fixpoint. Certainly, the time required by BC to store RDF(S) data is independent of the tree depth. The time was about 30 seconds and thus the line is very close to the x-axis. On the contrary, FC

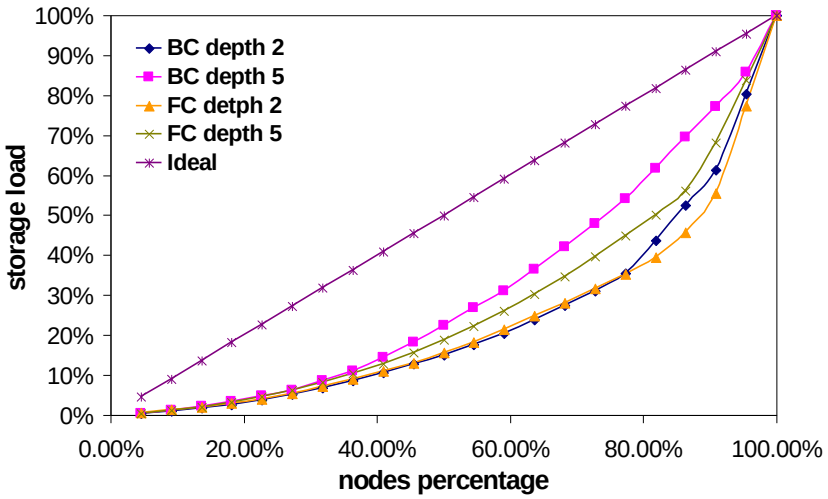


Figure 5.11: Database storage load distribution

and FC* require a time proportional to the tree depth. FC requires a larger amount of time to reach a fixpoint than FC*, a phenomenon that is magnified as the tree depth grows. Generally, both forward chaining algorithms require an enormous amount of time to complete which made the measurement of inserting more than $10,000$ triples of the RBench dataset infeasible. Our experiment while inserting $100,000$ triples using FC* was active for *16 hours* and still a fixpoint had not been reached.

In Figure 5.10, we show the number of messages and the bandwidth consumption when having instances that follow a Zipfian distribution under RDFS class hierarchies of varying tree depth. The distribution does not affect BC when storing RDF(S) data since the same number of triples are sent to be stored in the network. On the contrary, the performance of FC* deteriorates for a Zipfian distribution. In a Zipfian distribution, more instances belong to low level classes and as a result more triples need to be sent to upper level classes generating more network traffic. Therefore, both the number of messages sent and the bandwidth consumption increase significantly for a Zipfian distribution.

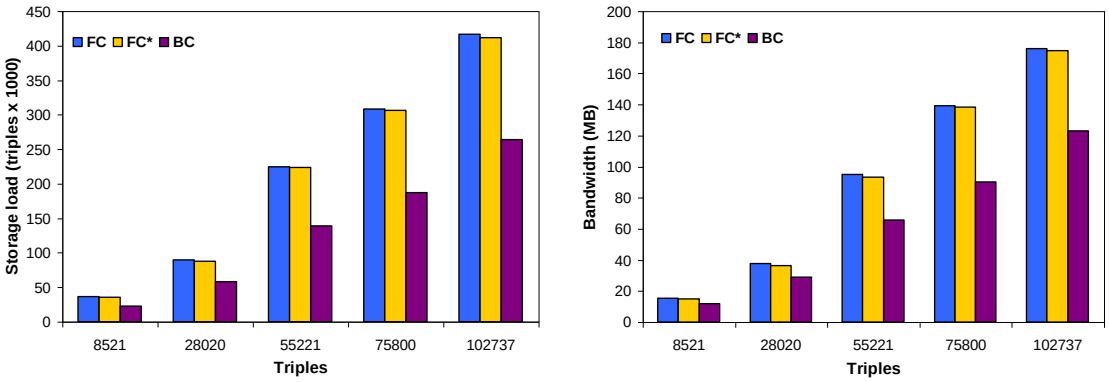
We have also conducted some measurements concerning the distribution of the database storage load for both algorithms. Figure 5.11 shows the storage load distribution among network nodes using Lorenz curves as proposed in [117]. The purpose of Lorenz curves is to show which percentage of the nodes hold which percentage of the total load. In the ideal case where all nodes share the same load, the curve is a straight diagonal line. The closer a Lorenz curve is to this diagonal, the better the load distribution is. In this figure, we show the database storage load of backward

and forward chaining for tree-depths 2 and 6. For readability reasons, we do not show the intermediate tree-depths. For example, for a hierarchy with depth 2, we see that, in BC, around 72% nodes share less than 50% storage load and the other 28% nodes have to deal with the other 50% load. In this graph, we observe that the deeper the hierarchy tree is, the better the load distribution is for both backward and forward chaining. This can be explained by the fact that the range of the object value of the triples stored is limited to the number of classes of the hierarchy. For a class hierarchy with depth 2, the number of distinct classes is 7 and nodes responsible for these classes are overloaded. As the depth increases, the number of classes increases exponentially and more nodes share the load resulting in a more balanced distribution. Furthermore, while both algorithms have almost the same load distribution for a tree with depth 2, BC distributes the load slightly better than forward chaining when the depth increases to 6. This is a result of a characteristic property of FC, namely that classes of higher levels of the hierarchy have more instances than classes from lower levels (since each class keeps all the instances of its subclasses). Therefore nodes that are responsible for classes of higher levels are more loaded with triples than nodes responsible for classes of lower levels. However, both approaches can take further improvements using well known load balancing techniques that distribute storage load more evenly among nodes [14, 78, 151, 170]. Such solutions can be applied to our algorithms but we consider this research area out of the scope of the thesis.

Let us now present our experiments using the LUBM dataset. For this set of experiments, we stored an increasing number of triples from the LUBM-1 dataset which consists of $102,737$ triples. We compare the behaviour of BC, FC and FC*. Figure 5.12 shows results regarding the store message load and the bandwidth consumption. The x-axis shows the number of triples initially inserted in the network. The results of this experiment are similar regarding the comparison of BC with FC and FC*. However, in this case, FC and FC* produce almost the same number of messages resulting in similar bandwidth consumption and cause almost the same load in the network, with FC* performance slightly better. The difference between FC and FC* in this case is not so evident because of the nature of the LUBM schema. LUBM schema includes only small RDFS class hierarchies whose depth is at most 2. Therefore, the number of redundant triples generated from FC is very small.

Semi-naive evaluation

We also performed some experiments to see the behaviour of a semi-naive evaluation as discussed in Section 5.1.4. We used both the RBench and LUBM benchmark and



(a) Message storage load

(b) Bandwidth

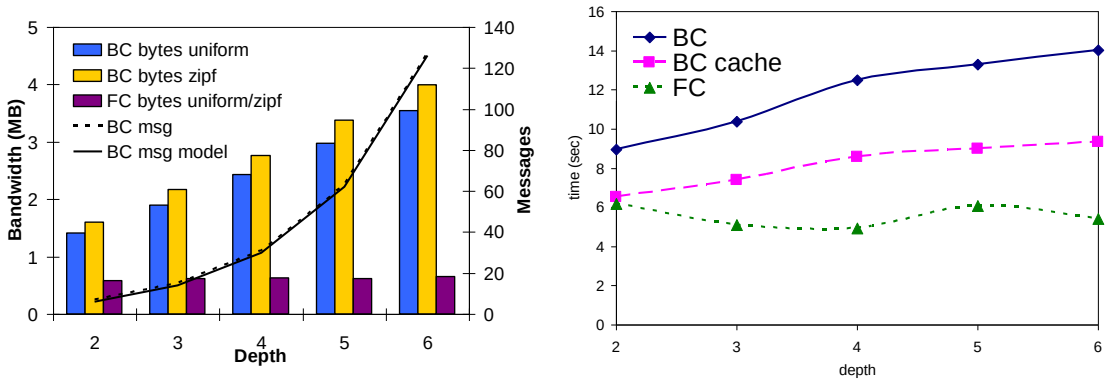
Figure 5.12: Storing LUBM-1

discovered that FC* outperformed the algorithm that uses semi-naive. For example, for the experiment with the RBench dataset shown in Figure 5.9, FC* required 6.48 minutes to reach a fixpoint while the algorithm that uses semi-naive required 9.76 minutes. This is caused by the fact that FC* with semi-naive evaluation sends more messages of smaller size (i.e., with fewer triples) which causes extra traffic to the network and puts extra processing load at the nodes that receive these messages. On the other hand, FC* is able to infer more triples at each inference step and hence sends bigger but fewer messages in the network. For the same experiment, 54 messages were sent by FC*, while 102 messages were sent in semi-naive evaluation.

5.5.3 Querying RDF(S) data

In this experiment, we present results while evaluating queries of the form "give me all instances of the root class". We store $10,000$ instances for both uniform and Zipfian distributions under an RDFS class hierarchy of varying depth of the RBench generator and pose the query. Measurements are averaged over 10 runs using the geometric mean which is more resilient to outliers.

Figure 5.13(a) shows two metrics concerning the network traffic; bandwidth (left y-axis) and messages sent (right y-axis). Since FC sends a single message regardless of the tree-depth, we do not depict it in the graph. In BC, the number of messages sent in the network is equal to the number of classes in the RDFS hierarchy since it sends one message for each class. Therefore, as the hierarchy becomes deeper, the number of classes increases and the total number of messages sent also increases. This was also shown by the analytical cost model of Section 5.4.2 and is depicted in the graph as



(a) bandwidth and messages

(b) query response time

Figure 5.13: Querying the root class

well. Figure 5.13(a) also shows the bandwidth used in the network for both approaches and for both types of instance distribution. FC bandwidth consumption is limited to the number of bytes sent for the delivery of the results of the query and is independent of both the tree-depth and the instance distribution. In comparison, the bandwidth of BC increases with the tree-depth as a result of the increasing number of messages that are sent in the network. Furthermore, a skewed instance distribution slightly affects the bandwidth used since more instances belong to lower classes and need to be sent towards the root class.

We have also implemented a caching optimization technique for the BC algorithm. Since there is a need to traverse the subclass hierarchy quite often, we could take advantage of the first time it is traversed and cache useful routing information. Assuming that different nodes are responsible for different classes in a hierarchy, we need to make $d * O(logn)$ hops to reach from a node responsible for the root class of the hierarchy to the nodes responsible for the leaf classes, where d is the depth of the hierarchy tree and n the number of the nodes in the network. We can minimize this by adding extra routing information to each node x which is responsible for a certain class c of the hierarchy. The first time a node x contacts a node y which is responsible for a subclass of c, it keeps the IP address of y and uses it for further communication. In this way, a direct subclass is found in just 1 hop and the whole hierarchy is traversed in d hops. This technique reduces network traffic and decreases query response time, while the overhead of maintaining such a table is not significant and it is only kept in memory. Figure 5.13(b) shows the query response time. A query needs constant time to be evaluated when FC takes place. This is reasonable, considering that only one

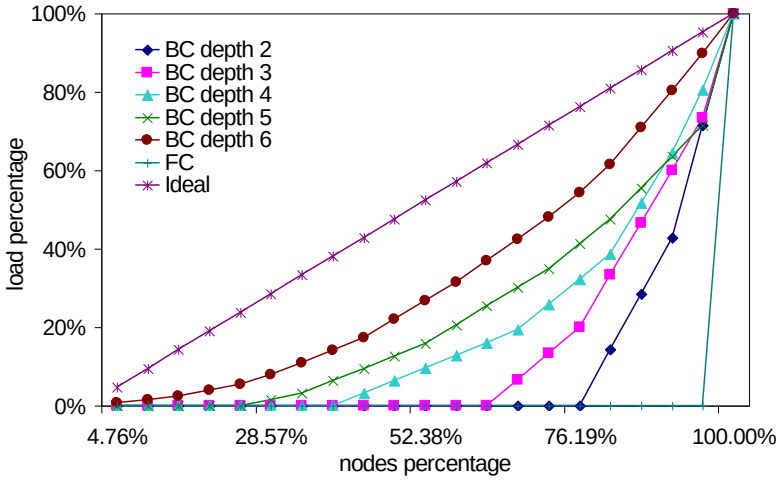


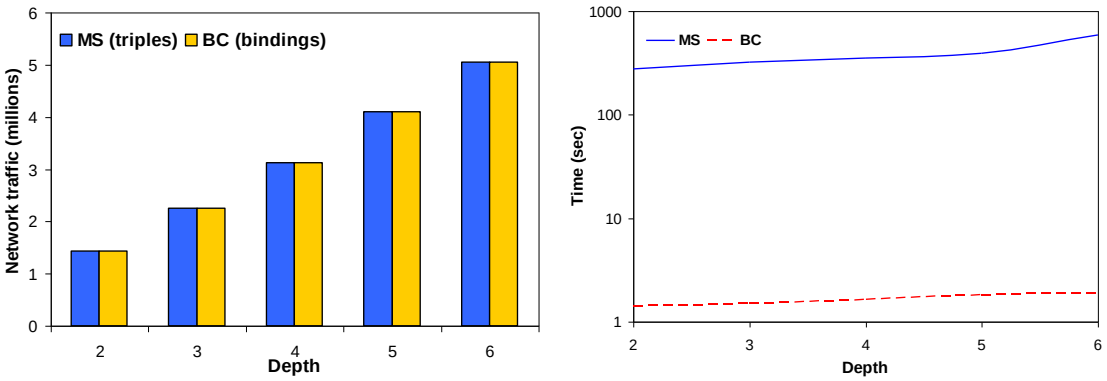
Figure 5.14: Query load distribution

node participates in the query processing, although the query processing it needs to handle is quite heavy. On the contrary, BC response time increases with the tree-depth but does improve significantly when the caching technique is used.

In addition, we conducted experiments regarding the query load distribution. As already shown in the analytical evaluation the total query load incurred is similar for both approaches and only the distribution differs. While FC involves a single node in the query processing, in BC the load is shared among the nodes that are responsible for the subclasses of the root class. Figure 5.14 shows the query load distribution using Lorenz curves. FC loads only one node in the network and shows the worst load distribution. We also depict in the figure how BC distributes the load for varying tree depth. BC shares the query load more evenly as the number of classes in the RDFS hierarchy increases. Similarly with the storage load, the same techniques can be applied here for balancing the query load among the nodes [14, 78, 151, 170] but this is out of the scope of our work.

5.5.4 Comparing backward chaining with magic sets

In this section, we compare the backward chaining algorithm (BC) with the algorithm using the magic sets transformation (MS) using as a testbed the local shared cluster. In the following set of experiments, we stored $1,000,000$ instances of the RBench dataset for RDFS class hierarchies of varying depth. Then, we send a request with the predicate `m_type` and argument the root class for MS, while we run the query that asks for all instances of the root class for BC.

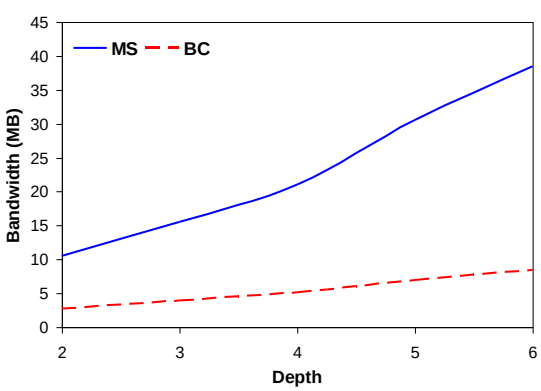


(a) Network traffic (b) Time

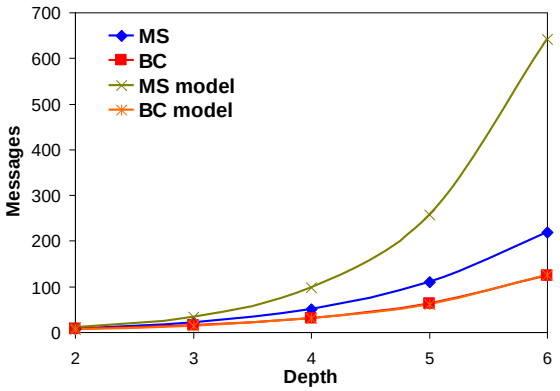
Figure 5.15: Network traffic and completion time for BC and MS

Figure 5.15(a) shows the network traffic in terms of the number of triples transferred in the network for MS and in terms of the number of bindings transferred for BC. The total number of triples generated and sent from MS and the total number of bindings generated from BC are equal. However, the amount of time required from MS to terminate is much greater than the time required from BC to answer the corresponding query. Figure 5.15(b) depicts the time difference. Note that y-axis shows the time in seconds on a logarithmic scale. We observe that BC outperforms MS by two orders of magnitude. The reasons for this are explained below.

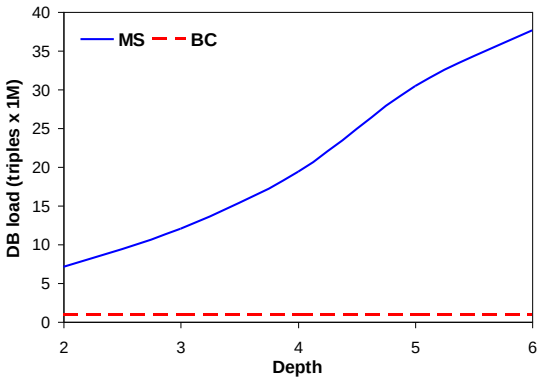
Firstly, the messages sent during MS contain whole triples, while the messages sent during BC contain only bindings, i.e., only the object of the matching triples. This fact is depicted in Figure 5.16(a) by the bandwidth consumption of the two algorithms. We observe that the total bandwidth consumed by MS is about three times greater than the total bandwidth spent by BC. In addition, Figure 5.16(b) shows the number of store messages (StoreMsg) sent by MS containing the inferred triples and the number of response messages (BwdRDFSResp) sent by BC containing the bindings of matching triples, as well as the predicted numbers of messages of both algorithms computed by the analytical model. BC sends exactly the number of messages computed by the analytical model, while MS sends less messages compared to the number of messages computed by analytical model due to the MultiPut functionality. Although Figure 5.15(a) showed that the total number of values (i.e., triples or bindings) sent by both algorithms is equal, Figure 5.16(b) shows that MS sends more messages than BC. This results from the nature of the algorithms. In BC, a node sends a response back to its parent node only if it has collected all answers from its children. On the contrary, in



(a) Bandwidth consumption



(b) Messages sent

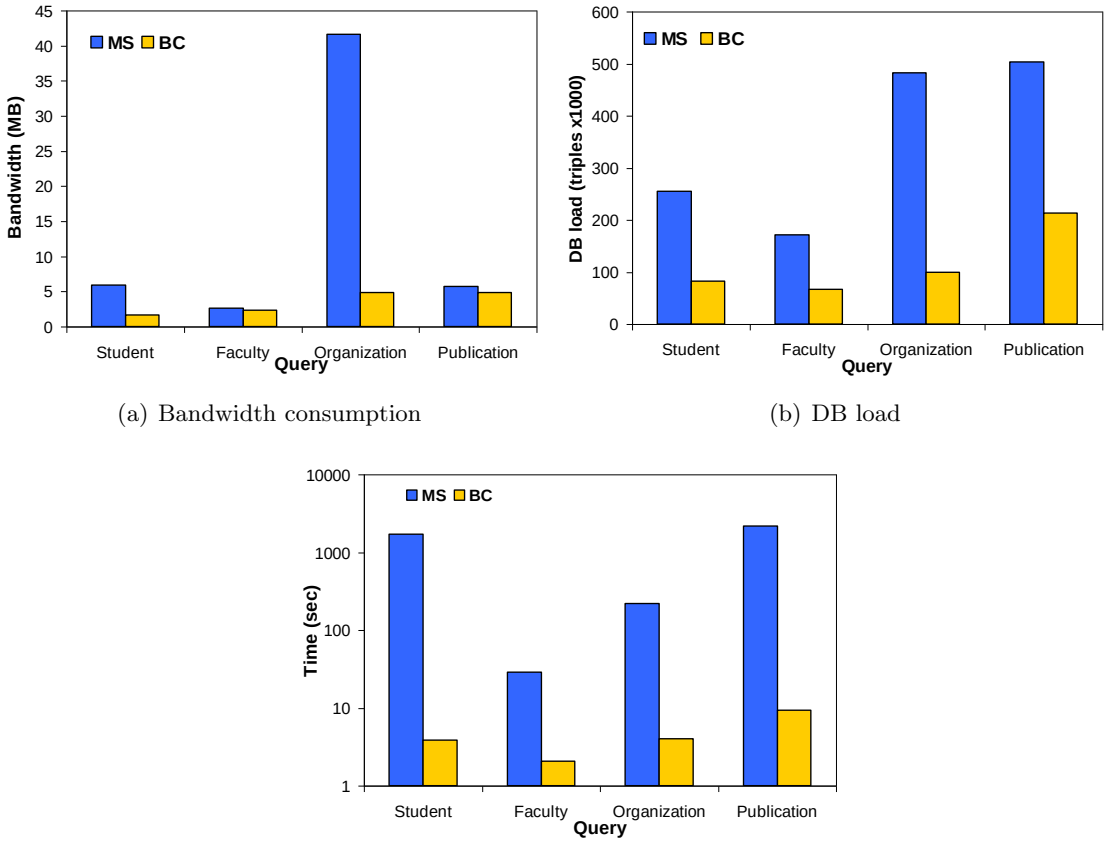


(c) Database load

Figure 5.16: MS vs. BC for RBench dataset

MS whenever a node receives a MSReq, it sends a StoreMsg message if any new triples were generated. As a consequence, the load incurred in the database of each node is affected. In both algorithms, each time a node receives a message, it retrieves from its local database matching triples. We define as *DB load* of a node the number of triples that the node retrieves from its local database. Figure 5.16(c) shows the total DB load incurred in all the nodes of the network for both algorithms. While BC incurs a constant DB load regardless from the depth of the RDFS hierarchy, DB load in MS is increasing significantly with the depth of the RDFS class hierarchy and is much greater than in BC. As a conclusion, BC outperforms MS due to the DB load incurred and the bandwidth consumed by MS.

The RBench dataset involves only one RDFS class hierarchy and hence the outcome of MS when computing all the instances of the root class is similar with the outcome



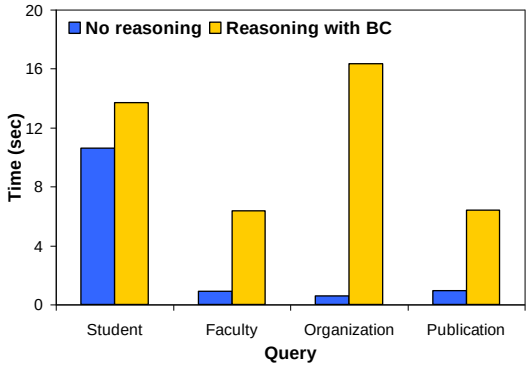
(a) Bandwidth consumption

(b) DB load

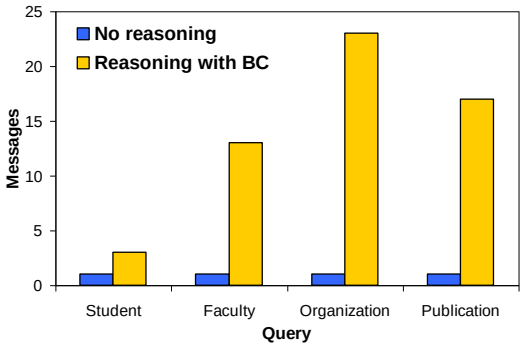
(c) Time

Figure 5.17: MS vs. BC for LUBM-20

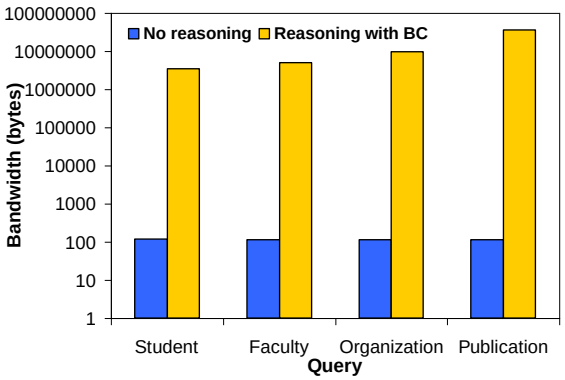
of FC*. In the next experiment, we use the LUBM dataset whose schema contains several independent class hierarchies. We created a network of 156 nodes in the cluster and stored the complete LUBM-20 dataset consisting of $2,782,435$ triples. We want to retrieve the instances of the following classes of the LUBM schema: `ub:Student`, `ub:Faculty`, `ub:Organization`, `ub:Publication`, where `ub` is the appropriate namespace. For MS, we sent a request with the predicate `m_type` and argument the class name, while for BC we run the query that asks for all instances of the respective class. Figure 5.17 shows the bandwidth consumption, the total DB load and the time required for each algorithm to terminate. In this experiment as well, we observe that BC outperforms MS. Since we deal with a bigger dataset the advantage of using BC is more evident, as shown in Figure 5.17(c).



