The 7th International Workshop on Scalable Semantic Web Knowledge Base Systems (SSWS 2011)

At the 10th International Semantic Web Conference (ISWC 2011), Bonn, Germany, October 24th, 2011

Preface

This volume contains the papers presented at 7th International Workshop on Scalable Semantic Web Knowledge Base Systems (SSWS 2011) held on October 23, 2011 in Bonn, Germany.

SSWS 2011 was the seventh instance in the sequence of successful Scalable Semantic Web Knowledge Base Systems workshops. This workshop focused on addressing scalability issues with respect to the development and deployment of knowledge base systems on the Semantic Web. Typically, such systems deal with information described in Semantic Web languages like OWL and RDF(S), and provide services such as storing, reasoning, querying and debugging. There are two basic requirements for these systems. First, they have to satisfy the applications semantic requirements by providing sufficient reasoning support. Second, they must scale well in order to be of practical use. Given the sheer size and distributed nature of the Semantic Web, these requirements impose additional challenges beyond those addressed by earlier knowledge base systems. This workshop brought together researchers and practitioners to share their ideas regarding building and evaluating scalable knowledge base systems for the Semantic Web.

This year we received 13 submissions. Each paper was carefully evaluated by two or three workshop Program Committee members. Based on these reviews, we accepted ten papers for presentation. The topics of the selected papers span the areas of large scale data stores, optimized representation mechanisms, and query processing. We sincerely thank the authors for all the submissions and are grateful for the excellent work by the Program Committee members.

September 2011

Achille Fokuoe
Thorsten Liebig
Yuanbo Guo

Table of Contents

Session 1: Invited Talk and RDF Literal Datatypes

RDF Literal Data Types in Practice 1
Ian Emmons, Suzanne Collier, Mounika Garlapati and Mike Dean

Session 2: Triple/Quad store management and query optimization

The Quad Economy of a Semantic Web Ontology Repository 14
Manuel Salvadores, Paul R. Alexander, Mark A. Musen and Natalya F. Noy

CumulusRDF: Linked Data Management on Nested Key-Value Stores . . . 30
Günter Ladwig and Andreas Harth

Optimizing Unbound-property Queries to RDF Views of Relational Databases 43
Silvia Stefanova and Tore Risch

Session 3: Scalable OWL reasoning and querying

Query Rewriting Under Query Extensions for OWL 2 QL Ontologies 59
Tassos Venetis, Giorgos Stoilos and Giorgos Stamou

Sound and Complete SHI Instance Retrieval for 1 Billion ABox Assertions 75
Sebastian Wandelt and Ralf Möller

Handling Cyclic Axioms in Dynamic, Web-Scale Knowledge Bases 90
Yingjie Li and Jeff Heflin

Probabilistic Abox Abduction in Description Logics 106
Murat Sensoy, Achille Fokoue, Mudhakar Srivatsa and Jeff Z. Pan

Session 4: Native and NoSQL RDF stores

Ontology Based Data Integration Over Document and Column Family Oriented NoSQL stores 122
Olivier Cure, Myriam Lamolle and Chan Le Duc

Resource centered RDF data management 138
Ralf Heese and Martin Znamirowski

Program Committee

Abraham Bernstein	University of Zurich, Switzerland
Oscar Corcho	Universidad Politécnica de Madrid, Spain
Mike Dean	Raytheon BBN Technologies, USA
Achille Fokoue	IBM Research, USA
Raúl García-Castro	Universidad Politécnica de Madrid, Spain
Yuanbo Guo	Microsoft, USA
Volker Haarslev	Concordia University, Canada
Pascal Hitzler	Wright State University, Ohio, USA
Thorsten Liebig	derivo GmbH
Marko Luther	DoCoMo Euro-Lab, Munich, Germany
Mariano Rodriguez	Free University of Bolzano, Italy
Sebastian Rudolph	AIFB, Karlsruhe Institute of Technology, Germany
Andy Seaborne	Epimorphics Ltd, UK
Kavitha Srinivas	IBM Research, USA
Jan Wielemaker	VU University Amsterdam, The Netherlands
Takahira Yamaguchi	Keio University, Japan