(a) Query Response time



(b) Messages sent

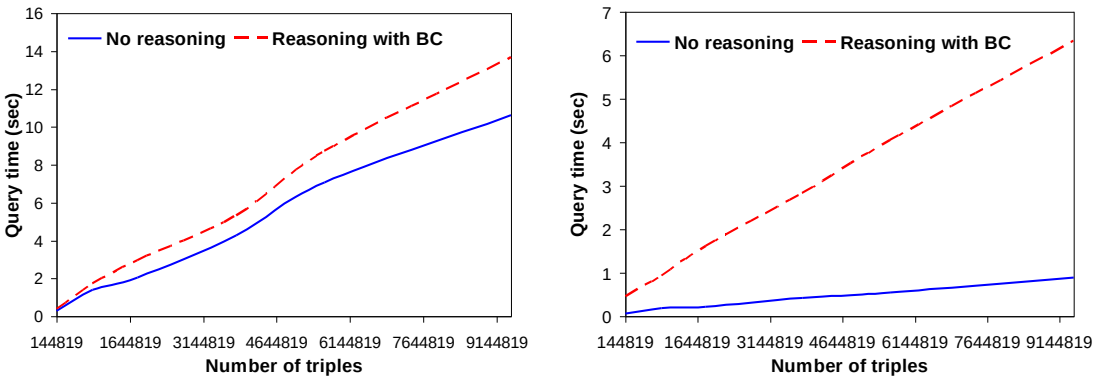


(c) Bandwidth consumption

Figure 5.18: BC performance in cluster for LUBM-50

5.5.5 Backward chaining performance in the cluster

Since we could not further examine the behaviour of a forward chaining approach for bigger datasets, in this set of experiments, we explore the performance of the backward chaining algorithm during query evaluation in the local cluster. First, we create a network of 156 nodes and use the LUBM-50 dataset in two different cases. In the first case, we store the dataset together with the inferred triples (9, 437, 221 triples in total) and then run the query without any reasoning involved. In the second case, we store only the initial dataset (6, 890, 949 triples in total) and use backward chaining during query evaluation to return a complete answer to the query. The queries we used ask for the instances of the following classes of the LUBM schema: `ub:Student`, `ub:Faculty`, `ub:Organization`, `ub:Publication`. Figure 5.18 depicts the results of different metrics for these queries.



(a) Query `ub:Student` (b) Query `ub:Faculty`

Figure 5.19: Increasing the number of triples stored

In Figure 5.18(a), we depict the time required to answer each query, i.e., the time from the moment a node receives the query request until the moment it receives all the answers. Depending on the RDFS schema, BC sends a different number of messages to retrieve the required inferences. On the contrary, just one message is sent during query evaluation without reasoning to a single node which retrieves all results from its local database. The number of messages is depicted in Figure 5.18(b). For example, LUBM schema does not contain many inferences for class `ub:Student` and thus for the query that asks for the instances of class `ub:Student` the number of messages sent by BC is only 3. On the other hand, for the query that asks for the instances of class `ub:Faculty`, BC sends 13 messages to retrieve all inferences. Certainly, this leads to a greater query response time difference between the algorithm that uses backward chaining and the algorithm where no reasoning is used, but still query response times of both approaches are in the same order of magnitude. As a result of the number of messages sent, bandwidth consumption is also greater when backward chaining is used. Figure 5.18(c) shows the bandwidth consumed because of the partial answers that are transferred in the network. Another parameter that affects query response time is the number of results that each query returns. We see that the query response time for class `ub:Student` is greater than the query response time for class `ub:Faculty`. This is because the number of results for query `Student` is $430,114$ while it is $35,973$ for `Faculty`.

We have also experimented with different dataset sizes. In a network of 156 nodes, we stored datasets from LUBM-1 to LUBM-50. Every time we measured the query response time of queries that ask for the instances of `ub:Student` and `ub:Faculty`.

Table 5.5: LUBM datasets and queries

Dataset	**Triples**	**Inferred triples**	**Answers of query** `ub:Student`	**Answers of query** `ub:Faculty`
LUBM-1	103,413	144,819	6,463	540
LUBM-5	646,144	887,461	40,087	3,373
LUBM-10	1,317,009	1,806,023	82,507	6,843
LUBM-20	2,782,435	3,812,865	174,750	14,457
LUBM-30	4,109,311	5,629,144	256,919	21,440
LUBM-50	6,890,949	9,437,221	430,114	35,973

Table 5.5 shows the sizes of all the datasets used together with the number of results that each query returns. Figure 5.19 shows the behavior of our system with and without the reasoning process as the dataset stored in the network grows. In the graphs of Figure 5.19 we show that query response time scales in a linear fashion with the number of triples stored in the network. This is a result of two factors. First, the local database of each node grows and as a result local query processing becomes more time-consuming. Second, the size of the answer set of the queries grows as the number of triples is increasing resulting in a greater bandwidth consumption. As we mentioned before, BC performs close to the query evaluation without any reasoning for the query that asks for the instances of `ub:Student`, while the difference becomes more evident for the query that asks for the instances of class `ub:Faculty`. In all cases, the increase of query response time of backward chaining remains linear with respect to the triples stored.

5.6 Summary

In this chapter, we presented and evaluated both bottom-up and top-down algorithms for RDFS reasoning on top of DHTs. We proved the correctness of our forward and backward chaining algorithms utilizing the minimal deductive system *mrdf* presented in [101]. In addition, we designed and implemented an algorithm which works in a bottom-up fashion using the magic sets transformation technique [17]. We provided a comparative study of our algorithms both analytically and experimentally. In the experimental evaluation, we deployed our system both in PlanetLab and in a local cluster.

As a conclusion, the forward chaining approach improves the time for answering a query but increases the storage load significantly by generating statements that might

never be required in a query. One might prefer to pay this storage cost if enough space is available and quick answering is paramount. Naturally, this method is the preferred one if one wants to compute the complete closure of a given dataset under RDFS entailment. Results from our experiments show that a simple forward chaining implementation such as FC presented in [74] and also presented in BabelPeers [16] cannot scale to more than a few thousand triples in a DHT environment, while the FC* algorithm we presented in this chapter improves scalability to hundreds of thousands of triples. In contrast, the backward chaining approach improves storage load and can scale to millions of triples. Certainly, this comes at the cost of an increase in query response time. Yet, the query response time of backward chaining remains in the same order of magnitude with the one of forward chaining and increases linearly with the number of triples stored in the network. A magic sets algorithm tries to exploit the advantages of both approaches and scores between the two.

As it has been proven recently, approaches that are based on distributed computing platforms consisting of powerful clusters and cloud computing platforms using MapReduce, such as [163, 166], can be very scalable for computing the closure of RDF(S) graphs and hence should be preferred if appropriate infrastructures are available. However, these approaches do not deal with the query answering problem, which is equally important.

In the next chapter we present a complete query processing algorithm which can be used to answer more complex queries than the ones presented here. The algorithm can be used with either a forward chaining reasoning process where all inferences have been precomputed, or it can combined with the backward chaining algorithm and compute inferences at query run time.

Chapter 6

Distributed RDF Query Processing

In Chapter 5 we focused on RDFS reasoning and how atomic SPARQL queries consisting of a single triple pattern can be evaluated on top of a DHT. In addition, in Section 4.2.2 we described the basic protocol used in Atlas to answer a query consisting of a single triple pattern. In this chapter, we present a complete query evaluation process which is used to answer arbitrary SPARQL queries of basic graph patterns in Atlas. The algorithm can be implemented on top of any DHT network.

The query processing algorithm presented, called QC*, is described in Section 6.1. This algorithm is concerned with RDF data only and does not deal with RDFS reasoning. If a forward chaining reasoning algorithm has taken place as described in Section 5.1, then QC* can retrieve all inferred information as well. For the case where inferences have not been computed a priori in the network, we propose combining the backward chaining algorithm of Section 5.2 with QC* to enable RDFS reasoning. The resulted algorithm is called QC*-BC and is presented in Section 6.2. Furthermore, in Section 6.3 we describe a distributed mapping dictionary we have designed and integrated in our algorithms to further increase our system's performance. Finally, Section 6.4 shows the experimental results of the query processing algorithm and the distributed mapping dictionary.

6.1 Query Processing Algorithm

In this section, we describe the query processing algorithm of Atlas, called QC*. QC* is based on algorithm QC presented in [75, 89] and extends it with the query

Algorithm 4: QC*: Query evaluation algorithm

```
1  event p.QEval (id, g, interRes, vars, retIP) from m
2      lR=Match (g.marked_node().triplepattern());
3      if interRes = ∅ then interRes = lR;
4      else
5          interRes' = lR ⋈ interRes;
6      end
7      if interRes' = ∅ then
8          sendto retIP.QResp(∅);
9          return;
10     end
11     g' := MERGE(g.hypernode(), g.marked_node());
12     if g'.nodes() = ∅ then
           // all triplepatterns are evaluated
13         answer = π_vars(interRes');
14         sendto retIP.QResp(answer);
15         return;
16     end
17     project out unnecessary vars from interRes';
18     MarkNextNode(g');
19     tp' := g'.marked_node().triplepattern();
20     key' = FindKey(tp');
21     id' = Hash(key');
22     sendto id'.QEval(id', g', interRes', vars, retIP);
23 end
```

graph representation introduced in Section 3.3.2. Unlike QC which uses lists of triple patterns, QC* employs the query graph representation which ensures that no Cartesian product will be computed and transferred through the network.

Query evaluation starts when peer x receives a request QueryRequest(q). The SPARQL translator of peer x translates the query q to a set of query graphs G and delivers G to the optimizer. For each graph $g \in G$, based on the query plan generated by the optimizer (details of how this is done are explained in Chapter 7), the peer also marks the node of the query graph which represents the triple pattern that should be evaluated first and sends a `QEval` message to the peer that will start the query evaluation. If more than one query graphs where created, peer x sends the relevant messages in parallel, gathers all the result sets and produces the final answer by computing the Cartesian product of these result sets. Notice that the Cartesian product is computed only at peer x and is not transmitted in the network.

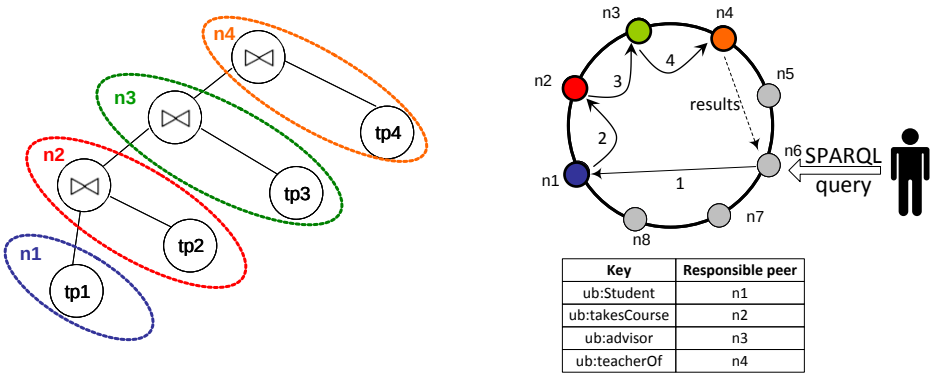
As in Chapter 5, we use the following notation for the description of the algorithm. Keyword **event** precedes every event handler for handling messages, while keyword **procedure** declares a procedure. In both cases, the name of the handler or the proce-

dure is prefixed by the node identifier in which the handler or the procedure is executed. Local procedure calls are not prefixed with the node identifier. Keywords **sendto** and **receive** declare the message that we want to send to a node with known either its identifier (thus DHT routing will be used) or the IP address (thus the node is immediately contacted), and the message we receive from a node respectively.

Algorithm 4 shows the pseudocode when a `QEval` message arrives at peer p. The message contains the identifier id which led to peer p, the query graph g representing the query, the intermediate results $interRes$ so far (initially empty), the answer variables of the query $vars$ and the IP address $retIP$ of the peer that received the request QueryRequest.

First, peer p evaluates the triple pattern which correspond to the marked node of query graph g and forms a temporary relation lR by posing a selection query to its triple relation. If relation $interRes$, which holds the intermediate results so far, is empty, peer p is the first peer of the query evaluation and assigns lR to $interRes'$. Otherwise, p assigns to $interRes'$ the natural join of lR and $interRes$. If the result of the join is an empty relation, p returns an empty set to the peer that posed the query (peer with IP address $retIP$) and query evaluation terminates. Otherwise, query evaluation continues and peer p merges the marked node with the hypernode in g creating a new query graph g'. In case peer p is the first peer participating in the query evaluation, p just transforms the marked node n into a hypernode. If the new graph g' consists only of a hypernode, all triple patterns have been evaluated and p computes the projection of $interRes'$ on the answer variables $vars$ and sends the answer to the peer with IP address $retIP$. Otherwise, query evaluation continues and p projects out from $interRes'$ variables that neither appear in $vars$ nor in the rest of the triple patterns. Then, a new node in g' is marked as the next triple pattern that should be evaluated and p sends a new `QEval` message to the next peer. Local procedure MarkNextNode ensures that the chosen node is connected with the hypernode of g', so that no Cartesian product will be computed.

The general idea behind QC* is that the algorithm evaluates one triple pattern at a time transferring the intermediate results from one peer to the other. Intermediate results are joined at the peer responsible for finding matching triples for the next triple pattern. This algorithm results in left-deep execution plans. As an example, assume the SPARQL query shown below in Listing 6.1 which is part of the complete query Q9 of the LUBM benchmark [48]. The query asks for the students that take courses taught by their advisors.



Key	Responsible peer
ub:Student	n1
ub:takesCourse	n2
ub:advisor	n3
ub:teacherOf	n4

Figure 6.1: Example of QC* algorithm

Listing 6.1: SPARQL query

```
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX ub:<http://www.lehigh.edu/~zhp2/2004/0401/univ-bench.owl#>
SELECT ?x ?y ?z
WHERE {
    ?x rdf:type ub:Student . (tp1)
    ?x ub:takesCourse  ?z .  (tp2)
    ?x ub:advisor  ?y.       (tp3)
    ?y ub:teacherOf  ?z   .  (tp4)
}
```

Figure 6.1 shows how this query can be evaluated in a network of DHT peers. For the sake of simplicity, let us assume that the triple patterns will be evaluated at the given order. Chapter 7 discusses optimization algorithms that choose an appropriate ordering. The peer that receives the query request sends the query to the node that is responsible for evaluating the first triple pattern of the query. In this case, it is node n_1 which is responsible for the key `ub:Student`. Node n_1 finds values from its local database that match the first triple pattern and sends them to node n_2 which is responsible for evaluating the second triple pattern. Node n_2 finds matches for the second triple pattern and joins them with the intermediate results from node $n1$. Similarly nodes n_3 and n_4 continue with the query evaluation and node n_4 which is the final node in the query chain sends the answer of the query to the node that received the query request.

Algorithm 5: QC*-BC: Query evaluation algorithm with backward chaining

```
1  event n.QEval (id, g, interRes, vars, retIP, rid)
   from m
2      tp := g.marked_node().triplepattern();
3      Let p^a be the adorned atom of tp;
4      localR=BC-RDFS (p^a, rid);
5      if interRes = ∅ then
6          interRes = localR;
7      else
8          interRes' = localR ⋈ interRes;
9      end
10     if interRes' = ∅ then
11         sendto retIP.QResp(∅);
12         return;
13     end
14     g' := MERGE(g.hypernode(), g.marked_node());
15     if g'.nodes() = ∅ then
           // all triplepatterns are evaluated
16         answer = π_vars(interRes');
17         sendto retIP.QResp(answer);
18         return;
19     end
20     project out unnecessary vars from interRes';
21     MarkNextNode(g');
22     tp' := g'.marked_node().triplepattern();
23     key' = FindKey(tp');
24     id' = Hash(key');
25     sendto id'.QEval(id', g', interRes', vars, retIP);
26 end
```

```
1  procedure p.BC-RDFS (p^a, rid)
2      if rid + p^a ∈ processedRequests then return ∅;
3      add rid + p^a to processedRequests;
4      adornedRules =ApplyRule(p^a);
5      forall the rules in adornedRules do
6          r ← RemoveFirst(adornedRules);
7          if r has one predicate then
8              R=MatchPredicate (p^a);
9          else
10             Let q1 be the adorned predicate of
               r.body with a k element in its
               adornment and q2 the other predicate
               of r.body;
11             if q1 is the edb triple then
12                 R' = MatchPredicate (q1);
13             else
14                 R' = BC − RDFS(q1, rid);
15             end
16             if R' = ∅ then return R;
17             for each value v_i of the shared variable
               Z in R' do
18                 id_i =Hash (v_i);
19                 rewrite q2 to q2';
20                 sendto id_i.BC-RDFSReq(q2')
21                 receive BC-RDFSResp(R_i) from id_i
22                 R = R ∪ R_i;
23             end
24         end
25     end
26     return R ;
27 end
```

```
1  event n.BC-RDFSReq (p^a, rid) from m
2      R=BC-RDFS (p^a, rid);
3      sendto m.BC-RDFSResp(R)
4  end
```

6.2 Evaluating SPARQL Queries using Backward Chaining

In Section 5.2 we designed a backward chaining algorithm for RDFS entailment in the distributed environment of a DHT. We focused mainly on atomic SPARQL queries consisting of one triple pattern. On the other hand, in the previous section we presented algorithm QC* which can return complete answers with respect to the RDFS entailment rules only if all the inferences have been computed a priori, i.e., in a forward chaining manner. In this section, we bridge the gap between these two algorithms and present algorithm QC*-BC. QC*-BC enables the evaluation of arbitrary SPARQL queries of basic graph patterns with RDFS entailment using a backward chaining approach.

Algorithm 5 shows the pseudocode of algorithm QC*-BC. The main idea of this

algorithm is that each triple pattern of a query graph is evaluated at a (potentially) different peer. At this peer a backward chaining process as described in Section 5.2 takes place for the triple pattern using the rules of Table 5.2. After the backward chaining process has terminated, the results are joined with the intermediate results from previous triple patterns. The evaluation of each triple pattern starts at the peer which is responsible for a key contained in the triple pattern, as we described in Section 4.2.2.

The left part of Algorithm 5 shows the pseudocode when a `QEval` message arrives at peer p. The message is similar with Algorithm 4 with an extra parameter rid which is a unique identifier for the query request. The unique identifier can be created by a number assigned by a local counter and the IP address of the peer that received the query request. rid is useful for the backward chaining algorithm to handle cyclic hierarchies, as explained in Section 5.2. Peer p firstly transforms the triple pattern that corresponds to the marked node of graph g to the equivalent adorned datalog atom and calls local procedure `BC-RDFS`. After the backward chaining process terminates, procedure `BC-RDFS` outputs a new relation $localR$ which contains all the bindings of the triple pattern's variables including the inferred ones.

The rest of the query evaluation process proceeds similarly with the QC* algorithm. The right part of Algorithm 5 shows the pseudocode for the procedure `BC-RDFS` which illustrates how the backward chaining algorithm works. This procedure is the same with the one we introduced in Section 5.2 and is also shown here for completeness.

6.3 Mapping Dictionary

Mapping dictionaries have been recently used in many centralized RDF stores [35, 108]. As URIs and literals may consist of long strings, they are mapped to integer values and then, triple storage and query evaluation is performed using these integer values. This enhances the system's performance since operations with integers are more efficient than operations with strings. We have adopted a similar mapping dictionary functionality carefully designed for the distributed setting of a DHT. In this way, our algorithms work similarly using the integer values.

The uniqueness of the integer values used in the mapping dictionary could be ensured in various ways. The simplest way would be the use of some kind of global counter for assigning constants with globally unique identifiers. This global counter could be maintained by a single peer or a subset of the peers in the network. However, this approach would require extra communication cost between the peers and would create

a bottleneck and a single point of failure. Therefore, we propose the following scheme which is fully distributed (thus scalable and fault tolerant) and does not require any kind of coordination among the peers. Each peer keeps a local integer counter consisting of l bits which is initially set to 0. l is incremented by 1 every time a new integer value needs to be generated. Each peer that joins the network is assigned a unique m-bit identifier by hashing its IP address. We create an n-bit identifier for a triple component by concatenating the m bits of the peer's unique identifier with the l bits of the current local counter. Depending on the network setting and the application requirements, we can determine an appropriate value for m and l so that each n-bit identifier is of reasonable space. For a maximum number of N peers in the network, where each peer can insert up to L new components, $log_2(N * L)$ bits are required. For example, for a network of 1000 peers, m should be equal to 12 bits ($log_2(1000) \simeq 10$ plus 2 bits to avoid conflicts). If then we choose the size of the counter l to be equal to 20 bits, each peer can insert up to $2^{20} \simeq 1M$ new components. Therefore, each peer creates identifiers of 32 bits and the size of each identifier remains the same with the size of an integer value. In the worst case scenario where a node runs out of unique integer values, it can forward the request to one of its neighbors.

When a peer receives a store request, it transforms the given triples into new triples containing integers. The peer, then, sends the new set of triples to be stored in the network using multiSend. Together with the new triples, it also sends the mapping from strings to integers that created these triples. Note that we use the *string values* of the triple's components as keys to create the identifiers and not their integer values. Each peer that receives a multiSend message, stores in its triple relation the triples it is responsible for, which now consist of integers. Each such peer also maintains a two-column relation which serves as a local dictionary which holds the mappings for *all* the components of its local triples (*dictionary relation*).

During query evaluation, each string appearing in the triple patterns of a query is transformed into the corresponding unique integer. This transformation is performed during the lookup operation of the query processing algorithm without adding any significant overhead. Whenever peer s wants to send a `QEval` message for triple pattern tp, it first sends a Lookup request to determine the peer responsible for this triple pattern (peer d). Peer d, which receives the Lookup request, retrieves the integers corresponding to the strings of tp from its local dictionary relation and sends them to s together with its IP address. Then, peer s replaces the strings of tp with the integers sent from peer d and continues query processing. In case any of the strings has no assigned integer, the answer to the query is empty and query processing terminates.

Finally, the peer which computes the answer to the query is responsible for replacing integers in the triples of the answer set with their string values. It contacts the least possible number of peers that have already participated in the query evaluation and have the appropriate values in their dictionary relation. To achieve this, the IP address of the peers participating in the query evaluation is appended within the `QEval` message (using an extra parameter).

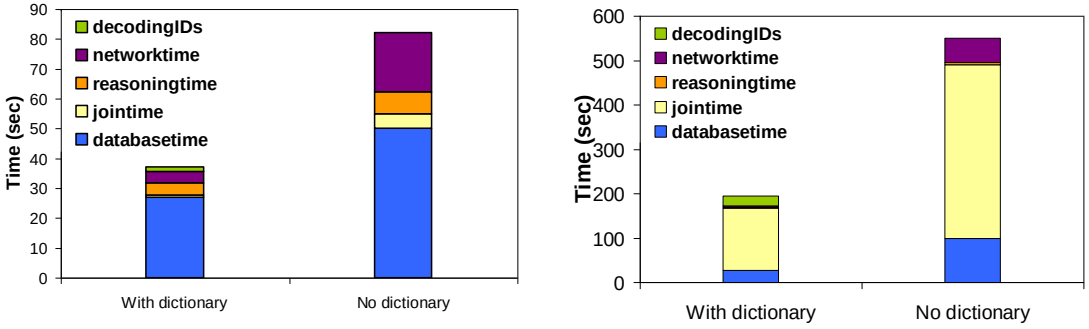
6.4 Experimental Evaluation

In this section, we present an experimental evaluation of our query processing algorithm and the mapping dictionary. For our experiments, we used as a testbed both the PlanetLab network and the local cluster. We used PlanetLab with 281 nodes across four continents that were available and lightly loaded at the time of the experiments. The cluster consists of 41 computing nodes, each one being a server blade machine with two processors at 2.6GHz and 4GB memory. We used 30 of these machines where we run up to 4 peers per machine, i.e., 120 peers in total.

For our evaluation, we use the LUBM [48] benchmark, described in Section 5.5, that provides synthetic RDF datasets of arbitrary sizes and 14 SPARQL queries. We mostly focus on queries with more than 4 triple patterns so that the benefits of the proposed optimizations can be clearly demonstrated. The SPARQL queries we used can be found in Appendix A. For each experiment, we show our results using the most representative LUBM queries. All measurements are averaged over 10 runs using the geometric mean which is more resilient to outliers.

6.4.1 Analyzing query response time

This set of experiments were conducted in an earlier version of Atlas when the Berkeley DB was used which was included in the Bamboo DHT implementation. The version of Berkeley DB in the Bamboo implementation is not a relational or an object-oriented database (although one could use it to build a relational database), and does not support any kind of high-level query languages, such as SQL. However, it provides a simple API in order to store and retrieve data efficiently. Data in the Berkeley DB are stored in records which are pairs in the form of (key, value) pairs. The following operations on records are supported: insert a pair in a table, delete a pair from a table, find a pair in a table by looking up its key and update a pair that has already been found. For this reason, we have implemented in Java several operators from relational algebra, e.g., join, select, project, on top of the Berkeley DB. As we will see from the results below



(a) Query Q4

(b) Query Q8

Figure 6.2: Query response time breakdown

such an implementation has not been proved efficient and therefore decided to move to a relational database instead.

First, we investigate our system performance in PlanetLab. We evaluate our query processing algorithm QC*-BC in PlanetLab and clearly demonstrate the importance of having a mapping dictionary. We analyze how the time for answering for a query is spent at the different phases of the query evaluation algorithm.

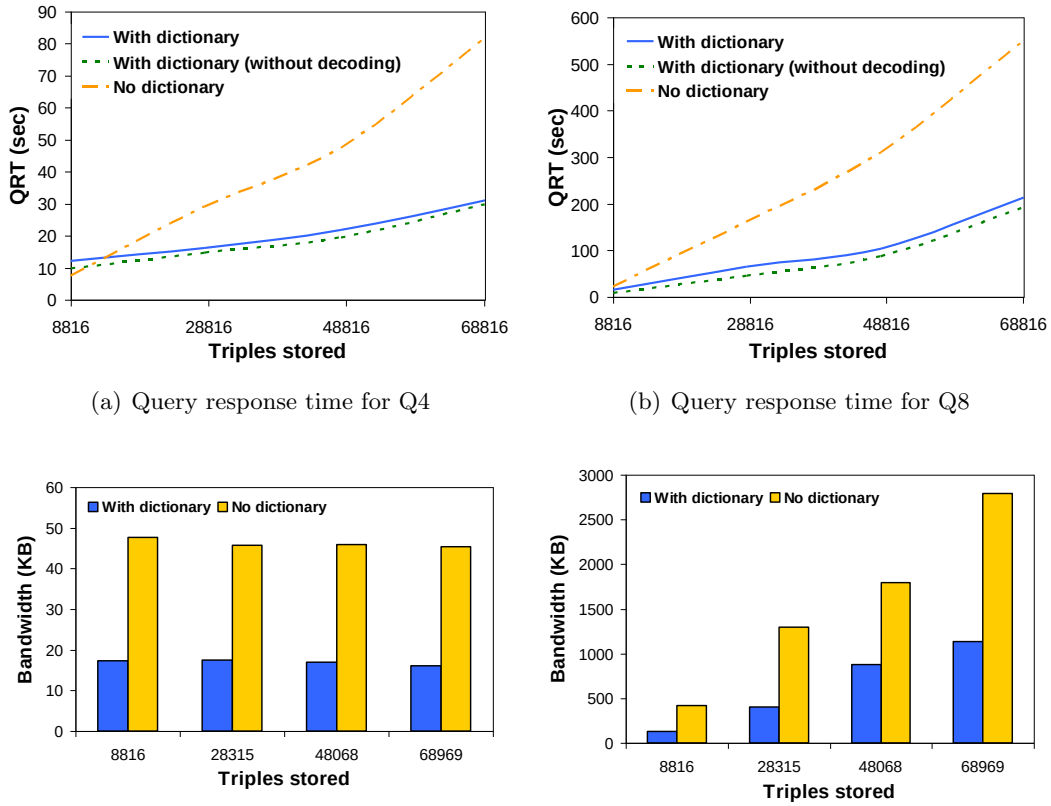
The total query response time of a query consists of the sum of the local processing time of each peer participating in the query processing plus the network time for transmitting the intermediate and the final results. The local processing time at each peer consists of the time for retrieving the triples matching a triple pattern from the local database using the lookup operation provided by Berkeley DB, the time for joining the matching triples with the intermediate results and the time needed for the RDFS reasoning using the adorned rules of Table 5.2. In the case where the mapping dictionary is used, we should also take into consideration the time for decoding the integer values to RDF components at the end of the query evaluation. Figure 6.2 presents two stacked bar graphs that show the breakdown of the total query response time with and without the mapping dictionary for the LUBM queries Q4 and Q8 after having stored $68,969$ triples. We observe that the time spent by each process of the query evaluation algorithm when the dictionary is used is smaller than the respective one when no dictionary is used. Therefore, the time required to decode the integer values to the corresponding RDF components does not impose a time overhead to the total query evaluation process. The same conclusion was drawn for the other LUBM queries as well.

Another important observation made from this set of experiments is that the most time-consuming operation in the query evaluation is retrieving triples from the peers' database and joining them with the intermediate results so far. From our experiments, we found that the time required to retrieve a single triple from a peer's local database ranged from $0,1ms$ to $128ms$ for the different peers in Q4 and from $0,18ms$ to $2096ms$ in Q8 at the time of the experiments. This phenomenon can be caused by the different sizes of the peers' databases as well as the slow nodes in PlanetLab. As demonstrated in [123], slow nodes in PlanetLab can cause such problems. In addition, our implementation of the join between the matching triples and the intermediate results is also a bottleneck in the case of query Q8 which produces more intermediate results. For these reasons, we decided to use a relational database and use SQL queries to perform these joins. From now on, we present experiments where Atlas peers use SQLite as the database backend.

6.4.2 The mapping dictionary effect in PlanetLab

We compare the query processing algorithm QC*-BC with and without the mapping dictionary functionality in PlanetLab. We measure the query response time and the bandwidth usage in both cases as the number of triples stored in the network grows. Due to the performance of the algorithm without the mapping dictionary, we show experiments with only up to $68,969$ triples, which correspond to part of the LUBM-1 dataset. The results for queries Q4 and Q8 is shown in Figure 6.3.

The query response time (QRT) for queries Q4 and Q8 is shown in the graphs of Figures 6.3(a) and 6.3(b). The x-axis shows the number of triples stored in the network. The blue continuous line shows the total query response time, while the green dashed line denotes the time required to answer the query without the time needed for decoding the RDF components from the identifiers returned in the result set. We observe that the slope of the line *With dictionary* is smaller than the slope of the *No dictionary* line. This means that the pace with which the query response time grows when no mapping dictionary is used is increasing more rapidly than when a mapping dictionary is used. As a result, the system becomes more scalable with respect to the number of triples stored in the network when a mapping dictionary is used. Another interesting result is that when less than about $10,000$ triples are stored in the network the query response time without the mapping dictionary for query Q4 is smaller than the query response time with the dictionary. This means that the overhead imposed from encoding and decoding the RDF components is significant and results in a worse system's performance. However, this happens only for a very small amount of triples.



(a) Query response time for Q4

(b) Query response time for Q8

(c) Bandwidth usage for Q4

(d) Bandwidth usage for Q8

Figure 6.3: Mapping dictionary effect in PlanetLab

Figures 6.3(c) and 6.3(b) show the total bandwidth usage for both queries. The bandwidth with the dictionary shown in the graphs is the total bandwidth spent and thus includes the bandwidth spent for decoding the final results from integer values to the corresponding RDF components. Q4 produces a constant number of results regardless of the number of stored triples. This number is determined from the first join of the first two triple patterns and hence, bandwidth consumption remains almost the same. Query Q8 also produces a constant number of results regardless of the number of triples stored. However, the size of the intermediate results of Q8 grows with the number of triples stored and hence bandwidth usage is also increasing. In both queries, we observe that the query processing algorithm when using the mapping dictionary achieves a significantly lower bandwidth consumption than when URIs and literals are used. In addition, the gain of transmitting integer values in the network as intermediate results is greater than the overhead bandwidth imposed when decoding

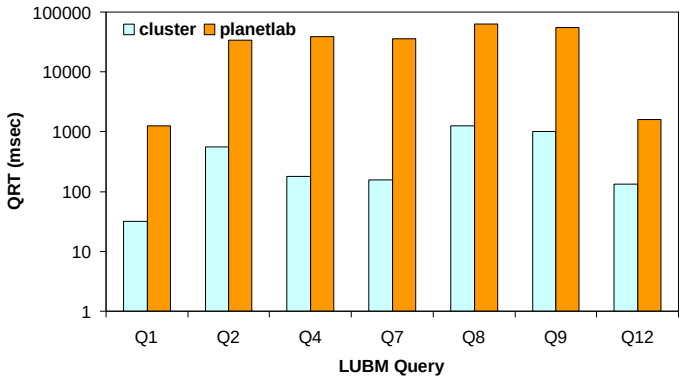
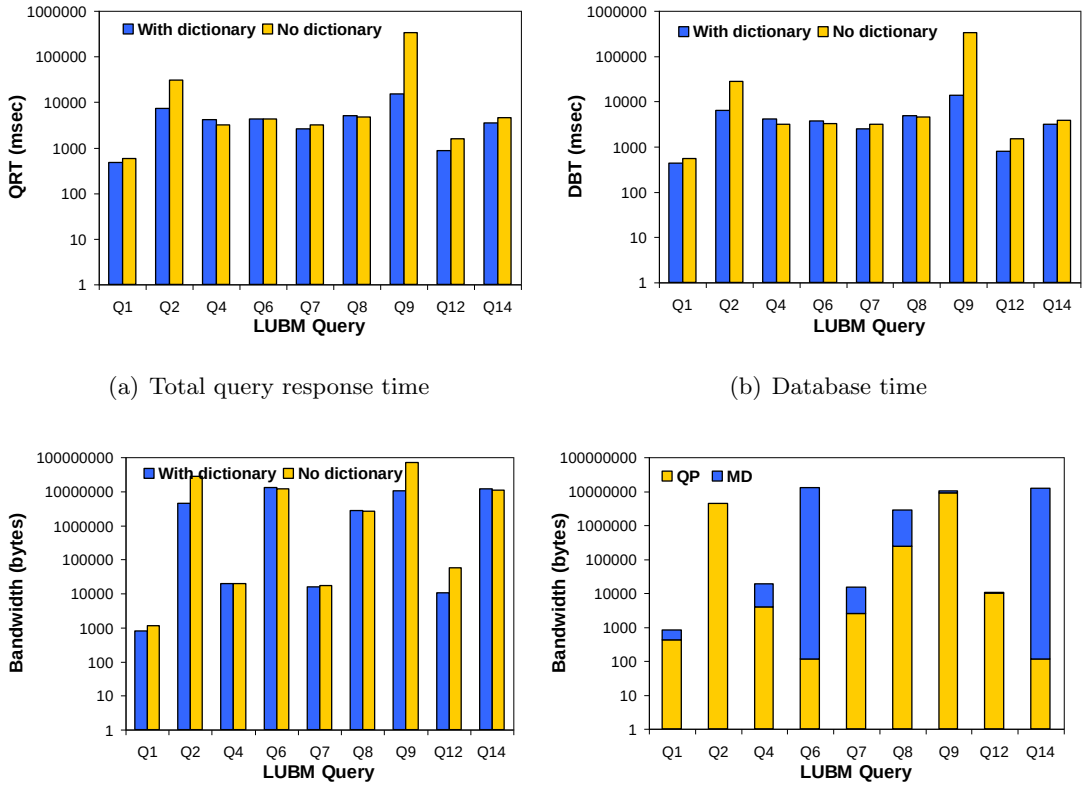


Figure 6.4: Cluster vs. PlanetLab performance

the integer values to RDF components at the end of the query evaluation.

6.4.3 Comparing PlanetLab and cluster performance

In this set of experiments we investigate the role of the two different testbeds in the system's performance. PlanetLab offers offers more real-world testing, while the more controlled environment of the local cluster enable us to experiment with bigger datasets. To examine the behaviour of these two testbeds we store the LUBM-1 dataset consisting of 144, 819 triples (given triples plus the inferred ones) in both of them. Then, we run QC* algorithm in both testbeds. Figure 6.4.3 demonstrates the difference in query response time between PlanetLab and the local cluster for various LUBM queries. All queries were executed using the mapping dictionary. The y-axis shows the query response time in msec on a logarithmic scale. As expected, the performance of the system in the cluster is substantially better than in PlanetLab. For example, the query response time of query Q2, which consists of 6 triple patterns, is 0.569 seconds in the cluster, while it is 33.9 seconds in PlanetLab. This results from the higher availability of the cluster machines as well as the bandwidth provided by a local cluster. The bad response times in PlanetLab are caused by the heavy load of PlanetLab nodes, the lower bandwidth between nodes and the bandwidth congestion. PlanetLab nodes are used simultaneously by many users which share the nodes' resources. This very often results in overloaded nodes and bandwidth congestion. In addition, PlanetLab nodes are spread across four continents and hence bandwidth is significantly lower than the bandwidth of a local cluster.



(a) Total query response time

(b) Database time

(c) Bandwidth usage

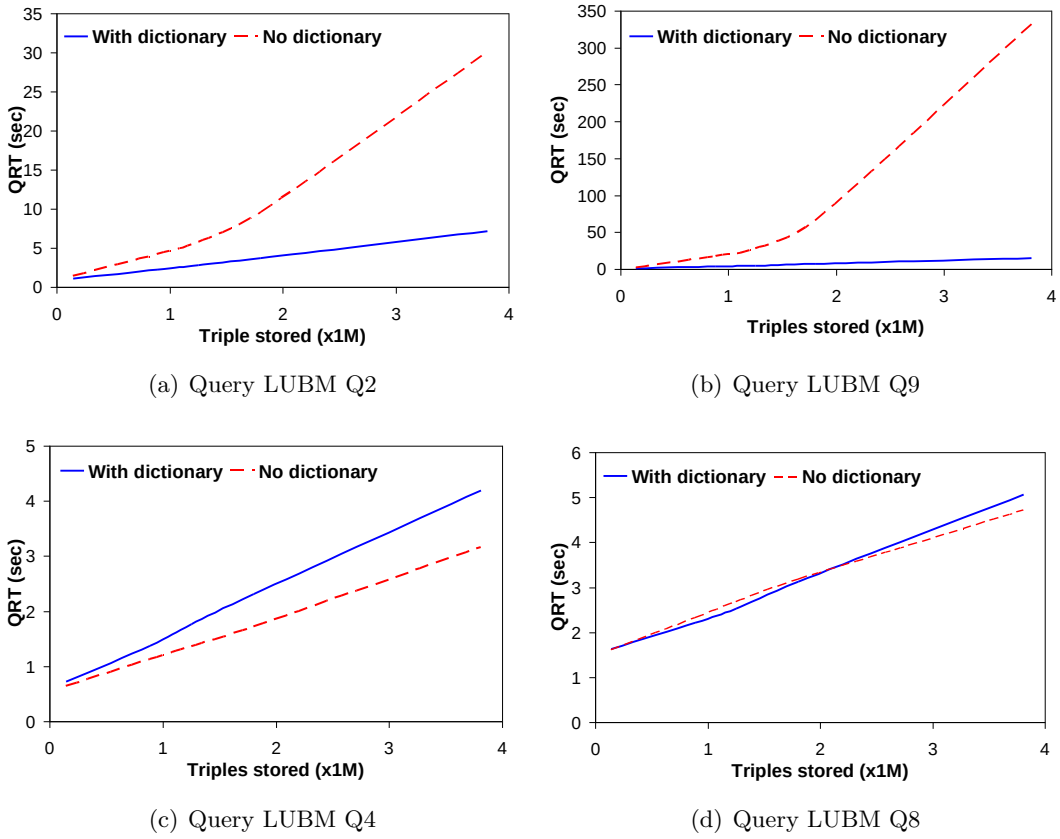
(d) Bandwidth break-down

Figure 6.5: Mapping dictionary effect in cluster for LUBM-20

6.4.4 The mapping dictionary effect in the cluster

In this section, we investigate the performance of the mapping dictionary in the cluster using bigger datasets than the ones we used in PlanetLab. Figure 6.5 shows the results of algorithm QC* with and without the mapping dictionary after having stored the LUBM-20 dataset in the network.

Figure 6.5(a) shows the query response time of the LUBM queries, while Figure 6.5(b) depicts the total time spent by the database at all peers participating in the query processing (DBT). DBT includes the time for joining the intermediate results with the locally found matching triples as well as the time for accessing the database relation for encoding and decoding the RDF components in case the dictionary is used. The x-axis in both graphs is in $msec$ on a logarithmic scale. We observe from the graph of Figure 6.5(a) that most queries perform better when the mapping dictionary is used. Queries Q2 and Q9 have a greater benefit from the use of the dictionary than the rest



(a) Query LUBM Q2

(b) Query LUBM Q9

(c) Query LUBM Q4

(d) Query LUBM Q8

Figure 6.6: Mapping dictionary as the dataset size increases

of the queries. In particular, Q2 improves from 30 sec to 7 sec and Q9 from 332 sec to 15 sec. Queries Q2 and Q9 include 6 triple patterns and require the computation of joins with large intermediate relations. Therefore, the computation of joins between relations with integer values decreases the time significantly for these two queries. This results in a database time one order of magnitude smaller when the dictionary is used, as shown in Figure 6.5(b). The query response time for the rest of the queries are quite close for both cases. Queries Q1, Q6, Q7, Q12, and Q14 perform slightly better when the mapping dictionary is used. However, this is not the case for queries Q4 and Q8. The query response time as well as the database time is slightly smaller when no mapping dictionary is used. Therefore, in these cases the mapping dictionary puts an overhead to the overall query evaluation process. However, this overhead is relatively small compared to the gain we have for more complex queries.

Figure 6.5(c) shows the total bandwidth spent for answering the queries, i.e., the messages required for the query processing as well as the bandwidth for encoding RDF

components to integer values and vice versa. Again queries Q2 and Q9, which include more triple patterns and create large intermediate relations, have a greater gain in bandwidth consumption when the dictionary is used than the other queries. The rest of the queries consume almost the same bandwidth in both cases. In Figure 6.5(d) we depict with a stacked bar the proportion of the bandwidth that is consumed from the query processing algorithm (QP) and the bandwidth consumed by the mapping dictionary functionality (MD). The query processing bandwidth includes messages which contain the intermediate relations encoded in integer values. The mapping dictionary bandwidth includes the bandwidth consumed mostly to decode the integer values to the corresponding RDF components. We observe that for queries Q2 and Q9 most of bandwidth is consumed by the query processing algorithm itself. Contrary to this, queries Q6 and Q14 which consist of one triple pattern actually spent most of the bandwidth when returning the results to the peer that posed the query.

Figure 6.6 depicts the query response time of several queries as the triples stored in the network increases. For queries Q2 and Q9 (Figures 6.6(a) and 6.6(b) respectively) we can clearly see the advantage of using a mapping dictionary. The query response time increases rapidly with the triples stored when no dictionary is used. On the contrary, with the mapping dictionary the increase in query response time is sublinear with the number of triples stored in the network. Such a feature makes the system more scalable in terms of the data stored in the network. On the other hand, queries Q4 and Q8 (Figures 6.6(c) and 6.6(d) respectively) do not indicate the same behaviour. Query Q4 performs better if no dictionary is used and this is magnified as the dataset size grows. However, in both cases the increase in query response time is linear. Query Q8 performs similarly for either case.

In order to further investigate when it is in one's interest to use a mapping dictionary, we run the same queries with re-ordering the triple patterns so that a different number of the total intermediate results produced occurs. Figure 6.7 shows how the gain from the mapping dictionary is affected by the number of intermediate results produced during the query evaluation. The x-axis shows in ascending order the total number of intermediate results produced during the query evaluation of different queries. The y-axis shows the query response time in $msec$ on a logarithmic scale. Queries suffixed with the letter a in the graph are the queries in the order given by the benchmark but without any Cartesian product. Queries suffixed with the letter b were re-ordered to achieve smaller intermediate result sets. We observe from the graph that the greatest benefit for using a mapping dictionary comes from queries that produce more than $50,000$ intermediate results during query processing. The query response

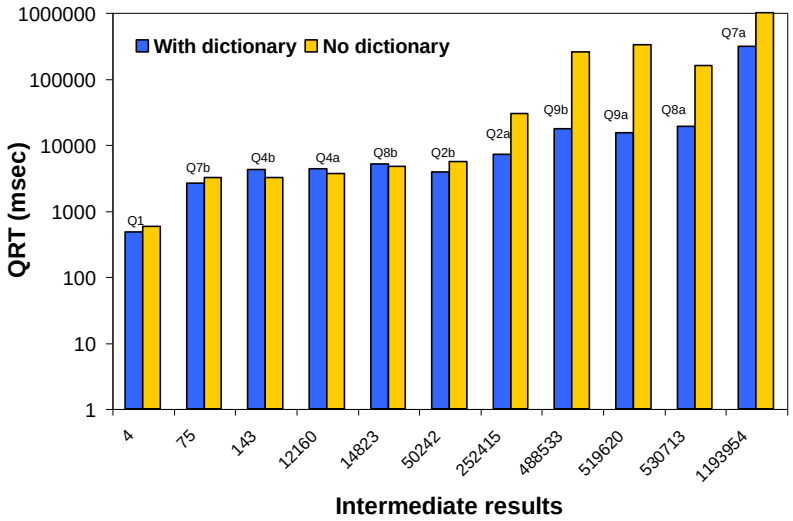


Figure 6.7: Intermediate results

time when using the dictionary can be one or more orders of magnitude smaller in these cases. For the cases where the total number of intermediate results is less than $50,000$ the mapping dictionary may cause an overhead to the query evaluation process. But still this overhead is not significant compared to the gain we can have for the more expensive queries. Certainly, such an observation depends on the testbed that the system runs.

To conclude, it is worthwhile using a mapping dictionary because the gain in time and bandwidth for complex queries is significant. In less complex queries whose intermediate relations are small it might cause an overhead but this overhead is negligible. In addition, we observed that in a DHT network where the bandwidth availability is lower than the one offered by a local cluster, such as PlaneLab, the gain from using a mapping dictionary is evident from very small datasets and for queries with smaller intermediate relations.

6.5 Summary

In this chapter we presented an algorithm for the distributed query processing of SPARQL queries of basic graph patterns. The algorithm can be used with either a forward or a backward chaining reasoning process. In addition, we described how a distributed mapping dictionary can be implemented on top of a DHT. We presented experimental results for the query processing algorithm and the distributed mapping

dictionary.

As a result of the usage the query processing algorithm of Atlas in various use cases, the order in which the triple patterns are evaluated can affect the system's performance significantly. For this reason, we study algorithms and techniques to optimize the performance of the system in respect to the time required to answer the query and the bandwidth consumed during query evaluation. The next chapter addresses the problem of SPARQL query optimization in a DHT environment.

Chapter 7

Distributed RDF Query Optimization

Query optimization has been at the center of attention in the field of databases for many years. The research area of query optimization is very large and has been studied in a great variety of contexts giving rise to several diverse solutions for each case. [66] describes some of the core issues and the proposed solutions. Although query optimization has been extensively studied in the database area and is widely used in modern DBMSs, SPARQL query optimization has been addressed only recently even in centralized environments [108, 109, 145]. This is a challenge we undertake in this chapter.

The chapter is organized as follows. In Section 7.1 we discuss the optimization issues in Atlas and state that minimizing the size of the intermediate results produced during query evaluation leads to a better system performance. Section 7.2 provides a cost model of our query processing algorithm that supports our claim. In Section 7.3, we present three greedy optimization algorithms which try to minimize the size of intermediate relations produced by the query processing algorithm utilizing selectivity-based heuristics. We describe both static and dynamic optimization algorithms. The static query optimization is completely executed by the peer that receives the initial query request and outputs a fully specified query plan. In the dynamic query optimization algorithm, optimization decisions take place at each step of the query processing algorithm and the execution plan is gradually created at each step of the query evaluation. Section 7.4 describes the methods we use to estimate the selectivity of a triple pattern and a conjunction of triple patterns and Section 7.5 discusses the statistics required to compute these selectivities. Finally, in Section 7.6 we present the experimental evalu-

ation of our optimization techniques. Results of this chapter have been published in [72].

7.1 Query Optimization in Atlas

The goal of query optimization is to find a query plan that optimizes the performance of a system with respect to a metric of interest. In our system, we are interested in improving the time required to answer a query (*query response time*) and the bandwidth consumed during query evaluation (*network bandwidth*).

The query response time of our algorithm can be improved if the time spent for query evaluation locally by each peer and the time required for network messages to reach relevant peers is improved. One way to accomplish this is by minimizing the size of the intermediate relations produced during query evaluation ($interRes'$ in QC*). In this case, we benefit in two ways: first, we achieve lower bandwidth consumption and second, we accomplish the computation of joins with smaller intermediate relations locally at peers. Lessons learned from earlier versions of Atlas persuaded us to concentrate on optimizing these metrics to improve the performance of our system.

As described in Chapter 6, the query processing algorithm QC* evaluates one triple pattern at a time transferring the intermediate results from one peer to the other resulting in left-deep execution plans. Therefore, we define a query plan for a SPARQL query q of basic graph patterns as follows.