Additional Reviewers

Jinan El-Hachem
Basil Ell
Lorenz Fischer
Rolf Grütter
Jörg-Uwe Kietz
Adila Alfa Krisnadhi
Raghava Mutharaju
Minh Khoa Nguyen
Cong Wang
Kejia Wu
Ming Zuo

RDF Literal Data Types in Practice

Ian Emmons, Suzanne Collier, Mounika Garlapati, and Mike Dean

Raytheon BBN Technologies, Inc., Arlington, VA 22209, USA
{iemmons,scollier,mgarlapa,mdean}@bbn.com

Abstract. One of the more mysterious aspects of RDF (Resource Description Framework) is typed literals. For instance, confusion over the difference between a plain character string ("foo") and a string that is explicitly typed ("foo"^^xsd:string) is common. Also, questions often arise about comparisons between literals of the various numeric types (e.g., long, integer, decimal, and float). This paper explores how several popular triple stores handle literals via direct testing, and also compares their behavior to the relevant standards. Along the way, we highlight a number of implementation inconsistencies and some surprising aspects of the standards themselves.

1 Introduction

One of the more mysterious aspects of RDF (Resource Description Framework) [14] is typed literals. For instance, new and even moderately experienced Semantic Web practitioners often ask questions like these:

– What is the difference between a plain character string ("foo") and a string that is explicitly typed ("foo"^^xsd:string)?
– Under what conditions can literals of the various numeric types (long, integer, decimal, float, etc.) be compared?
– Can literals with different lexical forms but the same value, such as "47"^^xsd:decimal and "47.0"^^xsd:decimal, be compared?

While using our own Parliament triple store [1], we have wrestled with these questions from time to time, and so decided to investigate this topic thoroughly. In particular, we wanted to understand how other triple stores implement literal data types, and compare this against both our own implementation and the standards themselves. We give a summary of the relevant portions of the standards in Section 2, and then present our empirical results in Section 3. Along the way, we highlight a number of implementation inconsistencies and some surprising aspects of the standards themselves.

2 Literal Data Types in the Standards

All literals in RDF have a lexical form encoded as a Unicode string, and are either typed or plain [14]. A plain literal is a lexical form with an optional language tag ("foo" or "foo"@en), whereas a typed literal is a lexical form together with a data type URI ("foo"^^xsd:string). A data type defines a value space, the set of possible values, and a lexical space, the set of valid lexical forms, for literals of that type. RDF defines a set of data types by borrowing from the XSD specification [3], and most typed literals use one of these data types. Figure 1 shows the complete XSD type hierarchy. RDF defines one other data type, rdf:XMLLiteral, and also allows user-defined data types.

In addition, some RDF serializations such as Turtle [2] allow unquoted literal forms. In reality, these are not separate kinds of literals, but syntactic shortcuts for certain typed literals. For instance:

- An unquoted 3 is a shortcut for "3"^^xsd:integer
- An unquoted 3.14 is a shortcut for "3.14"^^xsd:decimal
- An unquoted true is a shortcut for "true"^^xsd:boolean

Because SPARQL syntax is based on Turtle, these shortcuts are valid within SPARQL queries [20].

In the absence of entailment, RDF and OWL maintain that literals are only equal when both their lexical form and data type are equivalent. The RDF specification states that two literals are equal if and only if all of the following conditions hold [14]:

- The two lexical forms compare equal, character by character
- Either both or neither have language tags
- The language tags, if any, compare equal
- Either both or neither have data type URIs
- The two data type URIs, if any, compare equal, character by character

This means that any two typed literals must have exactly the same lexical form and data type URIs to be considered equal. OWL follows similar equality rules, declaring "Two literals are structurally equivalent if and only if both the lexical form and the data type are structurally equivalent; that is, literals denoting the same data value are structurally different if either their lexical form or the data type is different." [15]

Although these are the strict rules of literal equality, by applying the RDF D-Entailment regime to XSD data types, the equality rules become more flexible [11]. Note that this entailment regime applies only to XSD