Definition 7.1.1. *A* query plan q_g *for a query* q *with a connected query graph* $g : (N, H, E)$ *is a total order of the nodes in* N*.*

The query plan space of a query consists of all possible permutations of the triple patterns of the query. The size of the query plan space of a query with query graph $g : (N, H, E)$ is equal to $|N|!$.

7.2 Analytical Cost Model

In the following, we present a cost model for our query processing algorithm QC* which supports our intuition that minimizing the size of the intermediate relations leads to an improved system performance. We denote a conjunction of k triple patterns $tp_1 \wedge tp_2 \wedge ... \wedge tp_k$ in the specified order with a hypernode H_k in the query graph g, the number of variables appearing in H_k with v_k, and the number of tuples with arity v_k that match the conjunction of triple patterns H_k with T_{H_k}. The *network bandwidth*

consumption (NB) of a query plan q_g for evaluating these k triple patterns can be expressed as follows:

$$NB_{q_g} = \sum_{i=1}^{k} (T_{H_i} \times v_i \times S_b) \tag{7.1}$$

where H_i is the conjunction of triple patterns $tp_1 \wedge tp_2 \wedge ... \wedge tp_i$, T_{H_i} is the number of triples that match H_i, (i.e., the number of tuples of relation $interRes'$ at peer i according to algorithm QC*), v_i is the number of variables in H_i (i.e., the number of attributes of relation $interRes'$ at peer i according to algorithm QC*) and S_b is the average size in bytes of one binding value for the variables. Depending on the order in which the triple patterns are evaluated, the sum of T_{H_i} varies and hence the network bandwidth consumption changes.

The *query response time* (QRT) of a query plan q_g can be expressed in terms of the local processing time at each peer i (LPT_i) participating in the query evaluation and the time for transmitting the intermediate results (TT_i) from each peer i to peer $i+1$. Then, query response time can be computed as follows:

$$QRT_{q_g} = \sum_{i=1}^{k} (TT_{i-1} + LPT_i) \tag{7.2}$$

Let the peer that receives the query request be peer 0. We assume $TT_0 = 0$ since the transmission cost from the peer that received the query request to the first peer participating in the query evaluation is considerably smaller than the time of transmitting intermediate results. Let t_b be the time for transmitting one binding value through the network (i.e., serialization time plus transmission time plus deserialization time). We assume a constant time for transmitting a binding from one peer of the network to the other. Although this is not true for general networks, it is true for clusters of peers. In the case, where this is not true we take into consideration the worst case scenario for the communication between two peers. Then, the transmission time TT_i from peer i to peer $i+1$ is:

$$TT_i = T_{H_i} \times v_i \times t_b \tag{7.3}$$

The local processing time spent at peer i consists of the time for matching the triple pattern with the local triples and form relation lR_i and the time for joining lR_i with the intermediate results which correspond to hypernode H_{i-1}. Assume t_{sel} to be the time

for matching the triples to form relation lR, t_{join} the time for comparing two values, and T_{lR} the number of tuples of relation lR. If a simple nested loops join is performed, then the local processing time at peer i is given by the formula:

$$LPT_i = t_{sel} + (T_{H_{i-1}} \times T_{lR_i} \times t_{join}) \tag{7.4}$$

When being at the first peer (i.e., $i = 1$) where the relation of the intermediate results is empty, we have $T_{H_0} = 0$ and hence only the selection process accounts for the local processing time. Although we assumed a simple nested loops join, the cost of any join method depends on the size of the joining relations.

In the formulas 7.3 and 7.4, the most significant parameter which affects the query processing time and the network bandwidth is the sum of T_{H_i} which is the total number of intermediate results produced by the query processing algorithm. All other parameters are constants, e.g., the size of relations lR_i. Therefore, both local processing time and transmission time depend on the number of the intermediate results produced by a query plan during the query evaluation. As a result, the goal of the optimization algorithms described below is to choose a query plan that minimizes the number of intermediate results produced during query evaluation. In the experimental evaluation presented in Section 7.6, the assumption that query response time improves by minimizing the number of the intermediate results is validated. Other methods could also be applied for the optimization of our system performance such as compression techniques for the intermediate results transferred in the network and different join implementations locally at each peer. Certainly, different local database deployments at the peers would have different impact on the final query processing time. However, such techniques and exploration with different local databases are out of the scope of our work.

7.3 Query Optimization Algorithms

In the following, we present three greedy optimization algorithms which try to minimize the size of intermediate relations produced by the query processing algorithm utilizing selectivity-based heuristics. We describe both static and dynamic optimization algorithms. The two static query optimization algorithms to be presented below are completely executed by the peer that receives the initial query request and output a fully specified query plan. In the dynamic query optimization algorithm, optimization decisions take place at each step of the query processing algorithm and the execution

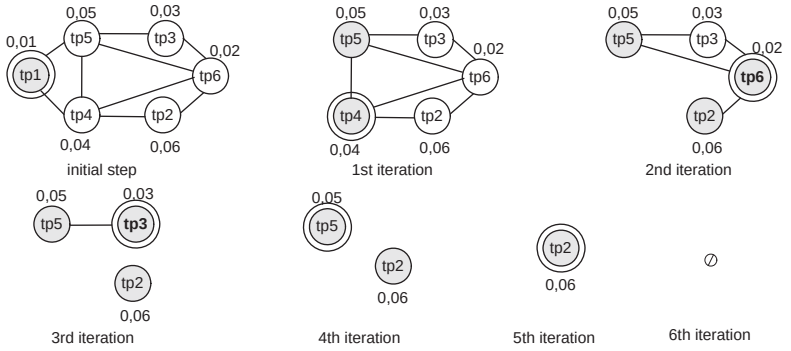


Figure 7.1: Naive query optimization example

Algorithm 6: NA: Naive optimization algorithm

Output: ordered list of triple patterns *tps*

```
1  procedure NA(query graph g)
2      assignNodesSel ();
3      n_0 := node with the minimum selectivity;
4      Mark the nodes in g that are joined with n_0;
5      tps ← g.remove(n_0);
6      while g.markedNodes() ≠ ∅ do
7          n_i := node with the minimum selectivity from g.markedNodes();
8          Mark the nodes in g that are joined with n_i;
9          tps ← g.remove(n_i);
10     end
11 end
```

plan is gradually created at each step of the query evaluation.

Our algorithms make use of the selectivity of a triple pattern as well as the selectivity of a conjunctions of triple patterns. Using standard terminology from relational database systems, the *selectivity of a triple pattern tp*, $sel(tp)$, is the fraction of the total number of triples in the network that match tp. Similarly, if H is a conjunction of triple patterns, the *selectivity of the conjunction of triple patterns*, $sel(H)$, is the fraction of total number of triples in the network that match H. In Section 7.4, we propose methods for estimating the selectivity of a triple pattern and the selectivity of a conjunction of triple patterns.

7.3.1 Naive static algorithm

The naive algorithm (NA) orders triple patterns based on their selectivity. It chooses triple patterns from the most selective to the least selective and in a fashion where a Cartesian product computation will not be required. The algorithm runs at the peer that receives the query request before the query evaluation begins.

The optimization algorithm works as follows. Using the initial query graph representation, each node is assigned with the selectivity of the corresponding triple pattern. The algorithm firstly selects the query graph node n_0 with the minimum selectivity and adds it to the query plan. Then, it marks the nodes that are connected with n_0 and removes n_0 from the graph. The algorithm then continues by iteratively choosing the node n_{min} with the minimum selectivity, selecting only from the marked ones, adds it to the query plan, marks the nodes connected with n_{min} and removes n_{min} from the graph. When two nodes of the query graph have the same selectivity, the one that precedes in the initial query is chosen. The algorithm terminates when no nodes are left in the graph.

NA is based on the assumption that after joining two very selective triple patterns, the joining result will also be selective. Certainly, this assumption is not always true, but the algorithm often performs in a satisfactory way, as we will see in the experimental section. The use of the query graph ensures that a Cartesian product computation will never occur. Algorithm 6 shows the pseudocode of the query optimization algorithm.

Figure 7.1 depicts how NA works for the example query of Figure 3.1. Each query graph node is assigned with the selectivity of the corresponding triple pattern. At each iteration, the marked nodes (i.e., the nodes that are connected with the previously selected nodes) are shown in grey circles. The selected triple pattern (i.e., the node with the minimum selectivity) is shown with a double circle.

7.3.2 Semi-naive static algorithm

Similarly with the naive algorithm, the semi-naive algorithm (SNA) runs at the peer that receives the query request before the query evaluation begins. SNA is a variation of the minimum selectivity algorithm [144] and has also been used in [145]. SNA goes beyond NA by taking into account the selectivity of *pairs* of triple patterns as well.

Besides assigning each node of the graph with the selectivity of its triple pattern, each *edge* of the graph is also assigned with the selectivity of the conjunction of the connected triple patterns. The algorithm begins by selecting the edge with the minimum selectivity, orders its nodes based on their selectivity and adds them to the query plan.

Algorithm 7: SNA: Semi-naive optimization algorithm

```
   Output: ordered list of triple patterns tps
 1 procedure SNA(query graph g)
 2     assignNodesAndEdgesSel ();
 3     e_0 := edge with the minimum selectivity;
 4     n_0 := node of e_0 with minimum selectivity;
 5     n_1 := node of e_0 with maximum selectivity;
 6     Mark the edges in g that are connected with n_0 and n_1;
 7     tps ← n_0, n_1;
 8     remove e_0 from g.markedEdges();
 9     while g.markedEdges() ≠ ∅ do
10         e_i := edge with the minimum selectivity from g.markedEdges();
11         n_i := node of e_i that is not included in tps;
12         Mark the edges in g that are connected with n_i;
13         tps ← n_i;
14         remove all edges from g.markedEdges() that have both nodes in tps;
15     end
16 end
```

Then, SNA iteratively chooses the edge that has the minimum selectivity, but also has one of its nodes in the query plan, and adds the other node to it. SNA terminates when all nodes have been added to the query plan. In case of a tie between the selectivities of two edges, the algorithm chooses the one that has the node with the smaller selectivity. The difference of the above algorithm and the original algorithm presented in [144] lies in that the semi-naive algorithm takes into account only the selectivity of the conjunction of two triple patterns without considering the intermediate results that might have occurred by joining more than two triple patterns.

Figure 7.2 shows how SNA works for the same example used in NA. Firstly, all nodes and edges of the query graph are assigned with their corresponding selectivities. At each iteration the algorithm chooses from the marked edges (shown in dashed lines) the one that has the minimum selectivity (colored in red). It appends the node of this edge to the execution plan and marks the edges which begin from this edge and have not been marked yet.

Note that both NA and SNA, which are executed by the peer that receives the query request, actually output an ordered list of the triple patterns of the query. The triple patterns can be then evaluated sequentially by the query evaluation algorithm and ensuring that a Cartesian product computation will not occur. This is crucial for the performance of a distributed system where intermediate results are transmitted

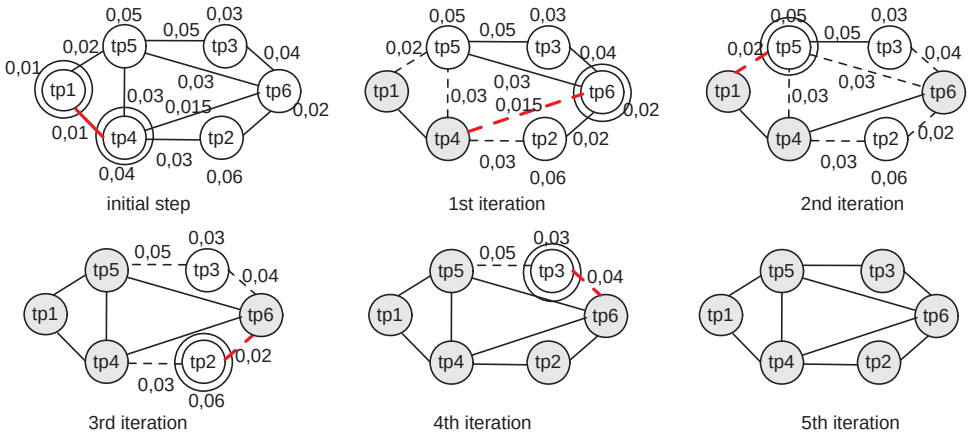


Figure 7.2: Semi-naive query optimization example

through the network.

7.3.3 Dynamic algorithm

Finally, we propose a dynamic optimization algorithm (DA) which seeks to construct query plans that minimize the number of intermediate results during query evaluation. The dynamic optimization algorithm takes into consideration the selectivity of conjunctions of triple patterns. The important difference with SNA is that the selectivity of more than two triple patterns is considered and the join selectivity is estimated between the *real intermediate results* and a triple pattern.

Initially, the peer that receives the query request, assigns all edges and nodes of the query graph with the corresponding selectivities and chooses the first two triple patterns to be evaluated as in SNA. The peer finds the edge with the minimum selectivity and sends a `QEval` message to the peer responsible for evaluating the triple pattern which has the minimum selectivity of the two nodes forming this edge. Then, the optimization step is carried out at each peer p which receives a `QEval` message. After the new query graph g' with the new hypernode H' has been created at peer p as shown in Algorithm 5, p should select the triple pattern that should be evaluated next. The candidate triple patterns are the triple patterns of the query graph nodes which are directly connected with the hypernode H'. In this way, the computation of a Cartesian product is avoided. Peer p estimates the selectivity of the join between the intermediate results so far which correspond to H' and each candidate triple pattern and assigns the corresponding edges. Then, peer p selects the node which is connected to H' with the edge with the minimum selectivity. In case a tie between the selectivities of two edges emerges, p chooses the

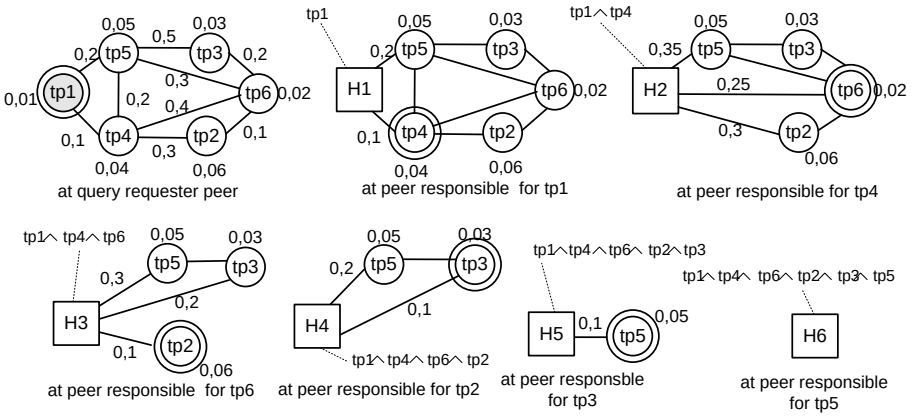


Figure 7.3: Dynamic query optimization example

node with the smaller selectivity. The algorithm then outputs the triple pattern that corresponds to this node. Local procedure markNextNode (line 21 of Algorithm 5) is responsible for conducting the query optimization process and find the triple pattern that should be evaluated next.

Figure 7.3 shows an example execution of the dynamic optimization algorithm. The query requestor peer assigns the edges and nodes of the query graph and chooses $tp1$ as the first node. At each query processing step, each peer finds the edge with the minimum selectivity from the set of edges connected to the hypernode and marks the corresponding node (shown with a double circle). In the last step of the query processing algorithm, the query graph consists only of the hypernode.

7.4 Selectivity Estimation

In this section, we propose methods for estimating the selectivity of a single triple pattern as well as the selectivity of a conjunction of triple patterns. The computation of these selectivities are required for the optimization algorithms we presented above.

7.4.1 Single triple patterns

We present two ways to estimate the selectivity of a single triple pattern; one based on a simple heuristic and one based on an analytical estimation technique using the attribute value independence assumption which is widely adopted in database systems [134].

Bound-is-easier heuristic

We consider a simple variation of the standard bound-is-easier heuristic of relational and datalog query processing [160, 161], and assume that the more bound components a triple pattern has, the more selective it is. A similar heuristic was also used in [145]. We further enrich this heuristic by considering the position of the bound components of a triple pattern, if two triple patterns have the same number of bound components. In this case, we assume that subjects are more selective than objects, which in turn are more selective than predicates. More specifically, the heuristic regarding the selectivity ordering of triple patterns is the following:
$sel(\texttt{(s, p, o)}) < sel(\texttt{(s, ?x, o)}) < sel(\texttt{(s, p, ?x)}) < sel(\texttt{(?x, p}_1\texttt{, o)}) < sel$ $(\texttt{?x, rdf:type, o}) < sel(\texttt{(s, ?x, ?y)}) < sel(\texttt{(?x, ?y, o)}) < sel(\texttt{(?x, p}_1\texttt{, ?y)} <$ $sel(\texttt{(?x, rdf:type, ?y)})$
where $\texttt{s, p, p}_1\texttt{, o}$ are constants, ?x and ?y are variables and $\texttt{p}_1 \neq \texttt{rdf:type}$. We choose to differentiate the predicate rdf:type from other predicates because rdf:type is a very popular built-in predicate used in RDF(S) databases and usually its occurrences are much more frequent than any other predicate. This ordering is only a heuristic where we consider to be accurate for the majority of the RDF(S) databases, even though there can be RDF(S) databases for which it can be proved poor. Finally, we extend this heuristic a little further by assuming that a triple pattern with a variable that will eventually be projected out is preferable from a triple pattern of the same form containing an answer variable.

Analytical estimation

We use an analytical estimation for the selectivity of a single triple pattern. Given a triple pattern $tp = (s, p, o)$, where s, p, o are variables or constants, the selectivity of tp using the attribute value independence assumption [134] is computed by the formula:

$$sel(tp) = sel(s) \times sel(p) \times sel(o) \tag{7.5}$$

where $sel(s)$, $sel(p)$, $sel(o)$ are the selectivities of the triple pattern's subject, predicate, and object, respectively. For the triple pattern components which are variables we assume a selectivity of 1.0. In addition, we assume a selectivity of 1.0 for the predicate value rdf:type, since it is is a very popular predicate in RDF(S) databases. The selectivity of the other components depends on the frequency with which their value appears in the set of triples stored in the network. We use a different definition for the frequency of a value depending on what component of the triple pattern the value is,

i.e., if it is a subject, predicate or object.

Definition 7.4.1. *The* frequency *of a triple component c with value v, denoted by* $freq_c(v)$ *where c is subject, predicate, or object, is the total number of occurrences of value v as a triple component c in the set of triples stored in the network.*

For example, $freq_s$(`ub:zoi`) is the number of occurrences of value `ub:zoi` as a subject, while $freq_o$(`ub:zoi`) denotes the number of occurrences of value `ub:zoi` as an object in the set of triples stored in the network.

Using the above definition for the frequency, the selectivity of a triple pattern component c with value v can now be computed by the formula:

$$sel_c(v) = \frac{freq_c(v)}{T} \tag{7.6}$$

where $freq_c(v)$ is the frequency of value v as a component c and T is the total number of triples. Although in [145], the attribute value independence assumption is also used, their method assumes a uniform distribution for subjects and requires a bound predicate for the objects. In Section 7.5, we describe how $freq_c(v)$ can be computed in a DHT network. For the computation of the total number of triples indexed in the network (T), we use a broadcast protocol for informing peers about the total number of triples indexed. More elegant solutions for distributed counting in P2P environments are proposed in [111], but adopting such a method is out of the scope of our work.

7.4.2 Conjunction of triple patterns

We first deal with the selectivity estimation of a conjunction of two triple patterns. Since we allow for triple patterns that have at least one bound component, we deal with triple patterns which share at least one and at most two variables. The selectivity of the conjunction of *two* triple patterns tp_1 and tp_2 can be computed by the formula:

$$\frac{joinCard(tp_1, tp_2)}{T^2} \tag{7.7}$$

where $joinCard$ is the number of tuples (cardinality) of the relation that results from joining tp_1 and tp_2 and T is the number of triples stored in the network. We already discussed how T can be computed. Therefore, we focus on estimating the cardinality of the relation resulting from the join of two triple patterns.

To compute the expression $joinCard$, we adopt a method proposed for relational systems in [160, 161] which is as follows. Let $R(A_1, ..., A_j, B_1, ...B_k)$ and $S(B_1, ..., B_k, C_1, ..., C_m)$

be two relations, and suppose that for each $i = 1, 2, ..., k$, either the domain of B_i in R is a subset of the domain of B_i in S, or vice versa. Assume that each tuple in the product of the domains of all the attributes of R is equally likely to be in R, and similarly for S. Then the expected size of $R \bowtie S$ is $\frac{T_R \times T_S}{I_{B1} \times I_{B2} \times ... \times I_{B_k}}$, where T_R and T_S is the number of tuples of relations R and S respectively, and I_{B_i} is the size of the domain B_i in R or S, whichever is a superset of the other.

Assume now that we have two triple patterns tp_1 and tp_2 and the corresponding relations R_1 and R_2 which contain all the tuples formed with values existing in the triples stored in the network that satisfy tp_1 and tp_2. Relations R_1 and R_2 have as attributes the variables of triple patterns tp_1 and tp_2, respectively. Since we deal with triple patterns that have at least one constant component, two triple patterns can share at most two variables. The cardinality of joining R_1 with R_2 if $?x_1$ is the only common variable is computed by the formula:

$$joinCard(R_1, R_2) = \frac{T_{R_1} \times T_{R_2}}{max(I_{R_1(?x_1)}, I_{R_2(?x_1)})} \tag{7.8}$$

If R_1 and R_2 share two variables, let them be $?x_1, ?x_2$, then the cardinality is computed by the formula:

$$joinCard(R_1, R_2) = \frac{T_{R_1} \times T_{R_2}}{max(I_{R_1(?x_1)}, I_{R_2(?x_1)}) \times max(I_{R_1(?x_2)}, I_{R_2(?x_2)})} \tag{7.9}$$

where T_{R_1} and T_{R_2} are the number of tuples of R_1 and R_2 respectively, $?x_1$ and $?x_2$ are the variables shared by tp_1 and tp_2, and $I_{R_i(?x_j)}$ is the size of the domain of attribute $?x_j$ of relation R_i. In other words, T_{R_i} is the number of triples that match tp_i, and $I_{R_i(?x_j)}$ is the number of distinct values that variable $?x_j$ has in the bindings of tp_i.

We can easily determine the number of triples T_R that match a triple pattern tp. If tp has one bound component c with value v, then the number of triples that match tp is equal to the number of occurrences of value v as a component c, i.e., $T_R = freq_c(v)$. If a triple pattern tp has two bound components, then we compute the number of triples that match tp using the selectivity of the triple pattern as explained earlier, i.e., $T_R = sel(tp) \times T$, where T is the total number of triples stored in the network. For the computation of the size of the domain of a variable $?x$ in a triple pattern tp, namely $I_{R(?x)}$, we distinguish two cases. If tp has one variable, (i.e., two bound components), then $I_{R(?x)}$ is equal to the number of bindings of variable $?x$. Since no duplicate triples exist in the network and $?x$ is the only variable in the triple pattern, each binding will

be unique. In this case, we have $I_{R(?x)} = T_R$. In the case where tp has two variables, (i.e., one bound component), the corresponding domain size for the shared variable can be determined by statistics created by the query optimizer of each peer. Details for the estimation of the size of the domain of a triple pattern's variable are given in Section 7.5.

7.4.3 Discussion

We now discuss the use of the above selectivity estimation techniques by the optimization algorithms of Section 7.3. NA requires only the selectivity of single triple patterns and thus both the bound-is-easier heuristic and the analytical estimation can be applied. SNA and DA require also the selectivity of conjunctions of triple patterns and hence only the analytical estimations for the single triple pattern selectivity and the conjunction of triple patterns will be used. Especially in DA, the estimation of the selectivity of the join between the intermediate results (R_1) and one triple pattern (R_2) is required. The formulas are the same as described above with the exception that relation R_1 is already formed. Therefore, the number of tuples of R_1 and the domain size of any variable in the attributes of R_1 can be computed on the fly by examining relation R_1.

Although the approaches used above for the selectivity estimation use several assumptions about the uniformity and independence of triple components, other solutions in the literature adopt an exhaustive computation of all join cardinalities. For example, the solution proposed in [145] to estimate the cardinality for the join between two triple patterns is to gather statistics by running SPARQL queries for all pairs of related predicates. However, in the absence of an ontology all combinations of distinct properties should be taken into consideration which could be very time-consuming. In addition, this method does not allow for joining triple patterns which have a variable in the predicate component. A similar approach is also adopted in [109], where the authors propose accurate selectivity estimations by precomputing exact join cardinalities for all possible choices of one or two constants in a triple pattern and materializing the results in additional indexes. However, in a distributed setting such solutions can be very expensive in terms of both time and bandwidth usage.

7.5 Statistics for RDF

In this section, we present an efficient DHT-based scheme for creating and using statistics that enable the estimation of the selectivities described in Section 7.4. These

subject	predicate	object	object-class
$freq_s$	$freq_p$	$freq_o$	$freq_c$
dp_s	ds_p	ds_o	–
do_s	dp_p	dp_o	–

Figure 7.4: Statistics kept at each peer

statistics are the frequency of a triple component and the size of the domain of a variable in a triple pattern.

7.5.1 Creating and summarizing statistics

Let us first introduce some new notation which is useful for keeping statistics for the sizes of the domain of the variables appearing in a query. We denote by $ds_c(v)$ ($dp_c(v)$ and $do_c(v)$, respectively) the total number of distinct subject values (predicate and object, respectively) that exist in the triples stored in the network which contain value v as component c. For example, let tp be the triple pattern `(?x,ub:advisor,?y)`. The size of the domain of variable `?x` of triple pattern tp is equal to the number of distinct subject values in the triples with predicate `ub:advisor`, denoted by ds_p(`ub:advisor`). The value of ds_p(`ub:advisor`) actually shows the number of students that have an advisor. Similarly, the size of the domain of variable `?y` is equal to the number of the distinct object values in the triples with predicate `ub:advisor`, denoted by do_p(`ub:advisor`). The value of do_p(`ub:advisor`) shows the number of professors that are advisors to some students. In addition, for any value v, we have $ds_s(v) = 1$, $dp_p(v) = 1$, $do_o(v) = 1$.

The following proposition allows us to to determine how statistics should be kept in a DHT environment.

Proposition 7.5.1. *Let v be the value of a bound subject of a triple pattern and p_v the peer responsible for key v. Then, peer p_v is capable of computing the exact frequency of v as a subject, ($freq_s(v)$), the exact number of the distinct predicate values with subject v ($dp_s(v)$), and the exact number of the distinct object values with subject v ($do_s(v)$) in the set of triples stored in the network by looking* only *in its local database.*

Proof. Given that peer p_v is responsible for key v, our indexing scheme forces all triples that contain v either as a subject, predicate or object to be stored at peer p_v. Therefore, peer p_v can retrieve from its local triple relation all triples that contain v as a subject and hence it can compute the occurrences of v as a subject in the set of all triples stored in the network, i.e., $freq_s(v)$. In addition, peer p_v can compute the number of

the distinct predicate values and object values for subject v by projecting the triples that contain v as subject on the predicate and object attribute respectively. □

The same result holds for a triple pattern's bound predicate or object. Following that, peers keep statistics only from their *local data* and specifically for the data values for which they are responsible (i.e., values that are the *keys* that led a triple to a specific peer). These turn out to be *global statistics* required by the optimization algorithms and can be obtained by sending low cost messages to responsible peers. This is a very good property of our indexing scheme and has not been pointed out in the literature before.

Statistics for the frequency and the number of distinct values are created for each triple component. Thus, a peer keeps the statistics shown in Figure 7.4. As shown in the figure, we differentiate between class objects, i.e., objects of triples of the form `(s, rdf:type, o)` which are RDFS classes, and resource objects, i.e., objects of triples `(s, p, o)` with `p` $\neq$ `rdf:type`. This differentiation is vital since resource objects follow a completely different distribution from RDFS classes. We also confirmed experimentally that a more accurate estimation for the statistics for the objects can be achieved in this way. In other words, in order to create the required statistics four queries are performed to the local database of each peer, one for each component.

A data structure which would keep the exact distribution of the required statistics for each triple component would require excessive memory space for very large amount of data. A commonly used method dating back from database systems is estimating the frequency distribution of an attribute by creating *histograms* [118]. Given a space budget B for each statistical structure, each peer decides if the exact distribution can be kept in memory or a summary of the distribution is required by creating a histogram.

In our system, we use v-optimal-end-biased histograms [118] to summarize the required statistics. Using terminology from [118], we use as a sort parameter the numerical values of the hashed RDF URIs and literals and as a source parameter the frequency of the values or the number of distinct values depending on the kind of the statistical structure we want to summarize. End-biased histograms require that all but one of their buckets are singleton, i.e., they contain a single element. V-optimal-end-biased histograms are serial histograms in which some of the highest frequencies and some of the lowest frequencies are placed in individual buckets, while the remaining frequencies are all grouped in a single bucket. The optimal of all possible histograms is the one that minimizes the weighted variance of the source parameter. One of the advantages of the end-biased histograms is their storage efficiency since singleton buckets occupy

less space than buckets containing multiple attribute values. The construction algorithm for this kind of histograms involves an exhaustive enumeration of all end-biased histograms in slightly over linear time [67]. Although it is confirmed from [67] that end-biased histograms are optimal for certain cases, we also found experimentally that they are much more accurate for the RDF data than the equi-width histograms used in [145].

At this point we should also note, that when the backward chaining algorithm is the chosen inference mechanism, collected statistics have not considered inferred data. In this case, the first time a query is evaluated maybe some of the statistics are not accurate and hence query optimization might not achieve the optimal goal. But when a triple pattern of a query is evaluated for the first time and new inferences are discovered, the corresponding statistics are updated. Therefore, queries that are posed later on will make use of the updated statistics. Updating the statistics does not affect the performance of the query evaluation since it is performed at a peer that has already participated in the query evaluation and has already forwarded the query to a different peer.

7.5.2 Retrieving statistics

Whenever the query optimizer of peer x needs statistics for the selectivity estimation of one triple pattern or a conjunction of triple patterns, it sends a StatsReq(v_i, c_i) message for each bound component with value v_i appearing in the triple patterns specifying also the type c_i of the component value (i.e., if it is a subject, predicate, resource object or class object). The message arrives to peer y responsible for key v_i. Peer y retrieves the required statistics for value v_i from the corresponding statistical structure (depending on the value c_i) and sends them back to peer x. The time required to retrieve the statistics is negligible compared to the time required for evaluating a query, as we will see in the experimental section. The cost of sending these messages is very low since: (a) messages are small in size, (b) messages are sent in parallel, (c) each message requires only $O(logn)$ hops to reach the destination peer, and (d) the statistics at the destination peer are kept in main memory.

As an example, consider that peer p_1 receives a query request and needs to retrieve statistics for the triple pattern `(?x, ub:advisor, ?y)`. It sends a StatsReq message to peer p_2 which is responsible for key `ub:advisor`. The query optimizer of peer p_2 receives the message, retrieves the frequency of predicate `ub:advisor` (i.e., $freq_p$(`ub:advisor`)), the number of the distinct subject values for predicate `ub:advisor` (i.e., ds_p(`ub:advisor`)), and the number of the distinct object values for predicate

`ub:advisor` (i.e., do_p(`ub:advisor`)) from its local statistics and sends them back to peer p_1. Peer p_1 sends similar messages for all bound components of the triple patterns in the query. After collecting these statistics, it computes the selectivities using the formulas we presented in Section 7.4 and runs one of the optimization algorithms described in Section 7.3 to determine the order with which the triple patterns of the query should be evaluated.

7.6 Experimental Evaluation

In this section, we present an experimental evaluation of our optimization techniques. All algorithms have been implemented as an extension to our prototype system Atlas. For our experiments, we used as a testbed both the PlanetLab network as well as the more controlled environment of the local cluster (`http://www.grid.tuc.gr/`). Although we have extensively tested our techniques on both testbeds, here we present results only from the cluster where we achieve much better performance as we showed in Section 6.4.3. The cluster consists of 41 computing nodes, each one being a server blade machine with two processors at 2.6GHz and 4GB memory. We used 30 of these machines where we run up to 4 peers per machine, i.e., 120 peers in total.

For our evaluation, we use the synthetic LUBM benchmark [48]. In each experiment we first infer all triples and then store them in the network. In these experiments we are not concerned with RDFS reasoning and hence we first infer all triples and then store them in the network. Table 5.5 in Section 5.5.5 shows the total number of inferred triples that corresponds to the datasets we used in our experiments. Some of the LUBM queries contain a significant number of triple patterns; for example, Q2 and Q9 contain 6 triple patterns. Q4 and Q8 contain 5 triple patterns, and Q7 contains 4 triple patterns. Q1, Q3, Q5, Q10, Q11, and Q13 contain only two triple patterns while Q6 and Q14 contain a single triple pattern. We mostly focus on queries with more than 4 triple patterns so that the benefits of the proposed optimizations can be clearly demonstrated. We omit query Q12 since it always produces an empty result set and does not exhibit any interesting results. The SPARQL queries can be found in Appendix A. All measurements are averaged over 10 runs using the geometric mean which is more resilient to outliers.

In addition, we use a real-world dataset which describes the US Congress vote results of the 2004 bills voting process. The dataset has a size of 3.613 MB and contains 67,392 triples. We use a set of nine queries which where used in [164] and can be found in Appendix A.

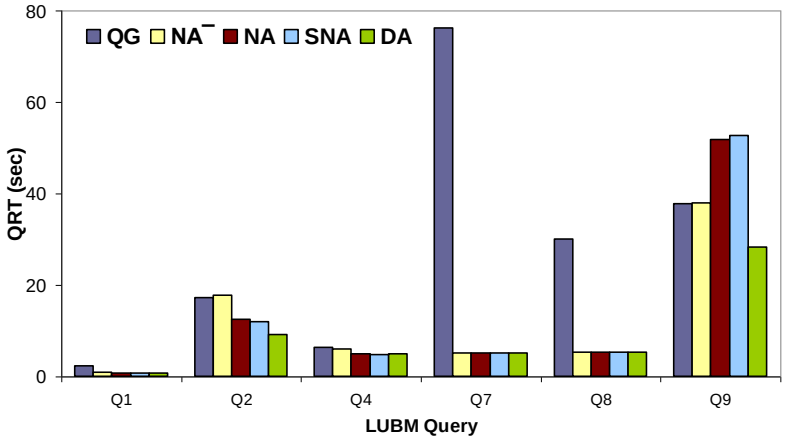


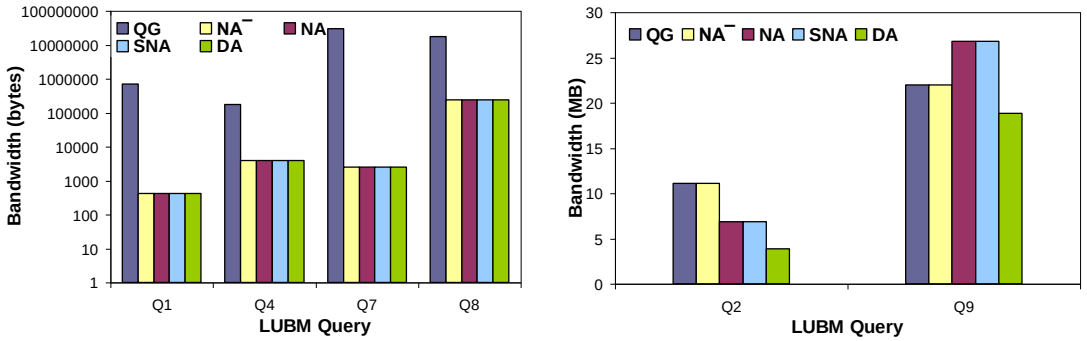
Figure 7.5: Total query response time for LUBM-50

In the following, QG denotes that the query graph is used to avoid Cartesian products but no other optimization is utilized. The naive optimization algorithm using the bound-is-easier heuristic is denoted by NA^{-}, while the naive and semi-naive optimization algorithms using the analytical estimation are denoted by NA and SNA, respectively. Finally, DA denotes the dynamic optimization algorithm. Measurements without the query graph where triple patterns were executed in the order given by the benchmark could not be conducted for some queries which were formed in a way that required the computation of Cartesian products. Query evaluation was not completed after 30 minutes and therefore such measurements are not depicted in the graphs. In all experiments below, we assume the use of the mapping dictionary as described in Section 6.3.

7.6.1 Comparing the optimization algorithms

In this section, we compare and evaluate the optimization algorithms described in Section 7.3. We achieve this by measuring the performance of the query processing algorithm QC* we presented in Section 6.1 when the different optimization techniques are used. For this set of experiments, we store all the inferred triples of the LUBM-50 dataset ($9,437,221$ triples) in a network of 120 peers. Then, using each optimization algorithm, we run the queries and measure the metrics of interest.

In all graphs of this section, the x-axis shows the LUBM queries while the y-axis depicts the metric of interest. Figure 7.5 shows the query response time (QRT) for the different LUBM queries. The query response time is the total time required to answer a query and it also includes the time required by the query optimizer for determining the



(a) Queries Q1, Q4, Q7, Q8

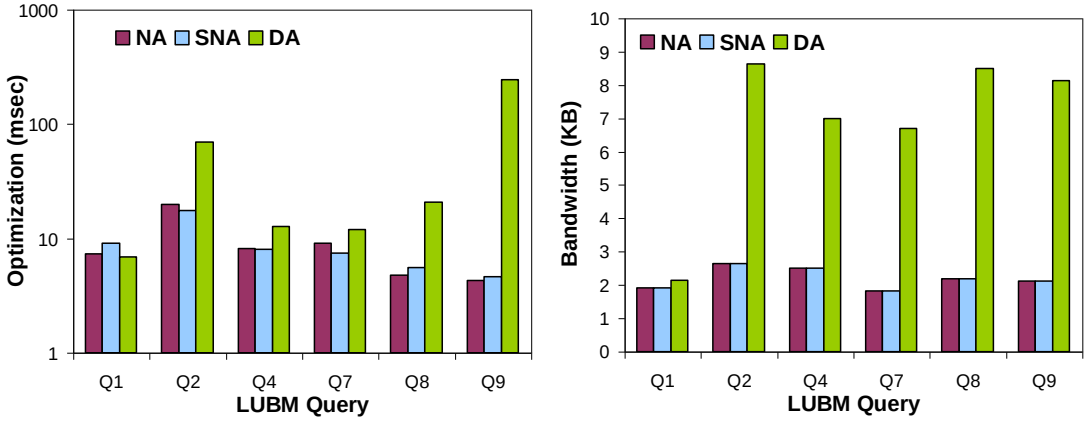
(b) Queries Q2, Q9

Figure 7.6: Bandwidth usage for LUBM-50

query plan. Figure 7.6 shows the total bandwidth consumed during query evaluation of each query including any message exchange required by the query optimizer. For readability reasons, we separated the queries in two different graphs (Figures 7.6(a) and 7.6(b)). The y-axis of Figure 7.6(a) is on logarithmic scale.

Query Q1 consists of two triple patterns which both have two bound components. The first one has a predicate `rdf:type` and is not selective, while the second one is the one that constrains the size of the result set. Therefore, *QG*, which evaluates triple patterns in the given order, performs worse than the other optimization algorithms which choose the second triple pattern to evaluate first. This can be also seen from Figure 7.6(a) where the bandwidth consumption is depicted. Queries Q4, Q7 and Q8 have also two triple patterns with two bound components and only one triple pattern does not have `rdf:type` as a predicate. Q4 is a star-shape query with all its triple patterns sharing the same subject variable, while only the first two triple patterns have two bound components. Therefore, since these two triple patterns are the more selective ones, all optimization algorithms choose the same query plan and perform identically in terms of both QRT and bandwidth. The same holds for query Q7 where QRT is significantly reduced when using either optimization technique compared to *QG*. Q8 is a query similar to Q7 and all optimization methods choose the same query plan and manage to reduce both the bandwidth consumption and the query response time.

Queries Q2 and Q9 are different from the previous queries. They consist of 6 triple patterns which have either only their predicates bound or a predicate `rdf:type`. In both queries, there exists a join among the last three triple patterns (in the order given by the benchmark) and the combination of all three triple patterns is the one that yields



(a) Optimization time

(b) Optimization bandwidth

Figure 7.7: Optimization overhead

a small result set. DA finds a query plan that combines these three triple patterns earlier than the other algorithms. This results in producing smaller intermediate result sets, as it is also shown by the bandwidth consumption in Figure 7.6(b), and thus results in better QRT. Although NA and SNA perform close to DA for query Q2, they fail to choose a good query plan for Q9 affecting both the QRT and the bandwidth consumption. At this point, we should note that QG and NA^- depend on the initial order of a query's triple patterns. For this reason, both algorithms choose a better query plan than NA and SNA for query Q9 since the order in which its triple patterns are given by the benchmark is a good one.

Figure 7.7 shows the overhead of the optimization process imposed by each algorithm. Figures 7.7(a) and 7.7(b) depict the total optimization time and the bandwidth consumption spent by the query optimizer, respectively. The optimization time contains the time for retrieving the required statistics from the network, the time for the selectivity estimation and the time spent by the optimization algorithm. For QG and NA^- the optimization overhead is only the local time spent by the peer that receives the query request and is negligible compared to the others. Thus it is not shown in the graph. The optimization time is shown in msec on a logarithmic scale. As expected, DA spends more time than the other optimization algorithms since it runs at each query processing step. However, the optimization time is still one order of magnitude smaller than the time required by the query evaluation process. Therefore, although DA requires more time than the other optimization algorithms, the system manages to perform efficiently for all queries when DA is used. We observe similar results for the

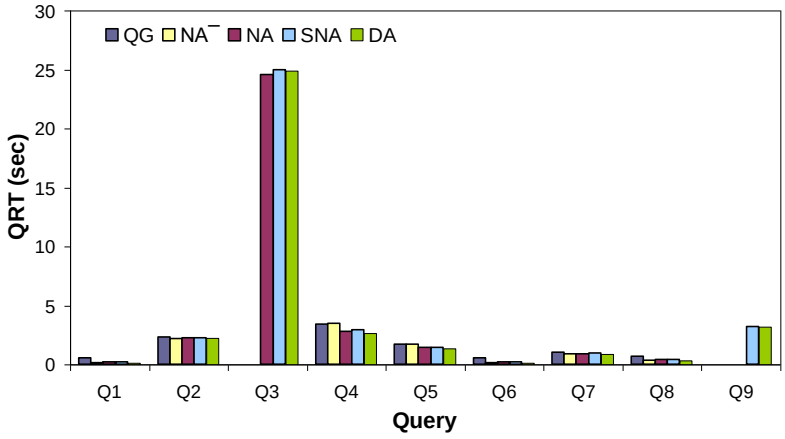


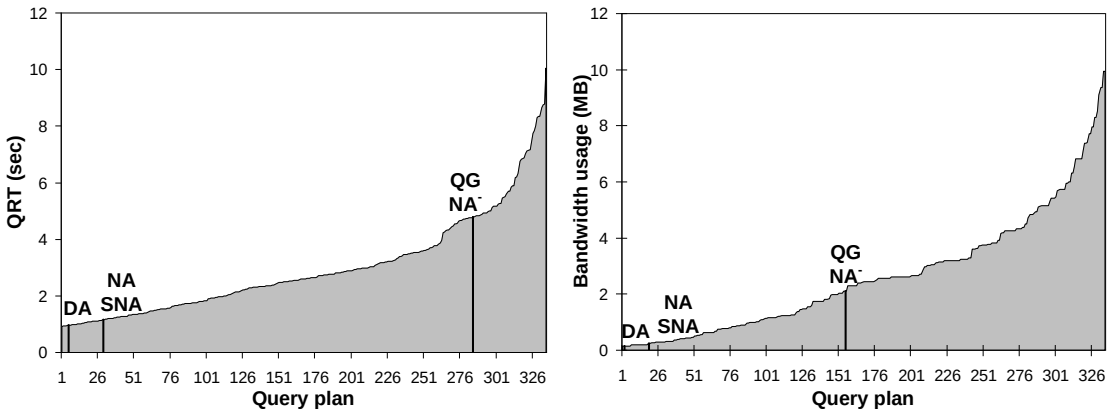
Figure 7.8: Total query response time for votes dataset

bandwidth consumed by the query optimizer. NA and SNA consume $\sim 2KB$ depending on the number of a query's bound components, while DA consumes $\sim 7KB$, which is still orders of magnitude less than the bandwidth spent during query evaluation.

Figure 7.8 shows the query response time of the queries on the real dataset of the US Congress voting process. In this figure, we observe the for most queries all optimization algorithms choose the same query plan. This is caused because all queries contain triple patterns with two bound components none of which is `rdf:type` and therefore even the simple bound-is-easier heuristic manages to find a good query plan. The only exceptions are queries Q3 and Q9, where the query processing algorithm for the optimization algorithms QG and NA^- (and NA for query Q9) could not proceed with the evaluation of the queries; the intermediate relations produced were too large causing memory exceptions at one peer.

7.6.2 Effectiveness of query optimization

In this section, we show how effective the optimization algorithms are, given the query processing algorithm. To achieve this we need to explore the performance of the query processing algorithm throughout the whole query plan space. The size of the query plan space of a query consisting of N triple patterns is $N!$. Since query plans that involve Cartesian products are very inefficient, even impossible, to evaluate in a distributed environment, we consider only triple pattern permutations which do not produce any Cartesian product. In this experiment, we store the LUBM-10 dataset in a network of 120 peers and run all possible query plans for several LUBM queries (i.e., triple pattern permutations which do not involve any Cartesian product).



(a) Query response time

(b) Bandwidth usage

Figure 7.9: Exploring the query plan space of Q2 for LUBM-10

Q2 consists of 6 triple patterns and therefore its query plan space includes 720 (i.e., 6!) query plans. The query plans of $Q2$ which do not involve any Cartesian product are 335 in total. In Figure 7.9(a), we depict the query response time of all possible query plans for query Q2 in ascending order. The x-axis shows the query plans ordered from the query plan with the minimum query response time to the query plan with the maximum query response time. In the graph, we highlight the query plans chosen by the different optimization algorithms. We observe that DA chooses one of the best query plans, while NA and SNA perform worse choosing the 27th best query plan. NA^{-} performs poorly choosing one the worst query plans. The graph which corresponds to the bandwidth consumption of the query plan space of Q2 is shown in Figure 7.9(b). Again, we observe that the the query plan chosen by the dynamic optimization algorithm is the one that consumes less bandwidth during the query evaluation process.

The graph in Figure 7.10 confirms the cost model presented in Section 7.2 and the assumption made earlier in Section 7.1 that query response time improves by minimizing the number of the intermediate results. For each query plan of the query plan space of Q2, we also measured the total number of intermediate results produced at each peer. The x-axis of the graph shows in ascending order the total number of the intermediate results produced by each query plan, while the y-axis shows time in seconds. Apart from the query response time (QRT) of each query plan, we also measured the total time spent by the database for joining the intermediate results (DBT). Figure 7.10 shows how QRT and DBT change with respect to the number of the intermediate

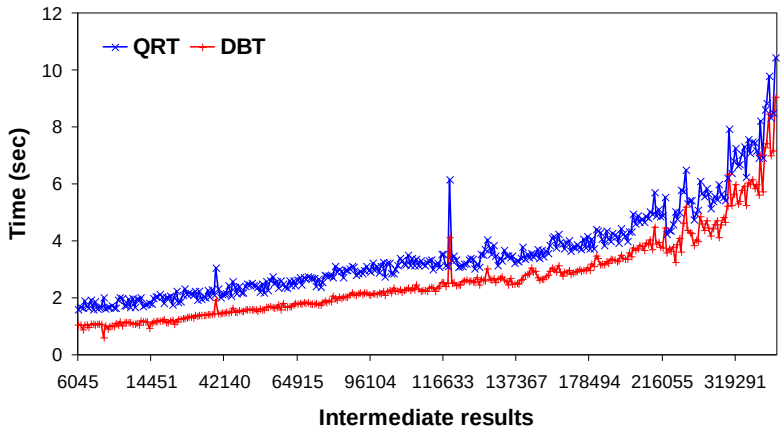


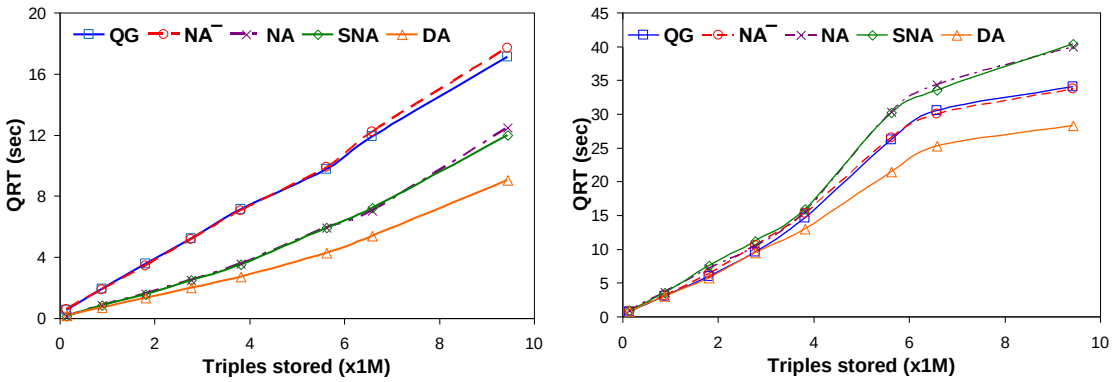
Figure 7.10: Intermediate results for query plan space of Q2 for LUBM-10

LUBM Query	Min QRT	Max QRT	DA QRT
Q2	0.91 s	10.06 s	0.96 s
Q4	2.54 s	17.39 s	2.89 s
Q7	1.53 s	16.62 s	1.55 s
Q8	2.88 s	10.20 s	3.06 s
Q9	3.30 s	64.06 s	4.41 s

Figure 7.11: QRT of LUBM query plans

results produced. Generally, we observe from the graph that as the total number of the intermediate results increases both the QRT and the DBT of the corresponding query plan deteriorates. The peaks shown in the graph are outliers caused by the nodes of the cluster. We also observe in this graph that DBT dominates the QRT in a cluster environment so it is very important to have an efficient local database at each peer. However, this was not the focus of our work and hence we have chosen to use SQLite as a simple and lightweight SQL database.