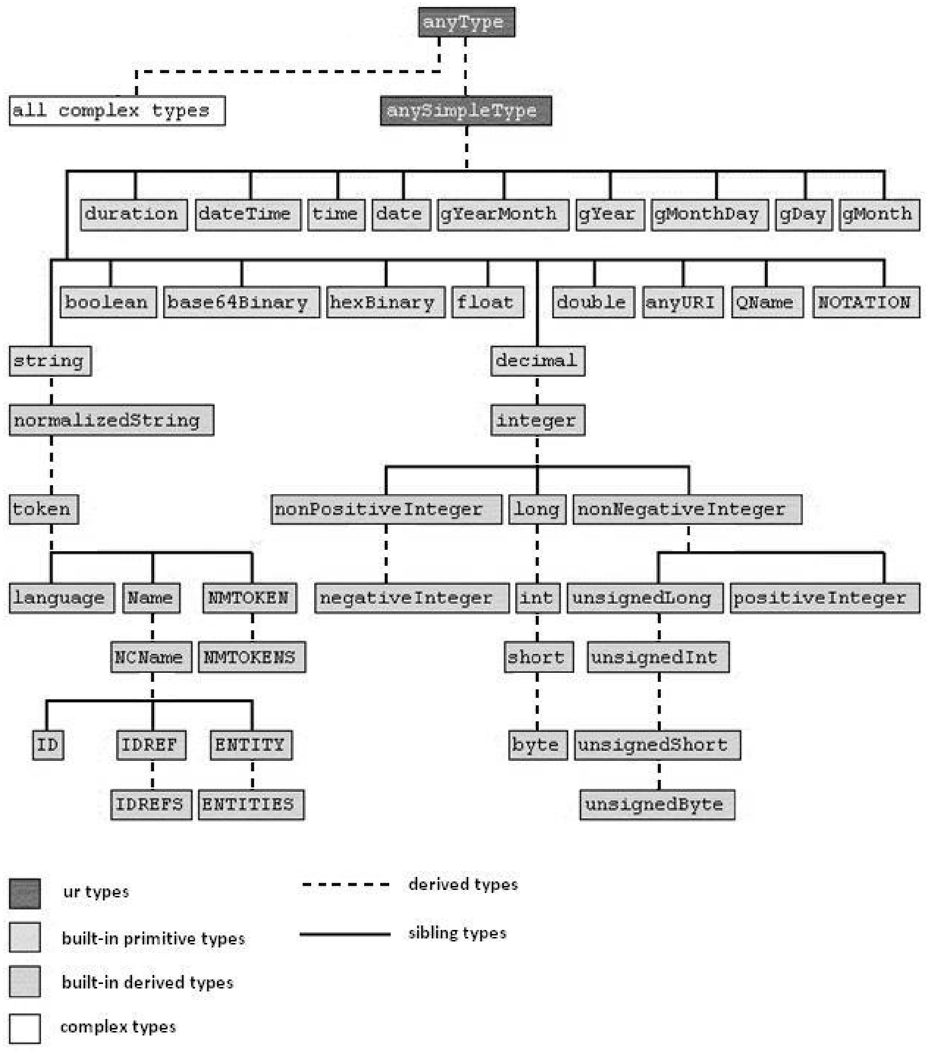


Fig. 1. Type Hierarchy of XSD Data Types [3]

data types, and adds nothing to comparisons of literals with language tags. Thus this paper will not discuss language-tagged literals any further. XML schema data types are defined in a hierarchical structure, with base types and derived types (see Figure 1). An XML schema derived type refers to a subset of the value space of its base type. Therefore, two literals that have the same primitive base data type and the same lexical forms are equal [3]. For example, because both int and byte are derived from decimal, “25”^^xsd:byte is equivalent to “25”^^xsd:int. Additionally, RDF Semantics explicitly equates plain literals (“foo”) and literals of type string (“foo”^^xsd:string):

> The value space and lexical-to-value mapping of the XSD data type xsd:string sanctions the identification of typed literals with plain literals without language tags for all character strings which are in the lexical space of the data type, since both of them denote the Unicode character string which is displayed in the literal [11].

The D-entailment regime also states that if two lexical forms map to the same value and have the same data type, then they entail each other [11]. For example, “14”^^xsd:decimal and “14.0”^^xsd:decimal are lexicographically different; however, because 14 and 14.0 map to the same value, these two literals are equivalent under this entailment rule.

Within SPARQL Filter clauses, additional type promotion rules apply, since the functions and operators are defined by the XML Query Language (XQuery) Operator Mapping. XQuery performs type promotion and subtype substitution as necessary in order to compare the values of operands separately from the data types [4]. The outcome (as seen in Section 3.3) is potentially different results for the following two SPARQL queries:

```
PREFIX xsd: <http://www.w3.org/2001/XMLSchema#>
SELECT ?x WHERE {
   ?x ?y "47"^^xsd:decimal .
}

PREFIX xsd: <http://www.w3.org/2001/XMLSchema#>
SELECT ?x WHERE {
   ?x ?y ?z .
   FILTER( ?z = "47"^^xsd:decimal )
}
```

There is considerable opportunity for confusion here: We found that even Raytheon BBN’s most experienced Semantic Web practitioners were surprised to learn that these queries are not equivalent.

Implementations of the recommendations are not required to support D-Entailment on XSD data types, but it is helpful. D-Entailment helps to reduce the unintended effects of syntactic choices made by authors, and makes data more interoperable. For instance, authors may choose to represent 12 as a decimal instead of an integer. However, when another user queries for “12”^^xsd:integer they would expect to find a match. Of the triple stores that we studied, Jena most strictly followed the RDF D-Entailment regime on XSD data types. AllegroGraph chose to only follow the string entailment rules, equating “foo” with “foo”^^xsd:string, but not allowing equivalence between different numeric types. Parliament, AllegroGraph, and Jena TDB follow strict literal equality. All of the triple stores we tested follow the SPARQL filter clause type promotion rules.

3 Existing Practice

An analysis of literal data types in relation to triple stores is important because most triple stores lack documentation on their storage and treatment of literals. Some users may expect triple stores to treat various literal forms (for example “foo”^^xsd:string versus “foo” or “3.14”^^xsd:decimal versus “3.14”^^xsd:float) as equivalent values. However, triple stores vary in their implementation. To analyze the relationship between literals and triple stores, we stored various literal representations in several different triple stores, and then queried for them to see what combinations of literals each triple store would match.

3.1 Methodology

In order to assess how triple stores handle a diverse set of literals, we created a test harness to drive each of several triple stores through a series of tests and compile the results. The particular triple stores we selected for testing are discussed in Section 3.2, and the results of our testing are discussed in Section 3.3. The harness stores a set of literal statements in a triple store and then runs a series of queries to assess which pairs of literal forms that triple store can match. Our test harness code and a spreadsheet containing our results can be found on our web site, here:

`http://asio.bbn.com/2011/10/ssws/LiteralDataTypes.zip`

To begin, the test harness reads RDF from an input file that contains a consistent selection of typed, untyped, and irregular literals. (By “irregular”, we mean typed literals whose lexical form are inconsistent with their type, such as “foo”^^xsd:integer. Such literals are interesting because they are valid RDF, even though most people would regard them as an error.) The input file containing these literals was in Turtle format when possible, but we used N-Triples for triple stores that do not support Turtle.

Based upon a W3C compilation of XSD data types and their mappings and relevance to the standards RDF, OWL, SPARQL, and RIF [23], we chose a sampling of XSD data types to test. The literals we tested fell into these categories:

- Strings: both plain and typed
- Numbers: decimal, integer, long, int, double, and float
- Dates and times: dateTime, date, time, and gYear
- Miscellaneous: anyURI, hexBinary, base64Binary, and boolean
- Irregular: Literals whose lexical form is not within the lexical space of their data type, such as “foo”^^xsd:integer

The test harness adds one literal from the input file at a time to the triple store under test, and then a preselected set of simple queries and filter queries are run against the literal. By *simple query,* we mean a query whose where clause consists of a single triple pattern whose object is the literal being tested. In contrast, a *filter query* is a query whose where clause consists of a single triple pattern with a variable in the object position, along with a filter that restricts that variable to be equal to the literal under test. Examples of simple and filter queries (specifically querying “47”^^xsd:decimal) can be seen in Section 2. After the queries are run on the current literal statement, the triple store is cleared and the process is repeated until all the literal statements in the input file have been tested. For each query, if its result set is non-empty, then the spreadsheet cataloging the results indicates this with a “Yes,” while “No” indicates an empty result set.