Similar results are observed for the other queries as well. In Figure 7.11, we list the query response times for all queries of the best and the worst query plan together with the query response time when the DA optimization algorithm is used. We observe that the QRT when using DA is very close to the QRT of the optimal query plan for all queries. Note that without the query plans that involve Cartesian products, the difference between the minimum and the maximum query response time of all query plans for all queries is not very large. For example, this difference is about 9 sec for Q2 and about 60 sec for Q9 which does not leave room for much optimization. Certainly,



(a) Query Q2

(b) Query Q9

Figure 7.12: Varying dataset size

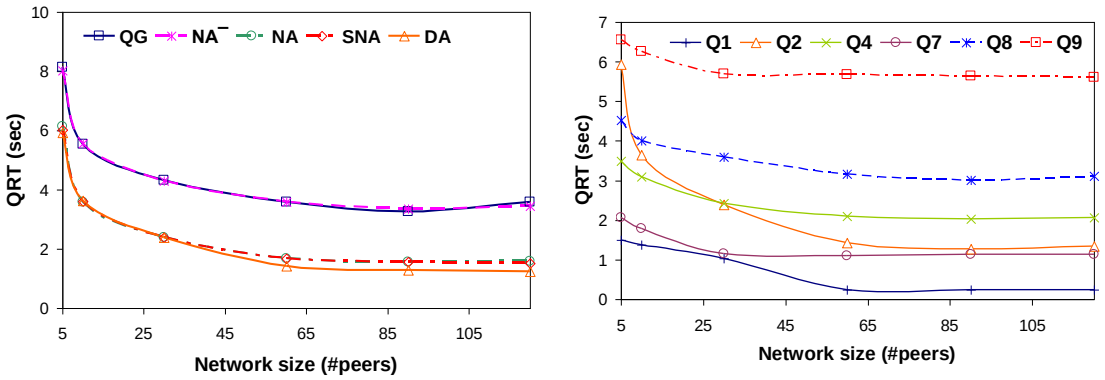
this difference increases for bigger datasets but it is still in the same order of magnitude.

7.6.3 Varying the dataset size

In this set of experiments, we study the performance of our system when varying the number of triples stored in the network. We show results for queries Q2 and Q9 which involve a join among three triple patterns. The optimization techniques for these queries perform differently providing us a better comparison study. The behaviour of our system using each optimization algorithm as the dataset stored in the network grows is depicted in Figure 7.12.

In a network of 120 peers, we store datasets from LUBM-1 to LUBM-50. Every time we measure the query response time of the queries using each optimization algorithm. Figure 7.12(a) shows the results for query Q2, while Figure 7.12(b) shows the results for query Q9. As expected, the query response time increases as the number of triples stored in the network grows in both cases. This is caused by two factors. Firstly, the local database of each peer grows and as a result local query processing becomes more time-consuming. Secondly, since queries Q2 and Q9 do not have a bound object or subject in a triple pattern (except for the ones that that have predicate `rdf:type`), the size of their result sets increases as the dataset stored grows. For example, in LUBM-1 the result set of query Q2 is empty, while in LUBM-50 the result set of Q2 contains 130 answers. Similarly, in LUBM-1 the size of result set of query Q9 is equal to 134, while in LUBM-50 it is 8627. This results in transferring larger intermediate result sets through the network which also affects the query response time of the query.

This experiment also brings forth an interesting conclusion regarding the optimiza-



(a) Varying dataset size for query Q2

(b) Varying network size using DA

Figure 7.13: Varying network size

tion techniques. We observe in Figure 7.12(a) that query plans chosen by NA, SNA and DA perform similarly up to approximately $1.8M$ triples stored (i.e., LUBM-10) and hence it makes no difference on which optimization algorithm is chosen. But for bigger datasets the query plan chosen by DA outperforms the others. This shows that the system becomes more scalable with respect to the number of triples stored in the network when the dynamic optimization algorithm is used. Similar results are observed in Figure 7.12(b) for query Q9 for datasets bigger than $3.8M$ triples (i.e., LUBM-20).

7.6.4 Varying the network size

In this set of experiments, we study the performance of our system when varying the number of peers forming the network. We bootstrap networks of 5, 10, 30, 60, 90 and 120 peers and store the LUBM-10 dataset. We then run the queries using all optimization techniques.

In Figure 7.13(a), we show the query response time for query Q2 as the network size increases. We observe that QRT improves significantly as the network size grows up to 60, while it remains almost the same for bigger network sizes. The decrease in QRT for smaller networks is caused by the fact that more peers share the storage load in the network. The more peers join the network the less triples are stored in each peer's database and thus local processing load during query evaluation is reduced. Since the dataset is relatively small, this characteristic ceases to exist for network sizes of more than 60 peers. In this query, we also observe that although NA, SNA choose different plans from DA their performance is similar for network sizes up to 30 nodes. The fact that DA performs better than the other two optimization algorithms emerges in

Statistics	Min size	Max size	Avg size
histograms (x6)	580	580	580
predicate	44	448	71.47
object-class	44	288	61.80
Total	3568	4216	3613,27

Figure 7.14: Size of statistics per peer (bytes)

networks consisting of more than 60 peers.

Figure 7.13(b) shows how the different queries perform as the number of peers in the network increases and when the dynamic optimization algorithm is used. For all queries, the query response time either improves or remains unaffected as the size of the network increases. A good property of the DHT is observed here; although the number of peers increases, query response time does not increase due to the logarithmic message routing.

7.6.5 RDF statistics

In this section, we study the efficiency and accuracy of the RDF statistics as well as the overhead for creating and maintaining them in the network.

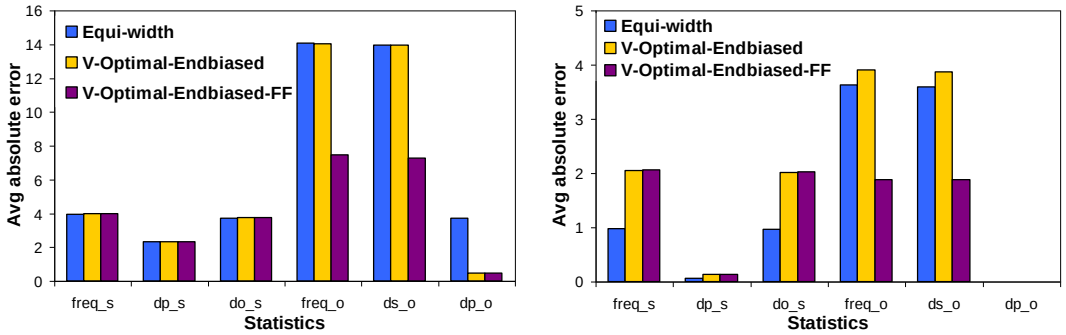
First, we present measurements concerning the size of the statistics kept by each peer. In all our experiments, we set a space budget of 500 bytes per statistical structure per peer. This resulted in keeping the exact statistics for the predicate and object-class values each peer. This is typical of a large DHT network where a peer is responsible for very few predicate or RDFS class values. All other statistics cannot fit within this space budget and are estimated by v-optimal-end-biased histograms with 10 buckets each. Each statistical structure for the subjects and objects is kept in a separate histogram resulting in a total of six histograms per peer.

The table in Figure 7.14 shows the size of the generated statistics for each peer in a network of 120 peers. Histograms always occupy the same amount of space, while the exact statistics for predicate and object-class vary depending on the number of predicate and class values the peer is responsible for. The total statistics kept at each peer result in a total amount of memory of $\sim 4K$ which is negligible compared to the data stored at each peer's local database and compared to today's powerful machines with main memory in the size of GB.

At this point, we should note that the size of the statistics does not depend on the size of the dataset stored in the network. The statistics for the predicate component

Statistics time	Min time	Max time	Avg time
database time	644	85895	29729.1
histogram construction	58	82352	19408.1
Total	702	168247	49137.2

Figure 7.15: Time required for creating statistics (msec)



(a) LUBM-1 dataset

(b) Votes dataset

Figure 7.16: Average absolute error of histograms

and the object-class component actually are affected by the RDF Schema. The RDF Schema of the dataset actually determines an upper bound for the number of predicate values and class values that will appear in the dataset. Therefore, as the dataset grows only the frequencies of these values grows and not the number of values.

Figure 7.15 demonstrates the time required for creating the RDF statistics at each peer. We present the minimum, maximum and average time from all 120 peers of the network after having stored the LUBM-50 dataset. As we have already explained in Section 7.5, creating statistics at each peer requires the retrieval of the statistics from the local database and, if necessary, the construction of the corresponding histograms. In the table, we show the time required to retrieve the statistics from the database and the total time required to construct all the corresponding histograms. The database time depends on the number of triples stored at each peer and thus, we observe a big deviation between the minimum and maximum database time. In addition, the database time is more time-consuming than the time required for the histograms construction, but still the total maximum time is in the magnitude of 2-3 minutes.

In the following, we present measurements that justify the choice of the histograms we used. Figure 7.16 depicts the average absolute error of three types of histograms

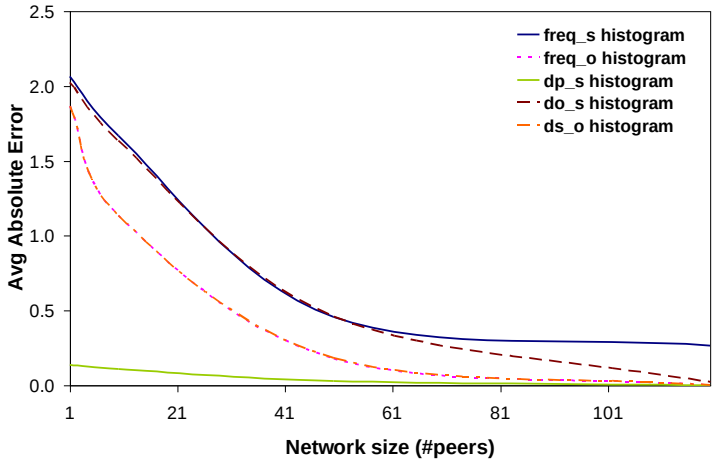


Figure 7.17: Histograms error

for two different datasets, the LUBM-1 dataset (Figure 7.16(a)) and the votes dataset (Figure 7.16(b)). The x-axis shows each type of statistics that needs to be summarized with a histogram. For every value v_i appearing in the dataset as a component c, we have measured the real statistic value (either frequency or distinct values) $stat_c(v_i)$ and the estimated statistic value $\widehat{stat}_c(v_i)$ taken from the corresponding histogram. The absolute error of the statistic $stat$ for value v_i appearing as a component c is equal to:

$$e^{abs}(v_i) = |stat_c(v_i) - \widehat{stat}_c(v_i)| \tag{7.10}$$

where $stat$ is one of $\{freq, ds, dp, do\}$. If N is the domain size of component c in the dataset, the average absolute error is computed as:

$$\frac{1}{N}\sum_{i=1}^{N} e^{abs}(v_i) \tag{7.11}$$

The three histograms depicted in the figure are the equal width histogram (*Equi-width*), the v-optimal-end-biased histogram using as a sort parameter the values of the dataset (*V-Optimal-Endbiased*) and the v-optimal-end-biased histogram using as a sort parameter the source parameter, i.e., either the frequency or the distinct values depending on the statistical structure the histogram summarizes (*V-Optimal-Endbiased-FF*). The graphs of Figure 7.16 indicate that for the subject statistics it is preferable to have an equi-width histogram. In the the votes dataset the use of an equi-width histogram results in a smaller error while in the LUBM-1 dataset the error is almost equal for

all histograms. For collecting statistics for the objects of the triples the v-optimal-end-biased-ff histogram summarizes better these statistics and achieves a significantly smaller average absolute error for both datasets.

In the next set of experiments, we investigate how the size of the network affects the summaries of the histograms. We compute the average absolute error as defined in equations (7.10) and (7.11) for each histogram for different network sizes ranging from 1 to 120 peers for LUBM-5. A network consisting of a single peer resembles a centralized system where a global histogram is created from all data stored. Results for each statistical structure that is estimated by a histogram are shown in Figure 7.17. We observe that as the number of peers increases the error drops significantly. This shows that the more peers join the network (i.e., more histograms created), the better the estimation becomes. As the size of the network increases, the combination of the local estimated distributions resembles the real one. This is caused by the fact that the overall size of the statistics in the network increases and therefore, the estimated distributions become more precise.

7.6.6 Discussion

We have also experimented with different datasets using the SP^{2}B benchmark [132] as well as a real world dataset of the US Congress vote results presented in [165]. The results were similar to the ones observed using LUBM. For all datasets, *DA* consistently chooses a query plan close to the optimal regardless of the query type or dataset stored and without posing a significant overhead neither to the total time for answering the query nor to the bandwidth consumed. On the contrary, the static optimization methods are dependent on the type of the query and the dataset, which make them unsuitable in various cases (as shown earlier for LUBM query Q9). In addition, we have also tested indexing all possible combinations of the triples' components, as proposed in [89]. In this case, we have used histograms at each peer for combinations of triples' components as well. However, we did not observe any difference in the choice of the query plan and thus, showed results only with the triple indexing algorithm. This results from the nature of the LUBM queries which mostly involve bound predicates and object-classes for which we kept an exact distribution in both cases. Hence the type of histograms used did not affect the choice of the query plans.

7.7 Summary

In this chapter we presented the query optimization techniques we have developed in Atlas. We started by analyzing the cost of the query processing algorithm and proposed optimization algorithms that try to minimize the size of the intermediate results produced during the query evaluation. In this way, we achieved the optimization of our system's performance in terms of query response time and bandwidth consumption. We proposed techniques for the selectivity estimation of SPARQL queries and the statistics required in a DHT environment to enable the selectivity estimation techniques. We demonstrated that it is sufficient for a node to create and maintain local statistics, i.e., statistics for its locally stored data. These local statistics are in fact global statistics needed by the optimization algorithms and can be obtained by other nodes by sending low cost messages. This is a very good property of the indexing scheme we adopt from [32] that has not been pointed out in the literature before.

This chapter concludes the results of this thesis. In the next chapter, we highlight the main achievements of our work and discuss open problems and possible directions for future work.

Chapter 8

Conclusions

In this thesis we designed and built a system for distributed RDF query processing and reasoning in P2P networks. This chapter presents a short summary of our work and describes our main contributions in the research field. We conclude this thesis with a discussion on open problems and possible directions for future work.

8.1 Summary

We presented a system that is able to support full-fledged management of RDF data in a large-scale decentralized environment. In particular, we fully designed and implemented a P2P system, called Atlas, for the distributed query processing and reasoning of RDF and RDFS data. In the architecture of Atlas, providers expose resource descriptions to the network by inserting RDF(S) information to the system. Consumers want to discover resources by submitting SPARQL queries to the system. Any node in the network can accept a request from a provider or a consumer. All nodes in the system are equal and there is neither centralized control nor global knowledge.

Apart from DHTs, other distributed and parallel computing platforms have been proposed for the RDF query processing and reasoning more recently. These include distributed computing platforms based on powerful clusters and cloud computing platforms using MapReduce. However, these approaches demand high-end, locally deployed infrastructures whose cost can be very high in many cases. Our work focused on a DHT-based architecture and proposed algorithms which can be run on commodity machines deployed all over the world, as it is the case with many other P2P applications. Our system is based on the DHT paradigm and uses the triple indexing scheme originally presented in [33] where each RDF triple is indexed in the DHT three times, based on

its subject, predicate, object. RDF data and RDFS schemas are handled uniformly in our system, i.e., both RDF and RDFS triples are indexed using the same scheme.

One of the main focus of our work was to enable Atlas to support RDFS reasoning. Two well-known reasoning techniques we studied are *forward chaining* and *backward chaining*. A forward chaining approach has minimal requirements during query answering, but needs a significant amount of storage for all the inferred data. In contrast, the backward chaining approach has minimal storage requirements, at the cost of an increase in query response time. There is a time-space trade-off between these two approaches which has been studied in centralized environments before. Our work was the first that investigated this trade-off in detail in a *distributed Internet-scale* scenario.

We designed and implemented both forward and backward chaining algorithms for RDFS reasoning in Atlas, as well as an algorithm which works in a bottom-up fashion using the magic sets transformation technique. Our forward chaining algorithm was the first one that dealt with an important case of generation of redundant RDF information, while our backward chaining algorithm was the first distributed backward chaining algorithm proposed for RDFS reasoning in a decentralized environment in general. Moreover, a magic sets transformation technique for distributed RDFS reasoning has not been studied in the literature before. We studied theoretically the correctness of our reasoning algorithms and proved that they are sound and complete. We also provide a comparative study of our algorithms both analytically and experimentally. In the experimental part of our study, we obtained measurements in the realistic large-scale distributed environment of PlanetLab as well as in the more controlled environment of a local cluster.

Furthermore, we presented a complete query evaluation process which is used to answer arbitrary SPARQL queries of basic graph patterns in Atlas. The algorithm uses a query graph for representing queries to avoid Cartesian products and is able to incorporate RDFS reasoning with either a forward or a backward chaining manner. We have also enhanced the query processing algorithm with a distributed mapping dictionary. As URIs and literals may consist of long strings that are transferred in the network and processed locally at the nodes, we can benefit from a mapping dictionary in reducing the local processing cost and the transmission cost. Although mapping dictionaries are by now standard in centralized RDF stores, our work was the first that discussed how to implement one in a DHT environment.

Finally, we addressed the problem of query optimization over RDF data stored on top of DHTs and fully implemented and evaluated a DHT-based optimizer. Although query optimization has been extensively studied in the database area and is widely used

in modern DBMSs, RDF query optimization has been addressed only recently even in centralized environments. Our optimization algorithms ranged from static to dynamic ones and focused on minimizing the size of the intermediate results computed during query evaluation. In this way, we achieved to decrease the time required for answering a query and the bandwidth consumed during the query evaluation. Our algorithms utilized selectivity estimates to determine a query plan that minimizes the size of the intermediate results. We also proposed methods for estimating the selectivity of RDF queries utilizing techniques from relational databases. We defined which statistics are required at each node of the network for the computation of the selectivity estimates and used histograms for summarizing the distribution of these statistics. We demonstrated that it is sufficient for a node to create and maintain local statistics, i.e., statistics for its locally stored data. These local statistics are in fact global statistics needed by the optimization algorithms and can be obtained by other nodes by sending low cost messages. This is a very good property of the indexing scheme we adopted from [32] that has not been pointed out in the literature before.

8.2 Future Directions

In this section we present a discussion on open problems related with our work and give possible directions for future work. These directions are only suggestions and are not exhaustively discussed here.

8.2.1 Load balancing

DHTs can suffer from load imbalances [151]. Load imbalances can appear at the level of the key distribution, i.e., keys are not evenly distributed among the nodes of the network, and at the level of the item distribution, i.e., items are not evenly shared among the nodes. The first phenomenon can be easily solved with a consistent hash function and by using virtual nodes [78, 147]. The second problem occurs mainly due to data skewness. RDF data are highly skewed since various values, and mostly properties such as `rdf:type`, `rdfs:label`, appear very frequently in a dataset. Therefore, most DHT systems which use the triple indexing scheme of [33] suffer from load imbalances concerning storage load. Such a storage load imbalance can limit the scalability of the system [85].

Load balancing methods can be distinguished to data replication and data relocation. Applying data replication to a problem of data skewness does not usually comprise a solution, rather it magnifies the problem by overloading more nodes than necessary.

A relocation method might be more suitable for the storage load balancing in such a setting. In [14] the authors propose a method based on relocation and utilize overlay trees among nodes to keep track of the relocated triples. However, many questions regarding the details and the behavior of this algorithm remain open.

Orthogonally to the storage load, there is also the problem of overloaded nodes due to the query processing algorithm (query load). Such a problem can occur in our system if a specific triple pattern is very popular and is used in most queries, leading to overloading the same node of the network. This could be solved by replicating triples to different nodes and share the query requests among these nodes. The question that arises now is which is the right balance between RDF data replication and relocation in a scenario where both storage and query load requires balancing.

8.2.2 Distributed recursive queries using Datalog

Datalog has been revived recently and is used in many modern distributed applications. In [90, 91], a variation of Datalog is used for expressing routing protocols in a simple and compact way based on the observation that recursive query languages are very suitable for expressing routing protocols. The implementation of the routing protocols in [90, 91] can then be done by evaluating recursive queries in a distributed manner. In [4] a rule-based language based on Datalog is proposed for the distributed management of Web data. The architecture and scenario where this language can be used is introduced in [5]. One of the main challenges the our reasoning algorithms cope with is recursive query processing of RDF(S) data. Our proposed algorithms can be easily adapted and extended to fit in this research field by enhancing our system to support arbitrary recursive rules.

8.2.3 RDF in the cloud

There has been an increasing interest in cloud computing techniques for many diverse domains and applications lately. Cloud computing infrastructures promise to offer scalability and reliability features more than P2P or database systems can deliver today. Our work can form the basis for developing algorithms and techniques for RDF query processing and reasoning in such environments. There are some recent proposals in the Semantic Web literature which utilize the cloud computing paradigm for the management of RDF and OWL data. As it has been proven, approaches that are based on distributed computing platforms consisting of powerful clusters and cloud computing platforms using MapReduce, such as [162, 163, 166], can be very scalable for computing

the closure of RDF(S) graphs. However, these approaches do not deal with the query answering problem, which is equally important. Approaches that do deal with SPARQL query answering such as [98, 104, 128] can scale to billions of triples, but exhibit poor performance with respect to the time required to answer the query (i.e., query answering is in the order of hours). Thus, it is still an open question how to achieve scalability and efficiency for SPARQL query answering as well as RDFS reasoning using top-down approaches in such environments.

References

[1] Karl Aberer. P-Grid: A Self-Organizing Access Structure for P2P Information Systems. In *Proceedings of 9th International Conference on Cooperative Information Systems (CoopIS)*, pages 179–194, 2001.

[2] Karl Aberer, Philippe Cudre-Mauroux, Anwitaman Datta, Zoran Despotovic, Manfred Hauswirth, Magdalena Punceva, and Roman Schmidt. P-Grid: A Self-Organizing Structured P2P System. *SIGMOD Record*, 32:29–33, September 2003.

[3] Karl Aberer, Philippe Cudre-Mauroux, Manfred Hauswirth, and Tim Van Pelt. GridVine: Building Internet-Scale Semantic Overlay Networks. In *Proceedings of the 13th World Wide Web Conference (WWW 2004)*, New York, USA, 2004.

[4] Serge Abiteboul, Meghyn Bienvenu, Alban Galland, and Emilien Antoine. A Rule-based Language for Web Data Management. In *Proceedings of the 30th PODS Conference*, Athens, Greece, 2011.

[5] Serge Abiteboul, Alban Galland, and Neoklis Polyzotis. Web Information Management with Access Control. In *Proceedings of the 14th International Workshop on the Web and Databases (WebDB 2011)*, Athens, Greece, 2011.

[6] Serge Abiteboul, Richard Hull, and Victor Vianu. *Foundations of Databases*. Addison-Wesley, 1995.

[7] Sofia Alexaki, Vassilis Christophides, Greg Karvounarakis, and Dimitris Plexousakis. On Storing Voluminous RDF Descriptions: The case of Web Portal Catalogs. In *Proceedings of the 4rth International Workshop on the Web and Databases (WebDB 2001, co-located with SIGMOD 2001)*, Santa Barbara, California, USA, 2001.

[8] Luc Onana Alima, Ali Ghodsi, Sameh El-Ansary, Seif Haridi, and Per Brand. Multicast in DKS(N, k, f) Overlay Networks. In *Proceedings of the 3rd International Conference on Peer-to-Peer Computing (P2P'2003)*, pages 196–197, 2002.

[9] Luc Onana Alima, Ali Ghodsi, and Seif Haridi. A Framework for Structured Peer-to-Peer Overlay Networks. In *Proceedings of the International Workshop on Global Computing 2004 (GC 2004)*, pages 223–250, Rovereto, Italy, 2004.

[10] Tom Heath an Christian Bizer. *Linked Data: Evolving the Web into a Global Data Space*. Morgan & Claypool, 2011.