3.2 Software Tested

We tested the literal handling behavior of five triple stores: Jena’s in-memory store, Jena TDB, Parliament, AllegroGraph, and OWLIM.

Jena In-Memory: Jena is an open source Java framework for building Semantic Web applications [13]. It can read and write RDF in many file formats, and it also includes a SPARQL query processor called ARQ. Its highly layered and pluggable architecture makes it an ideal front end for other triple stores as well — Mulgara, Virtuoso, OWLIM, AllegroGraph, and Parliament can all use Jena in this fashion. When used on its own, it provides an easy-to-use and well documented in-memory triple store. Jena's in-memory store accepts irregular literals.

Jena TDB: TDB is an optional subsystem of Jena for persisting RDF and OWL data that allows for high performance and large scale storage and query [13].

Parliament: Parliament is an open source, high performance, and standard compliant triple store for the Semantic Web, written by Raytheon BBN Technologies [1]. It pairs Jena's query processor with an innovative back-end store, and customizes Jena's query processor to optimize queries so as to derive maximum benefit from the back-end's data organization. For this paper in order to meaningfully derive how each triple store uniquely handles literals, the Jena API/interface was not used in conjunction with any other triple store other than itself, Jena TDB and Parliament. The test harness loaded Parliament from a Turtle file.

AllegroGraph: AllegroGraph is a high performance database and application framework for the Semantic Web from Franz [8]. It supports various clients (Python, Java, Jena, Lisp, Ruby, etc.) and has well documented examples and tutorials for implementation. AllegroGraph can read and write RDF in RDF/XML and N-Triples file formats but not Turtle. For this reason we were unable to add unquoted literals to AllegroGraph. AllegroGraph also will not accept irregular data types such as "foo"^^xsd:integer. Consequently, there are some blank cells in the AllegroGraph column of the test results.

OWLIM: OWLIM is a triple store from Ontotext [16]. Just as Parliament uses Jena as a front end and query processor, OWLIM uses another popular Java framework for Semantic Web applications called Sesame [18]. There are several versions of OWLIM available with varying levels of scalability and price points. OWLIM-Lite was used in this study in conjunction with the Sesame library and the SPARQL query language. It supports many file formats, including Turtle, used here. OWLIM cannot accept irregular literals such as "foo"^^xsd:integer.

Other Triple Stores: We chose the triple stores above because of their popularity, free availability, and relevance to our work here at BBN. We also tried to achieve a diversity of query processors, while still emphasizing Jena because it is used in our own triple store, Parliament. There are several other well-known triple stores that we would like to test, but did not due to time constraints. These include (but are not limited to) Virtuoso, Mulgara, and Oracle.

3.3 Analysis of Results

This section gives an overview of the results of our testing. Our complete results can be found on our web site, here:

`http://asio.bbn.com/2011/10/ssws/LiteralDataTypes.zip`

Keep in mind that Jena and Parliament accept all literals. AllegroGraph does not support Turtle syntax, and therefore does not accept unquoted literals (47, true). Also, OWLIM and AllegroGraph do not accept irregular literals (e.g., “foo”^^xsd:integer). The empty blanks in the spreadsheet are indicative of these limitations.

Most of the literal types chosen are siblings in the inheritance hierarchy, but the types decimal, integer, long, and int form an inheritance chain. The following sections will explain how each category of literals (numbers, strings, times, miscellaneous, and irregular) was handled by the various triple stores.

Numeric Types: With the numeric types (float, double, decimal, and all the types derived from decimal), the following kinds of numeric comparisons are of particular interest:

1. Literals that match exactly, according to the strict rules of RDF without entailment
2. Literals with identical lexical forms but differing types that share a common primitive base data type, e.g., decimal and integer
3. Literals with identical lexical forms but differing types that do not share a common primitive base data type (most often, this means sibling types), and yet whose types are intuitively compatible, e.g., float and integer
4. Literals with identical types but differing lexical forms that map to the same point in value space, e.g., “47”^^xsd:long versus “+47”^^xsd:long

Table 1 summarizes the results for numeric literals in terms of these categories. (The numbers in the table correspond to the categories of comparisons in the list above.)

Strings: The key question with strings is which triple stores see plain literals (“foo”) and typed strings (“foo”^^xsd:string) as equivalent. Table 2 summarizes how strings are handled by the various triple stores.

Triple Store	**Query Type**	**Matched Relationships**	**Unmatched Relationships**
Jena in-memory	Simple	1, 2, 4	3
Jena TDB, AllegroGraph, Parliament, OWLIM	Simple	1	2, 3, 4
Jena in-memory, Jena TDB, AllegroGraph, Parliament, OWLIM	Filter	1, 2, 3, 4	

Table 1. Comparison Results for Numeric Literals

Triple Store	**Query Type**	**Typed String versus Plain Literal**
Jena in-memory, AllegroGraph	Both (simple and filter)	Match
Jena TDB, Parliament, OWLIM	Simple	No Match
Jena TDB, Parliament, OWLIM	Filter	Match

Table 2. Comparison Results for Strings

Temporal Types: The temporal XSD types (dateTime, date, time, and gYear) are treated by all the triple stores in the same manner. For all triple stores, and in both simple and filter queries, only exact matches are found.

anyURI: XSD type anyURI acts much like the temporal data types in that only exact matches result in a positive comparison, except in one case: Jena's in-memory store finds matches between anyURI literals and strings (either typed or plain) in simple queries only. Within a filter, anyURI literals and strings do not match in any triple store.

hexBinary and base64Binary: Across the board, the XSD types hexBinary and base64Binary compare equal only in the case of exact matches. Interestingly, a hexBinary literal will not compare equal to the base64Binary literal that represents the same octet sequence. For example, 1111 in hex converts to ERE= in base64, but "1111"^^xsd:hexBinary does not match "Ere="^^xsd:base64Binary in any triple store. This is due to the fact that hexBinary and base64Binary are sibling data types and do not derive from a common primitive base data type.

Boolean: All triple stores match unquoted booleans (true) and typed booleans ("true"^^xsd:boolean) in both simple and filter queries (except AllegroGraph as it does not accept unquoted literals). This is not a surprise, because unquoted booleans are simply a syntactic shortcut for typed booleans in Turtle. A more interesting result is that no triple store matches unquoted or typed booleans against typed or plain string literals with the same lexical form (e.g., “true”^^xsd:string and “true”).

Irregular Literals: OWLIM and AllegroGraph do not recognize and will not accept irregular data types such as “foo”^^xsd:integer. With other triple stores, such literals compare equal in the case of exact matches. The more interesting result is what happens when comparing an irregular literal against a string (either typed or plain) with the same lexical form. Table 3 shows what happens in this case.

Triple Store	**Query Type**	**Typed or Plain String versus Irregular Literal**
Jena in-memory	Simple	Match
Jena in-memory	Filter	No Match
Jena TDB, Parliament	Both	No Match
AllegroGraph, OWLIM	Both	N/A — unable to store irregular literals

Table 3. Comparison Results for Irregular Literals

Overall in our tests, Jena’s in-memory store stood out as the most accepting of types that displayed a conceptual similarity (e.g., “foo” versus “foo”^^xsd:string or “47”^^xsd:decimal versus “47”^^xsd:integer) for both simple queries and filter queries.The Parliament triple store is most consistent with the way in which AllegroGraph and OWLIM treat literals in that these three triple stores are less permissive for most conceptually similar data types.

4 Related Work

Most of the related work in this domain is in the language specifications themselves [14] [24] [11] [20]. Because these specifications can be complex, the W3C produced a Working Group note, “XML Schema Datatypes in RDF and OWL” [6], which clarifies typed literal equality via the D-entailment regime.