[11] Marcelo Arenas, Claudio Gutierrez, Jorge Perez, and Catolica Chile. An Extension of SPARQL for RDFS. In *Proceedings of Semantic Web, Ontologies and Databases, VLDB Workshop (SWDB-ODBIS 2007)*, Vienna, Austria, 2007.

[12] Hari Balakrishnan, M. Frans Kaashoek, David R. Karger, Robert Morris, and Ion Stoica. Looking up Data in P2P Systems. *Communications of the ACM*, 46(2):43–48, 2003.

[13] Francois Bancilhon, David Maier, Yehoshua Sagiv, and Jeffrey D Ullman. Magic sets and other strange ways to implement logic programs (extended abstract). In *Proceedings of the fifth ACM SIGACT-SIGMOD symposium on Principles of database systems (PODS '86)*, pages 1–15, New York, NY, USA, 1986. ACM.

[14] Dominic Battre, Felix Heine, Andre Hoing, and Odej Kao. Load-balancing in P2P based RDF stores. In *Proceedings of the 2nd International Workshop on Scalable Semantic Web Knowledge Base Systems (SSWS 2006, co-located with ISWC 2006)*, Athens, Georgia, USA, 2006.

[15] Dominic Battre, Felix Heine, Andre Hoing, and Odej Kao. BabelPeers: P2P based Semantic Grid Resource Discovery. *High Performance Computing and Grids in Action*, 16:288–307, 2008.

[16] Dominic Battre, Andre Hoing, Felix Heine, and Odej Kao. On Triple Dissemination, Forward-Chaining, and Load Balancing in DHT based RDF stores. In *Proceedings of the 4th International Workshop on Databases, Information Systems and Peer-to-Peer Computing (DBISP2P 2006, co-located with VLDB 2006)*, Seoul, Korea, September 2006.

[17] Catriel Beeri and Raghu Ramakrishnan. On the Power of Magic. In *PODS '87: Proceedings of the sixth ACM SIGACT-SIGMOD-SIGART symposium on Principles of database systems*, pages 269–284, New York, NY, USA, 1987. ACM.

[18] BerkeleyDB. `http://sleepycat.com/`.

[19] Tim Berners-Lee, Roy Fielding, and Larry Masinter. RFC 2396 - Uniform Resource Identifiers (URI): Generic Syntax. http://www.isi.edu/in-notes/rfc2396.txt, Augoust 1998.

[20] Tim Berners-Lee and Mark Fischetti. *Weaving the Web : The Original Design and Ultimate Destiny of the World Wide Web by its Inventor*. Harper San Francisco, 1999.

[21] Tim Berners-Lee, James Hendler, and Ora Lassila. The Semantic Web. *Scientific American*, 284(5):34–43, 2001.

[22] Philip A. Bernstein, Fausto Giunchiglia, Anastasios Kementsietsidis, John Mylopoulos, Luciano Serafini, and Ilya Zaihrayeu. Data Management for Peer-to-Peer Computing : A Vision. In *Workshop on the Web and Databases (WebDB 2002)*, pages 89–94, 2002.

[23] Kenneth P. Birman. *Reliable Distributed Systems: Technologies, Web Services, and Applications*. Springer Verlag, 2005.

[24] Chris Bizer and Andreas Schultz. Benchmarking the Performance of Storage Systems that Expose SPARQL Endpoints. In *4th International Workshop on Scalable Semantic Web Knowledge Base Systems (SSWS2008)*, Karlsruhe, Germany, 2008.

[25] Christian Bizer, Tom Heath, and Tim Berners-Lee. Linked Data - The Story So Far. *International Journal on Semantic Web and Information Systems (IJSWIS)*, 2009.

[26] Christian Bizer, Jens Lehmann, Georgi Kobilarov, Sören Auer, Christian Becker, Richard Cyganiak, and Sebastian Hellmann. DBpedia - A Crystallization Point for the Web of Data. *Journal of Web Semantics: Science, Services and Agents on the World Wide Web*, 7:154–165, September 2009.

[27] Dan Brickley and Ramanathan V. Guha. Resource Description Framework (RDF) Schema Specification 1.0. Technical report, W3C Recommendation, 2000.

[28] Dan Brickley and Ramanathan V. Guha. RDF Vocabulary Description Language 1.0: RDF Schema. Technical report, W3C Recommendation, 2004.

[29] Jeen Broekstra and Arjohn Kampman. SeRQL: An RDF Query and Transformation Language. http://www.cs.vu.nl/ jbroeks/papers/SeRQL.pdf, 2004.

[30] Jeen Broekstra, Arjohn Kampman, and Frank van Harmelen. Sesame: A Generic Architecture for Storing and Querying RDF and RDF Schema. In *Proceedings of the 1st International Semantic Web Conference (ISWC 2002)*, Sardinia, Italy, 2002.

[31] Min Cai, Martin Frank, and Pedro Szekely. MAAN: A Multi-Attribute Addressable Network for Grid Information Services. In *Proceedings of the 4th International Workshop on Grid Computing (Grid2003)*, Phoenix, Arizona, USA, 2003.

[32] Min Cai, Martin R. Frank, Baoshi Yan, and Robert M. MacGregor. A Subscribable Peer-to-Peer RDF Repository for Distributed Metadata Management. *Journal of Web Semantics: Science, Services and Agents on the World Wide Web*, 2(2):109–130, December 2004.

[33] Min Cai and Marting Frank. RDFPeers: A Scalable Distributed RDF Repository based on A Structured Peer-to-Peer Network. In *Proceedings of the 13th World Wide Web Conference (WWW 2004)*, New York, USA, 2004.

[34] E. I. Chong, S. Das, G. Eadon, and J. Srinivasan. An Efficient SQL-based RDF Querying Scheme. In *Proceedings of the 31st Very Large Data Bases Conference (VLDB 2005)*, Trondheim, Norway, 2005.

[35] Eugene Inseok Chong, Souripriya Das, George Eadon, and Jagannathan Srinivasan. An Efficient SQL-based RDF Querying Scheme. In *Proceedings of the 31st international conference on Very large data bases (VLDB 2005)*, Trondheim, Norway, 2005.

[36] Philippe Cudré-Mauroux, Adriana Budura, Manfred Hauswirth, and Karl Aberer. PicShark: Mitigating Metadata Scarcity Through Large-Scale P2P Collaboration. *The VLDB Journal*, 17(6):1371–1384, 2008.

[37] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: Simplified Data Processing on Large Clusters. In *Proceedings of the USENIX Symposium on Operating Systems Design & Implementation (OSDI)*, pages 137–147, 2004.

[38] Songyun Duan, Anastasios Kementsietsidis, Kavitha Srinivas, and Octavian Udrea. Apples and Oranges: A Comparison of RDF Benchmarks and Real RDF Datasets. In *Proceedings of the 2011 international conference on Management of data (SIGMOD'11)*, pages 145–156, Athens, Greece, 2011.

[39] Marc Ehrig, Peter Haase, F. van Harmelen, Ronny Siebes, Steffen Staab, Heiner Stuckenschmidt, Rudi Studer, and Christoph Tempich. The SWAP Data and Metadata Model for Semantics-based Peer-to-Peer Systems. In *Proceedings of MATES-2003. First German Conference on Multiagent Technologies*, pages 144–155, Erfurt, Germany, 2003.

[40] Orri Erling and Ivan Mikhailov. Towards Web Scale RDF. In *Proceeindgs of the 4th International Workshop on Scalable Semantic Web knowledge Base Systems (SSWS2008)*, Karlsruhe, Germany, October 2008.

[41] Orri Erling and Ivan Mikhailov. RDF support in the virtuoso DBMS. In *Networked Knowledge - Networked Media*, pages 7–24, 2009.

[42] Qiming Fang, Ying Zhao, Guangwen Yang, and Weimin Zheng. Scalable Distributed Ontology Reasoning Using DHT-Based Partitioning. In *ASWC '08: Proceedings of the 3rd Asian Semantic Web Conference on The Semantic Web*, pages 91–105, Berlin, Heidelberg, 2008. Springer-Verlag.

[43] Fausto Giunchiglia and Ilya Zaihrayeu. Making Peer Databases Interact - A Vision for an Architecture Supporting Data Coordination. In *Proceedings of the Conference on Information Agents (CIA 2002)*, Madrid, Spain, September 2002.

[44] Alasdair J. G. Gray, Raul Garcia-Castro, Kostis Kyzirakos, Manos Karpathiotakis, Jean-Paul Calbimonte, Kevin Page, Jason Sadler, Alex Frazer, Ixent Galpin, Alvaro A. A. Fernandes, Norman W. Paton, Oscar Corcho, Manolis Koubarakis, David De Roure, Kirk Martinez, and Asuncion Gomez-Perez. A Semantically Enabled Service Architecture for Mashups over Streaming and Stored Data. In *Proceedings of the 8th Extended Semantic Web Conference (ESWC 2011)*, Heraklion, Greece, 2011.

[45] Steven Gribble, Alon Halevy, Zachary Ives, Maya Rodrig, and Dan Suciu. What Can Databases do for Peer-to-Peer? In *Proceedings of the 4th Workshop on Databases and the Web (WebDB 2001)*, Santa Barbara, California, 2001.

[46] Krishna Gummadi, Ramakrishna Gummadi, Steve Gribble, Sylvia Ratnasamy, Scott Shenker, and Ion Stoica. The Impact of DHT Routing Geometry on Resilience and Proximity. In *SIGCOMM '03: Proceedings of the 2003 conference on Applications, technologies, architectures, and protocols for computer communications*, pages 381–394, New York, NY, USA, 2003. ACM Press.

[47] Krishna P. Gummadi, Richard J. Dunn, Stefan Saroiu, Steven D. Gribble, Henry M. Levy, and John Zahorjan. Measurement, modeling, and analysis of a peer-to-peer file-sharing workload. In *Proceedings of the nineteenth ACM symposium on Operating systems principles*, pages 314–329. ACM Press, 2003.

[48] Yuanbo Guo, Zhengxiang Pan, and Jeff Heflin. LUBM: A Benchmark for OWL Knowledge Base Systems. *Journal of Web Semantics*, 3(2-3):158–182, 2005.

[49] Peter Haase, Jeen Broekstra, Marc Ehrig, Maarten Menken, Peter Mika, Michal Plechawski, Pawel Pyszlak, Björn Schnizler, Ronny Siebes, Steffen Staab, and Christoph Tempich. Bibster - a semantics-based bibliographic peer-to-peer system. In Sheila A. McIlraith, Dimitris Plexousakis, and Frank van Harmelen, editors, *Proceedings of the Third International Semantic Web Conference, Hiroshima, Japan, 2004*, volume 3298 of *LNCS*, pages 122–136. Springer, November 2004.

[50] Alon Y. Halevy, Zachary G. Ives, Jayant Madhavan, Peter Mork, Dan Suciu, and Igor Tatarinov. The Piazza Peer Data Management System. *IEEE Transactions on Knowledge and Data Engineering*, 16(07):787–798, July 2004.

[51] Matthew Harren, Joseph M. Hellerstein, Ryan Huebsch, Boon Thau Loo, Scott Shenker, and Ion Stoica. Complex Queries in DHT-based Peer-to-Peer Networks. In *Proceedings of the 1st International Workshop on Peer-to-Peer Systems (IPTPS'02)*, Cambridge, MA, USA, March 7-8 2002.

[52] Steve Harris and Nicholas Gibbins. 3Store: Efficient Bulk RDF Storage. In *Proceedings of the 1st International Workshop on Practical and Scalable Semantic Systems(PSSS 2003)*, Sanibel Island, Florida, USA, 2003.

[53] Steve Harris, Nick Lamb, and Nigel Shadbolt. 4store: The Design and Implementation of a Clustered RDF Store. In *5th International Workshop on Scalable Semantic Web Knowledge Base Systems (SSWS2009)*, Washington DC, USA, 2009.

[54] Andreas Harth, Katja Hose, Marcel Karnstedt, Axel Polleres, Kai-Uwe Sattler, and Jürgen Umbrich. Data Summaries for On-Demand Queries over Linked Data. In *Proceedings of the 19th international conference on World wide web*, WWW '10, pages 411–420, Raleigh, North Carolina, USA, 2010.

[55] Andreas Harth, Jurgen Umbrich, Aidan Hogan, and Stefan Decker. YARS2: A Federated Repository for Querying Graph Structured Data from the Web. In *Proceedings of the 6th International Semantic Web Conference (ISWC/ASWC 2007)*, pages 211–224, Busan, South Korea, 2007.

[56] P. Hayes. RDF Semantics. W3C Recommendation, February 2004. `http://www.w3.org/TR/rdf-mt/`.

[57] Felix Heine. Scalable P2P based RDF Querying. In *Proceedings of the 1st International Conference on Scalable Information Systems (Infoscale 2006)*, Hong Kong, 2006.

[58] Felix Heine, Matthias Hovestadt, and Odej Kao. Processing Complex RDF Queries over P2P Networks. In *Proceedings of Workshop on Information Retrieval in Peer-to-Peer-Networks (P2PIR 2005)*, Bremen, Germany, 2005.

[59] Ian Hickson. HTML Microdata. W3C Working Draft, 24 May 2011.

[60] Pascal Hitzler, Markus Krftzsch, Bijan Parsia, Peter F. Patel-Schneider, and Sebastian Rudolph. OWL 2 Web Ontology Language Primer. W3C Recommendation, `http://www.w3.org/TR/owl2-primer/`, 2009.

[61] Ryan Huebsch, Brent Chun, , and Joseph Hellerstein. PIER on PlanetLab: Initial Experience and Open Problems, 2003. Technical Report.

[62] Ryan Huebsch, Brent Chun, Joseph M. Hellerstein, Boon Thau Loo, Petros Maniatis, Timothy Roscoe, Scott Shenker, Ion Stoica, and Aydan R. Yumerefendi. The Architecture of PIER: an Internet-Scale Query Processor. In *Proceedings of the Biennial Conference on Innovative Data Systems Research (CIDR 2005)*, Asilomar, CA, USA, January 2005.

[63] Ryan Huebsch, Joseph M. Hellerstein, Nick Lanham, Boon Thau Loo, Scott Shenker, and Ion Stoica. Querying the Internet with PIER. In *Proceedings of the 29th International Conference on Very Large Data Bases (VLDB 2003)*, Berlin, Germany, September 9-12 2003.

[64] Giovambattista Ianni, Thomas Krennwallner, Alessandra Martello, and Axel Polleres. A Rule System for Querying Persistent RDFS Data. In *Proceedings of the 6th European Semantic Web Conference (ESWC 2009)*, Heraklion, Greece, 2009. (Demo paper).

[65] Giovambattista Ianni, Thomas Krennwallner, Alessandra Martello, and Axel Polleres. Dynamic Querying of Mass-Storage RDF Data with Rule-Based Entailment Regimes. In *Proceedings of the 8th International Semantic Web Conference (ISWC 2009)*, Washington DC, USA, October 2009.

[66] Yannis Ioannidis. *Query Optimization*, chapter 45, pages 1038–1057. Handbook of Computer Science, 1996.

[67] Yannis Ioannidis and Viswanath Poosala. Balancing Histogram Optimality and Practicality for Query Result Size Estimation. In *Proceedings of the ACM International Conference on Management of Data (SIGMOD 1995)*, pages 233–244, San Jose, California, United States.

[68] H. V. Jagadish, Rakesh Agrawal, and Linda Ness. A Study of Transitive Closure as a Recursion Mechanism. *SIGMOD Record*, 16(3):331–344, 1987.

[69] Zoi Kaoudi and Manolis Koubarakis. Distributed RDFS Reasoning over Structured Overlay Networks. Submitted to a journal, 2011.

[70] Zoi Kaoudi, Manolis Koubarakis, Kostis Kyzirakos, Matoula Magiridou, Iris Miliaraki, and Antonis Papadakis-Pesaresi. Publishing, Discovering and Updating Semantic Grid Resources using DHTs. In *CoreGRID Workshop on Grid Programming Model, Grid and P2P Systems Architecture, Grid Systems, Tools and Environments*, Heraklion, Crete, Greece, June 2007.

[71] Zoi Kaoudi, Manolis Koubarakis, Kostis Kyzirakos, Iris Miliaraki, Matoula Magiridou, and Antonios Papadakis-Pesaresi. Atlas: Storing, Updating and Querying RDF(S) Data on Top of DHTs. *Journal of Web Semantics*, 2010.

[72] Zoi Kaoudi, Kostis Kyzirakos, and Manolis Koubarakis. SPARQL Query Optimization on Top of DHTs. In *Proceedings of the 9th International Conference on The Semantic Web (ISWC 2010)*, Shanghai, China, 2010.

[73] Zoi Kaoudi, Kostis Kyzirakos, Manolis Koubarakis, George Leoleis, Nikolaos Skarmeas, Katia Sycara, Lydia Stamelou, Christos Tryfonopoulos, George Tzoumas, and Roman Vaculin. Semantic Web service discovery using DHTs: theoretical analysis, algorithms and experimental evaluation. Technical report, 2007.

[74] Zoi Kaoudi, Iris Miliaraki, and Manolis Koubarakis. RDFS Reasoning and Query Answering on Top of DHTs. In *Proceedings of the 7th International Conference on The Semantic Web (ISWC 2008)*, Karlsruhe, Germany, 2008.

[75] Zoi Kaoudi, Iris Miliaraki, Matoula Magiridou, Erietta Liarou, Stratos Idreos, and Manolis Koubarakis. Semantic Grid Resource Discovery in Atlas. *Knowledge and Data Management in Grids*, 2006. Talia Domenico and Bilas Angelos and Dikaiakos Marios D. (editors), Springer.

[76] Zoi Kaoudi, Iris Miliaraki, Matoula Magiridou, Antonis Papadakis-Pesaresi, and Manolis Koubarakis. Storing and Querying RDF Data in Atlas. 3rd European Semantic Web Conference, 11 - 14 June 2006, Budva, Montenegro, Demo paper, 2006.

[77] David R. Karger, Eric Lehman, Tom Leighton, Matthew Levine, Daniel Lewin, and Rina Panigrahy. Consistent Hashing and Random Trees: Distributed Caching Protocols for Relieving Hot Spots on the World Wide Web. In *Proceedings of the Twenty-Ninth Annual ACM Symposium on Theory of Computing*, pages 654–663, El Paso, Texas, May 1997.

[78] David R. Karger and Matthias Ruhl. Simple Efficient Load Balancing Algorithms for Peer-to-Peer systems. In *Proceedings of the 16th ACM Symposium on Parallelism in Algorithms and Architectures (SPAA 2004)*, Barcelona, Spain, 2004.

[79] Marcel Karnstedt. Query Processing in a DHT-Based Universal Storage - The World as a Peer-to-Peer Database. PhD thesis, 2009.

[80] Marcel Karnstedt, Kai-Uwe Sattler, Martin Richtarsky, Jessica Muller, Manfred Hauswirth, Roman Schmidt, and Renault John. UniStore: Querying a DHT-based Universal Storage. In *Proceedings of the 23rd International Conference on Data Engineering, ICDE 2007 (Demo paper)*, Istanbul, Turkey, April 2007.

[81] Greg Karvounarakis, Sofia Alexaki, Vassilis Christophides, Dimitris Plexousakis, and Michel Scholl. RQL: A Declarative Query Language for RDF. In *Proceedings of the 11th World Wide Web Conference (WWW 2002)*, Honolulu, Hawaii, U.S.A., 2002.

[82] Georgi Kobilarov, Tom Scott, Yves Raimond, Silver Oliver, Chris Sizemore, Michael Smethurst, Christian Bizer, and Robert Lee. Media Meets Semantic Web - How the BBC Uses DBpedia and Linked Data to Make Connections. In *Proceedings of the 6th European Semantic Web Conference (ESWC)*, Heraklion, Crete, Greece, 2009.

[83] George Kokkinidis and Vassilis Christophides. Semantic Query Routing and Processing in P2P Database Systems: The ICS-FORTH SQPeer Middleware. In *EDBT Workshops*, Heraklion, Crete, Greece, March 2004.

[84] George Kokkinidis, Lefteris Sidirourgos, and Vassilis Christophides. *Semantic Web and Peer-to-Peer*, chapter Query Processing in RDF/S-based P2P Database Systems. Springer-Verlag, 2006.

[85] Spyros Kotoulas, Eyal Oren, and Frank van Harmelen. Mind the Data Skew: Distributed Inferencing by Speeddating in Elastic Regions. In *Proceedings of the 19th International World Wide Web Conference (WWW 2010)*, Raleigh NC, USA, 2010.

[86] Manolis Koubarakis and Kostis Kyzirakos. Modeling and Querying Metadata in the Semantic Sensor Web: the Model stRDF and the Query Language stSPARQL. In *Proceedings of the 7th Extended Semantic Web Conference (ESWC 2010)*, Heraklion, Greece, 2010.

[87] Manolis Koubarakis, Iris Miliaraki, Zoi Kaoudi, Matoula Magiridou, and Antonis Papadakis-Pesaresi. Semantic Grid Resource Discovery using DHTs in Atlas. In *3rd GGF Semantic Grid Workshop*, Athens, Greece, 13-16 February 2006.

[88] Ora Lassila and Ralph R. Swick. Resource Description Framework (RDF) Model and Syntax Specification. Technical report, W3C Recommendation, 1999.

[89] Erietta Liarou, Stratos Idreos, and Manolis Koubarakis. Evaluating Conjunctive Triple Pattern Queries over Large Structured Overlay Networks. In *Proceedings of 5th the International Semantic Web Conference (ISWC 2006)*, Athens, GA, USA, November 2006.

[90] Boon Thau Loo, Tyson Condie, Minos Garofalakis, David E. Gay, Joseph M. Hellerstein, Petros Maniatis, Raghu Ramakrishnan, Timothy Roscoe, and Ion Stoica. Declarative Networking: Language, Execution and Optimization. In *Proceedings of the 2006 ACM SIGMOD international conference on Management of data (SIGMOD '06)*, pages 97–108, New York, NY, USA, 2006. ACM.

[91] Boon Thau Loo, Joseph M. Hellerstein, Ion Stoica, and Raghu Ramakrishnan. Declarative Routing: Extensible Routing with Declarative Queries. In *Proceedings of the 2005 conference on Applications, technologies, architectures, and protocols for computer communications (SIGCOMM '05)*, pages 289–300, New York, NY, USA, 2005. ACM.

[92] Eng Keong Lua, J. Crowcroft, M. Pias, R. Sharma, and S. Lim. A Survey and Comparison of Peer-to-Peer Overlay Network Schemes. *Communications Surveys Tutorials, IEEE*, 7(2):72 – 93, quarter 2005.

[93] Angela Maduko, Kemafor Anyanwu, Amit Sheth, and Paul Schliekelman. Graph Summaries for Subgraph Frequency Estimation. In *Proceedings of the 5th European Semantic Web Conference (ESWC'08)*, pages 508–523, Tenerife, Canary Islands, Spain, 2008.

[94] Aimilia Magkanaraki, Val Tannen, Vassilis Christophides, and Dimitris Plexousakis. Viewing the Semantic Web Through RVL Lenses. In *Proceedings of the Second International Semantic Web Conference (ISWC 2003)*, Florida, USA, 2003.

[95] Dahlia Malkhi, Moni Naor, and David Ratajczak. Viceroy: a scalable and dynamic emulation of the butterfly. In *Proceedings of the Twenty-First Annual ACM Symposium on Principles of Distributed Computing (PODC)*, pages 183–192, 2002.

[96] Mehdi Mani, Anh-Minh Nguyen, and Noel Crespi. What's up 2.0: P2P Spontaneous Social Networking. In *Proceedings of the 28th Conference on Computer Communications (IEEE INFOCOM)*, 2009.

[97] Petar Maymounkov and David Mazieres. Kademlia: A Peer-to-Peer Information System Based on the XOR Metric. In *Proceedings of the International Workshop on Peer-to-Peer Systems (IPTPS)*, pages pages 53–65, Cambridge, MA, USA, March 2002.

[98] Peter Mika and Giovanni Tummarello. Web Semantics in the Clouds. *IEEE Intelligent Systems*, 23(5):82–87, 2008.

[99] Dejan S. Milojisic, Vana Kalogeraki, Rajan Lukose, Kiran Nagaraja, Jim Pruyne, Bruno Richard, Sami Rollins, and Zhichen Xu. Peer-to-Peer Computing. Technical Report HP-2002-57, HP Laboratories, Palo Alto, 2001.

[100] Guido Moerkotte and Thomas Neumann. Analysis of two existing and one new dynamic programming algorithm for the generation of optimal bushy join trees without cross products. In *Proceedings of the 32nd international conference on Very large data bases*, VLDB '06, pages 930–941. VLDB Endowment, 2006.

[101] Sergio Muñoz, Jorge Pérez, and Claudio Gutierrez. Simple and Efficient Minimal RDFS. *Web Semantics: Science, Services and Agents on the World Wide Web*, 7(3):220–234, September 2009.

[102] Sergio Munoz, Jorge Perez, and Claudio Gutierrez. Minimal deductive systems for rdf. In *Proceedings of the 4th European Semantic Web Conferene (ESWC 2007)*, pages 53–67, 2007.

[103] Raghava Mutharaju, Frederick Maier, and Pascal Hitzler. A MapReduce Algorithm for EL+. In *Proceedings of the 23rd International Workshop on Description Logics (DL2010)*, Waterloo, Canada, 2010.

[104] Jaeseok Myung, Jongheum Yeon, and Sang-goo Lee. SPARQL Basic Graph Pattern Processing with Iterative MapReduce. In *Proceedings of the 2010 Workshop on Massive Data Analytics on the Cloud*, MDAC '10, pages 6:1–6:6, Raleigh, North Carolina, 2010.

[105] Wolfgang Nejdl, Boris Wolf, Changtao Qu, Stefan Decker, Michael Sintek, Ambjörn Naeve, Mikael Nilsson, Matthias Palmér, and Tore Risch. EDUTELLA: A P2P Networking Infrastructure based on RDF. In *Proceedings of the 11th World Wide World Conference (WWW 2002)*, WWW '02, pages 604–615, Honolulu, Hawaii, USA, 2002.

[106] Wolfgang Nejdl, Boris Wolf, Steffen Staab, and Julien Tane. In *Semantic Web Workshop 2002*, volume 55 of *CEUR Workshop Proceedings*, 2002.

[107] Wolfgang Nejdl, Martin Wolpers, Wolf Siberski, Christoph Schmitz, Mario Schlosser, Ingo Brunkhorst, and Alexander Loser. Super-Peer-Based Routing and Clustering Strategies for RDF-Based Peer-To-Peer Networks. In *Proceedings of the 12th WWW Conference*, Budapest, Hungary, May 2003.

[108] Thomas Neumann and Gerhard Weikum. RDF-3X: a RISC-style engine for RDF. In *Proceedings of 34th International Conference on Very Large Data Bases (VLDB 2008)*, volume 1, pages 647–659, Auckland, New Zealand, 2008.

[109] Thomas Neumann and Gerhard Weikum. Scalable Join Processing on Very Large RDF Graphs. In *Proceedings of the 35th SIGMOD international conference on Management of data*, pages 627–640, Providence, Rhode Island, USA, 2009.

[110] Wee Siong Ng, Beng Chin Ooi, Kian-Lee Tan, and Aoying Zhou. PeerDB: A P2P-based System for Distributed Data Sharing. In *Proceedings of the 19th International Conference on Data Engineering*, Bangalore, India, March, 2003.

[111] Nikos Ntarmos, Peter Triantafillou, and Gerhard Weikum. Distributed Hash Sketches: Scalable, Efficient, and Accurate Cardinality Estimation for Distributed Multisets. *ACM Transactions on Computer Systems (ACM TOCS)*, 27(1):1–53, 2009.

[112] Eyal Oren, Spyros Kotoulas, George Anadiotis, Ronny Siebes, Annette ten Teije, and Frank van Harmelen. MaRVIN: A platform for large-scale analysis of Semantic Web data. In *Proceedings of Web Science Conference*, 2009.

[113] Eyal Oren, Spyros Kotoulas, George Anadiotis, Ronny Siebes, Annette ten Teije, and Frank van Harmelen. MaRVIN: Distributed Reasoning over Large-scale Semantic Web Data. *Journal of Web Semantics: Science, Services and Agents on the World Wide Web*, 7(4):305–316, 2009.

[114] Alisdair Owens, Andy Seaborne, Nick Gibbins, and mc schraefel. Clustered TDB: A Clustered Triple Store for Jena. Technical Report (Unpublished), 2008.

[115] Jorge Perez, Marcelo Arenas, and Claudio Gutierrez. nSPARQL: A Navigational Language for RDF. In *Proceedings of the 7th International Conference on The Semantic Web (ISWC 2008)*, Karlsruhe, Germany, 2008.

[116] Jorge Perez, Marcelo Arenas, and Claudio Gutierrez. Semantics and Complexity of SPARQL. *ACM Transactions on Database Systems*, 34:16:1–16:45, 2009.

[117] Theoni Pitoura and Peter Triantafillou. Load Distribution Fairness in P2P Data Management Systems. In *Proceedings of the 21st Interntational Conference on Data Engineering (ICDE 2007)*, Tokyo, Japan, 2007.

[118] Viswanath Poosala, Yannis Ioannidis, Peter Haas, and Eugene Shekita. Improved Histograms for Selectivity Estimation of Range Predicates. In *ACM SIGMOD 1996*, 1996.

[119] Eric Prud'hommeaux and Andy Seaborn. SPARQL Query Language for RDF. W3C Recommendation `http://www.w3.org/TR/rdf-sparql-query/`, 2008.

[120] Bastian Quilitz and Ulf Leser. Querying Distributed RDF Data Sources with SPARQL. In *Proceedings of the 5th Extended Semantic Web Conference (ESWC 2008)*, Tenerife, Spain, 2008.

[121] Sylvia Ratnasamy, Paul Francis, Mark Handley, Richard Karp, and Scott Shenker. A Scalable Content-addressable Network. In *Proceedings of the ACM SIGCOMM '01 Conference*, San Diego, California, August 2001.

[122] Sean Rhea, Dennis Geels, Timothy Roscoe, and John Kubiatowicz. Handling Churn in a DHT. In *USENIX Annual Technical Conference*, 2004.

[123] Sean Rhea, Byung gon Chun, John Kubiatowicz, and Scott Shenker. Fixing the Embarrassing Slowness of OpenDHT on PlanetLab. In *Proceedings of the 2nd conference on Real, Large Distributed Systems (WORLDS'05)*, pages 25–30, San Francisco, CA, 2005.

[124] Patricia Rodríguez-Gianolli, Anastasios Kementsietsidis, Maddalena Garzetti, Iluju Kiringa, Lei Jiang, Mehedi Masud, Renée J. Miller, and John Mylopoulos. Data sharing in the Hyperion peer database system. In *Proceedings of the 31st international conference on Very large data bases*, VLDB '05, pages 1291–1294, Trondheim, Norway, 2005.

[125] Antony Rowstron and Peter Druschel. Pastry: Scalable, Distributed Object Location and Routing for Large-Scale- Peer-to-Peer Storage Utility. In *Proceedings of the 3rd IFIP/ACM International Conference on Distributed Systems Platforms (Middleware 2001)*, Heidelberg, Germany, 2001.

[126] Manuel Salvadores, Gianluca Correndo, Tope Omitola, Nick Gibbins, Steve Harris, and Nigel Shadbolt. 4s-reasoner: RDFS Backward Chained Reasoning Support in 4store. In *Web-scale Knowledge Representation, Retrieval, and Reasoning (Web-KR3)*, Toronto, Canada, 2010.

[127] Stefan Saroiu, Krishna Gummadi, and Steven Gribble. A Measurement Study of Peer-to-Peer File Sharing Systems. In *Proceedings of Multimedia Conferencing and Networking*, San Jose, CA, January 2002.

[128] Alexander Schatzle, Martin Przyjaciel-Zablocki, and Georg Lausen. PigSPARQL: Mapping SPARQL to Pig Latin. In *Proceedings of the International Workshop on Semantic Web Information Management*, SWIM '11, pages 4:1–4:8, Athens, Greece, 2011.

[129] Mario Schlosser, Michael Sintek, Stefan Decker, and Wolfgang Neijdl. A Scalable and Ontology-Based P2P Infrastructure for Semantic Web Services. In *2nd IEEE International Conference on Peer-to-Peer Computing*, 2002.

[130] Mario Schlosser, Michael Sintek, Stefan Decker, and Wolfgang Nejdl. HyperCuP - Hypercubes, Ontologies and Efficient Search on Peer-to-peer Networks. In *Agents and Peer-to-Peer Computing*, volume 2530. Springer, May 2003.

[131] Michael Schmidt, Thomas Hornung, Norbert Küchlin, Georg Lausen, and Christoph Pinkel. An Experimental Comparison of RDF Data Management Approaches in a SPARQL Benchmark Scenario. In *Proceedings of the 7th International Semantic Web Conference (ISWC 2008)*, pages 82–97, Karlsruhe, Germany, 2008.

[132] Michael Schmidt, Thomas Hornung, Georg Lausen, and Christoph Pinkel. SP^{2}Bench: A SPARQL Performance Benchmark. In *Proceedings of the 29th International Conference on Data Engineering (ICDE2009)*, Shanghai, China, 2009.

[133] Andy Seaborne. RDQL - A query Language for RDF, W3C Member Submission 9 January 2004. http://www.w3.org/Submission/2004/SUBM-RDQL-20040109/, Jan 2004.

[134] P. Griffiths Selinger, M. M. Astrahan, D. D. Chamberlin, R. A. Lorie, and T. G. Price. Access Path Selection in a Relational Database Management System. In *Proceedings of the 1979 International Conference on Management of Data (SIGMOD'79)*, pages 23–34, Boston, Massachusetts, 1979.

[135] Subhabrata Sen and Jia Wang. Analyzing peer-to-peer traffic across large networks. In *Proceedings of the second ACM SIGCOMM Workshop on Internet measurment*, pages 137–150. ACM Press, 2002.

[136] Luciano Serafini, Fausto Giunchiglia, John Mylopoulos, and Philip Bernstein. The Local Relational Model: Model and Proof Theory. Technical Report DIT-02-0009, 2001.

[137] SHA-1. Secure hash standard. National Institute of Standards and Technology. Publication 180-1, 1995.

[138] E. Patrick Shironoshita, Michael T. Ryan, and Mansur R. Kabuka. Cardinality Estimation for the Optimization of Queries on Ontologies. *SIGMOD Record*, 36(2):13–18, 2007.

[139] Lefteris Sidirourgos, George Kokkinidis, Theodore Dalamagas, Vassilis Christophides, and Timos Sellis. Indexing Views to Route Queries in a PDMS. *Journal of Distributed Parallel Databases*, 23:45–68, February 2008.

[140] Bernd Simon, Zoltan Miklos, Wolfgang Neijdl, Michael Sintek, and Joaquin Salvachua. Elena: A Mediation Infrastructure for Educational Services. In *Proceedings of Twelfth International World Wide Web Conference (WWW2003)*, Budapest, Hungary, May 2003.

[141] Michael Sintek and Stefan Decker. TRIPLE – A Query, Inference and Transformation Language for the Semantic Web. In *Proceedings of Deductive Databases and Knowledge Management (DDLP'2001)*, 2001.

[142] Ramakrishna Soma and V. K. Prasanna. Parallel Inferencing for OWL Knowledge Bases. In *ICPP '08: Proceedings of the 2008 37th International Conference on Parallel Processing*, pages 75–82, Washington, DC, USA, 2008. IEEE Computer Society.

[143] Steffen Staab and Heiner Stuckenschmidt, editors. *Semantic Web and Peer-to-Peer: Decentralized Management and Exchange of Knowledge and Information.* Springer-Verlag, 2006.

[144] Michael Steinbrunn, Guido Moerkotte, and Alfons Kemper. Heuristic and Randomized Optimization for the Join Ordering Problem. *VLDB Journal*, 1997.

[145] 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 World Wide Web Conference (WWW 2008)*, Beijing, China, 2008.

[146] Ion Stoica, Dan Adkins, Sylvia Ratnasamy, Scott Shenker, Sonesh Surana, and Shelley Zhuang. Internet Indirection Architecture. In *Proceedings of ACM SIGCOMM'02*, pages 73–86, August 2002.

[147] Ion Stoica, Robert Morris, David Liben-Nowell, David Karger, M. Frans Kaashoek, Frank Dabek, and Hari Balakrishnan. Chord: A Scalable Peer-to-Peer Lookup Protocol for Internet Applications. *IEEE/ACM Transactions on Networking*, 11(1):17–32, 2003.

[148] Jeremy Stribling, Isaac G. Councill, Jinyang Li, M. Frans Kaashoek, David R. Karger, Robert Morris, and Scott Shenker. OverCite: A Cooperative Digital Research Library. In *Proceedings of the 4th International Workshop on Peer-To-Peer Systems (IPTPS'05)*, pages 69–79, Ithaca, New York, USA, 2005.

[149] Heiner Stuckenschmidt and Jeen Broekstra. Time-Space Trade-offs in Scaling up RDF Schema Reasoning. In *Proceedings of Web Information Systems Engineering Workshop (WISE 2005)*, New York, NY, USA, 2005.

[150] Heiner Stuckenschmidt, Richard Vdovjak, Jeen Broekstra, Geert jan Houben, Tu Eindhoven, and Aduna Amersfoort. Towards Distributed Processing of RDF Path Queries. *International Journal of Web Engineering and Technology*, 2005.

[151] Sonesh Surana, Brighten Godfrey, Karthik Lakshminarayanan, Richard Karp, and Ion Stoica. Load Balancing in Dynamic Structured Peer-to-Peer Systems. *Perform. Eval.*, 63(3):217–240, 2006.

[152] Valentina Tamma, Ian Blacoe, Ben Lithgow Smith, and Michael Wooldridge. SERSE: Searching for Semantic Web Content. In *Proceedings of the AAMAS 2004 workshop on Challenges in the coordination of large scale multi-agent systems*, New York, July 2004.

[153] Christoph Tempich, Steffen Staab, and Andrian Wranik. REMINDIN': Semantic Query Routing in Peer-to-Peer Networks Based on Social Metaphors. In *Proceedings of the Thirteenth International World Wide Web Conference (WWW2004)*, New York, May 2004.

[154] Herman J. ter Horst. Completeness, Decidability and Complexity of Entailment for RDF Schema and a Semantic Extension Involving the OWL Vocabulary. *Web Semantics*, 3(2-3):79–115, 2005.

[155] Uwe Thaden, Wolf Siberski, Wolf tilo Balke, and Wolfgang Nejdl. Top-k Query Evaluation for Schema-Based Peer-to-Peer Networks. In *Proceedings of the 3rd International Semantic Web Conference (ISWC)*, 2004.

[156] Yannis Theoharis, Vassilis Christophides, and Greg Karvounarakis. Benchmarking Database Representations of RDF/S Stores. In *Proceedings of the 4rth International Semantic Web Conference (ISWC 2005)*, Galway, Ireland, 2005.

[157] Peter Triantafillou, Chryssani Xiruhaki, Manolis Koubarakis, and Nikos Ntarmos. Towards high-performance peer-to-peer content and resource sharing systems. In *Proceedings of the First Biennial Conference on Innovative Data Systems Research (CIDR 2003)*, January 2003.

[158] Christos Tryfonopoulos, Stratos Idreos, and Manolis Koubarakis. LibraRing: An Architecture for Distributed Digital Libraries Based on DHTs. In *ECDL*, pages 25–36, 2005.

[159] Dimitrios Tsoumakos and Nick Roussopoulos. A comparison of peer-to-peer search methods. In *Proceedings of International Workshop on Web and Databases (WebDB)*, pages 61–66, 2003.

[160] Jeffrey D. Ullman. *Principles of Database and Knowledge-Base Systems, Volume I.* Computer Science Press, 1988.

[161] Jeffrey D. Ullman. *Principles of Database and Knowledge-Base Systems, Volume II.* Computer Science Press, 1988.

[162] Jacopo Urbani, Spyros Kotoulas, Jason Maassen, Frank Van Harmelen, and Henri Bal. OWL reasoning with WebPIE: calculating the closure of 100 billion triples. In *Proceedings of the 8th Extended Semantic Web Conference (ESWC2010)*, Heraklion, Greece, 2010.

[163] Jacopo Urbani, Spyros Kotoulas, Eyal Oren, and Frank van Harmelen. Scalable Distributed Reasoning using MapReduce. In *Proceedings of the 8th International Semantic Web Conference (ISWC2009)*, October 2009.

[164] Maria Esther Vidal, Tomas Lampo, Edna Ruckhaus, Javier Sierra, and Amadis Martinez. OneQL: An Ontology-based Architecture to Efficiently Query Resources on the Semantic Web. In *Proceedings of the 5th International Workshop on Scalable Semantic Web Knowledge Base Systems (SSWS2009)*, Washington DC, USA, October 2009.

[165] Maria-Esther Vidal, Edna Ruckhaus, Tomas Lampo, Amadís Martínez, Javier Sierra, and Axel Polleres. Efficiently Joining Group Patterns in SPARQL Queries. In *Proceedings of the 7th Extended Semantic Web Conference (ESWC2010)*, pages 228–242, Heraklion, Greece, 2010.

[166] Jesse Weaver and James Hendler. Parallel Materialization of the Finite RDFS Closure for Hundreds of Millions of Triples. In *8th International Semantic Web Conference (ISWC2009)*, October 2009.

[167] Cathrin Weiss, Panagiotis Karras, and Abraham Bernstein. Hexastore: Sextuple Indexing for Semantic Web Data Management. In *Proceedings of 34th International Conference on Very Large Data Bases (VLDB 2008)*, volume 1, pages 647–659, Auckland, New Zealand, 2008.

[168] Kevin Wilkinson, Craig Sayers, Harumi A. Kuno, and Dave Raynolds. Efficient RDF Storage and Retrieval in Jena2. In *Proceedings of the 1st International Workshop on Semantic Web and Databases (SWDB 2003, co-located with VLDB 2003)*, Berlin, Germany, 2003.

[169] Ben Y. Zhao, Ling Huang, Jeremy Stribling, Sean C. Rhea, Anthony D. Joseph, and John D. Kubiatowicz. Tapestry: A Resilient Global-scale Overlay for Service Deployment. *IEEE Journal on Selected Areas in Communications*, 22(1), 2004.

[170] Yingwu Zhu and Yiming Hu. Efficient, Proximity-Aware Load Balancing for DHT-Based P2P Systems. *IEEE Transanction on Parallel and Distributed Systems*, 16:349–361, April 2005.

Appendix A

SPARQL Queries

For completeness we include the SPARQL queries used in our experimental evaluation.

LUBM benchmark

The LUBM benchmark [48] provides synthetic datasets and a collection of 14 queries. Below, we present the SPARQL queries we used in our experimental evaluation. A complete description of the benchmark can be found here `http://swat.cse.lehigh.edu/projects/lubm/`.

```
Q1: PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
    PREFIX ub: <http://www.lehigh.edu/~zhp2/2004/0401/univ-bench.owl#>
    PREFIX u0d0: <http://www.Department0.University0.edu/>
    SELECT ?X
    WHERE {
        ?X rdf:type ub:GraduateStudent .
        ?X ub:takesCourse u0d0:GraduateCourse0 .
    }

Q2: PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
    PREFIX ub: <http://www.lehigh.edu/~zhp2/2004/0401/univ-bench.owl#>
    SELECT ?X ?Y ?Z
    WHERE {
        ?X rdf:type ub:GraduateStudent.
        ?Y rdf:type ub:University .
        ?Z rdf:type ub:Department .
        ?X ub:memberOf ?Z .
        ?Z ub:subOrganizationOf ?Y .
        ?X ub:undergraduateDegreeFrom ?Y .
    }
```

```
Q4: PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
    PREFIX ub: <http://www.lehigh.edu/~zhp2/2004/0401/univ-bench.owl#>
    SELECT ?X ?Y1 ?Y2 ?Y3
    WHERE {
        ?X rdf:type ub:Professor .
        ?X ub:worksFor <http://www.Department0.University0.edu> .
        ?X ub:name ?Y1 .
        ?X ub:emailAddress ?Y2 .
        ?X ub:telephone ?Y3 .
    }

Q7: PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
    PREFIX ub: <http://www.lehigh.edu/~zhp2/2004/0401/univ-bench.owl#>
    PREFIX u0d0: <http://www.Department0.University0.edu/>
    SELECT ?X ?Y
    WHERE {
        ?X rdf:type ub:Student .
        ?Y rdf:type ub:Course .
        ?X ub:takesCourse ?Y .
        u0d0:AssociateProfessor0 ub:teacherOf ?Y .
    }

Q8: PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
    PREFIX ub: <http://www.lehigh.edu/~zhp2/2004/0401/univ-bench.owl#>
    SELECT ?X ?Y ?Z
    WHERE {
        ?X rdf:type ub:Student .
        ?Y rdf:type ub:Department .
        ?X ub:memberOf ?Y .
        ?Y ub:subOrganizationOf <http://www.University0.edu> .
        ?X ub:emailAddress ?Z .
    }

Q9: PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
    PREFIX ub: <http://www.lehigh.edu/~zhp2/2004/0401/univ-bench.owl#>
    SELECT ?X ?Y ?Z
    WHERE {
        ?X rdf:type ub:Student .
        ?Y rdf:type ub:Faculty.
        ?Z rdf:type ub:Course .
        ?X ub:advisor ?Y .
        ?X ub:takesCourse ?Z .
        ?Y ub:teacherOf ?Z .
```

```
    }

Q12: PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
     PREFIX ub: <http://www.lehigh.edu/~zhp2/2004/0401/univ-bench.owl#>
     SELECT ?X ?Y
     WHERE {
        ?X rdf:type ub:Chair .
        ?Y rdf:type ub:Department .
        ?X ub:worksFor ?Y .
        ?Y ub:subOrganizationOf <http://www.University0.edu> .
     }
```

Votes dataset

The votes dataset is a real-world dataset which describes the US Congress vote results of the 2004 bills voting process (`http://www.govtrack.us/data/rdf/`). In our experimental evaluation, we used the following queries found here `http://ldc.usb.ve/~mvidal/OneQL/datasets/queries/benchmark1`.

```
Q1: PREFIX vote:<http://govshare.info/2005/rdf/vote/>
    PREFIX votes: <http://www.rdfabout.com/rdf/usgov/congress/108/senate/votes/>
    SELECT ?A1 ?P1
    WHERE {
        ?A1 vote:voter ?P1.
        ?A1 vote:option "NoVote".
        votes:2004-69 vote:hasBallot ?A1 .
    }

Q2: PREFIX vote:<http://govshare.info/2005/rdf/vote/>
    SELECT ?A1 ?P1 ?E1
    WHERE {
        ?A1 vote:voter ?P1 .
        ?A1 vote:option "Aye".
        ?E1 vote:hasBallot ?A1 .
    }

Q3: PREFIX vote:<http://govshare.info/2005/rdf/vote/>
    PREFIX dc: <http://purl.org/dc/elements/1.1/>
    PREFIX people:<http://www.rdfabout.com/rdf/usgov/congress/people/>
    SELECT ?E ?T ?I ?X ?J
    WHERE {
        ?E vote:winner "Nay" .
        ?E dc:title ?T .
        ?E vote:hasBallot ?I .
        ?I vote:option ?X .
```

```
        ?J vote:option ?X .
        ?E vote:hasBallot ?J .
        ?J vote:voter people:L000174 .
    }

Q4: PREFIX vote:<http://govshare.info/2005/rdf/vote/>
    SELECT ?A ?B ?C ?D
    WHERE {
        ?A vote:voter ?B.
        ?A vote:option ?C.
        ?D vote:hasBallot ?A.
        ?D vote:winner ?C .
    }

Q5: PREFIX vote:<http://govshare.info/2005/rdf/vote/>
    SELECT ?A1 ?P1 ?E
    WHERE {
        ?A1 vote:voter ?P1.
        ?A1 vote:option "Aye".
        ?E vote:hasBallot ?A1.
        ?E vote:winner "Aye" .
    }

Q6: PREFIX vote:<http://govshare.info/2005/rdf/vote/>
    PREFIX votes: <http://www.rdfabout.com/rdf/usgov/congress/108/senate/votes/>
    SELECT ?P1 ?A1
    WHERE {
        ?A1 vote:voter ?P1.
        ?A1 vote:option "Nay".
        votes:2004-69 vote:hasBallot ?A1 .
    }

Q7: PREFIX vote:<http://govshare.info/2005/rdf/vote/>
    SELECT ?A ?B ?C
    WHERE {
        ?A vote:voter ?B.
        ?A vote:option "Nay".
        ?C vote:hasBallot ?A .
    }

Q8: PREFIX vote:<http://govshare.info/2005/rdf/vote/>
    SELECT ?A ?B ?C
    WHERE {
        ?A vote:voter ?B.
```

```
        ?A vote:option "NoVote".
        ?C vote:hasBallot ?A  .
    }

Q9: PREFIX vote:<http://govshare.info/2005/rdf/vote/>
    PREFIX people:<http://www.rdfabout.com/rdf/usgov/congress/people/>
    SELECT ?A1 ?P1 ?A2 ?X ?E
    WHERE {
        ?A1 vote:voter ?P1.
        ?A1 vote:option ?X.
        ?A2 vote:option ?X.
        ?A2 vote:voter people:L000174.
        ?E vote:hasBallot ?A1.
        ?E vote:hasBallot ?A2 .
    }
```