Other work emphasizes the importance of using D-entailment in dealing with inconsistencies of instances [19] and specifics of the entailment regimes [12]. Many triple store studies have also been completed; however, they have mostly focused on performance [21] [22]. Although not directly targeted at RDF data types, Garcia-Castro and Gomez-Perez explored the interoperability of semantic web technologies using OWL. They created an interoperability benchmark to evaluate tools and they concluded that interoperability between Semantic Web tools is very low [9].

5 Conclusions

The treatment of typed literals by triple stores is fairly consistent, with the exception of Jena's in-memory store. These triple stores differ in their treatment of conceptually similar types for a simple query. Jena TDB, Parliament, OWLIM, and AllegroGraph implement strict literal equality. The Jena developers, on the other hand, stated "the formal semantics of both RDF and OWL makes it clear that entailment is a core feature of the Semantic Web recommendation" [5]. Consequently, the Jena in-memory model uses the D-Entailment regime to equate literals with derived types [10]. Jena also implements the D-entailment rule which equates literals with identical types and values but different lexical forms. Additionally, both Jena's in-memory store and AllegroGraph equate typed and untyped strings in simple queries. It is surprising that the AllegroGraph developers decided to implement D-Entailment for strings, but not for numeric types. Jena's in-memory store is also more permissive with the XSD data type anyURI, when matched against string type in simple queries. Interestingly, Jena's in-memory store matches irregular literals with strings with the same lexical form. This seems to deviate from the D-entailment rules. All triple stores followed the same conventions for temporal, hexBinary, base64Binary and boolean types. Overall, Jena follows D-Entailment, allowing more flexibility in user input, while the other triple stores tend to follow strict literal equality.

It is also crucial to add that in contrast to simple queries, filter queries recognize sibling and derived type relationships in query results for most triple stores. This indicates that filter queries have a more permissive path in parsing derived and sibling types. As defined by SPARQL [20], filter queries use the XML Query language (XQuery) type promotion [4]. This results in differences in query responses between apparently equivalent filter queries and simple queries.

There are several possibilities for expanding on this research in the future. The most obvious is to expand the testing to include more triple stores (such as Virtuoso, Mulgara, and Oracle) as well as SPARQL front ends to relational databases (such as D2RQ, Revelytix' Spyder, and BBN's Asio). Another possibility is to profile the use of typed literals in various subsets of the Semantic Web community to better understand how aspects of the type system affect real users. Some work has been done in this regard based on the billion triples challenge corpus from ISWC 2008 [7], but a more in-depth look at current data is warranted.

Finally, it would be interesting to better understand how the disparity between literal matching in simple queries and filter queries affects query optimizers. For instance, a filter that compares a variable for equality to a known URI can usually be optimized away by directly substituting the URI in place of the variable throughout the rest of the query. However, we have shown that a filter testing for equality to a literal cannot be so easily optimized away. This may impact how developers write their queries and improve overall performance time.

As evidenced by semantic tools research [9] and our studies, semantic technologies lack interoperability due to conflicting standards and developer design decisions. In order for the semantic web scalability to become a reality, interoperability is essential. The design decisions made by developers not only effect the results users receive via queries, but the performance of their system. OWLIM specifically states in their documentation that they do not use the D-entailment regime because the performance penalty is too high [17]. In further studies we would like to use larger data sets to investigate how the triple stores implementation of RDF data type equality effects their scalability. This data will help users decide which triple store to select based on performance and treatment of RDF literals.

Differing standards in the treatment of typed literals pose issues for users who expect conceptually similar types to match in their queries. Also the triple stores we studied differed greatly in their RDF implementation, yet their design decisions were undocumented. The different implementations lead to varying results, and it is pertinent that users know what they are using. As the semantic web becomes an increasingly popular framework for users, syntactic differences in typed literals are inevitable. It is critical that the community be aware of the inherent differences in the treatment of typed literals in order to promote correctness and maintainability of implementations, uniformity of practice, and simplicity of standards.

References

1. BBN Technologies: Parliament, `http://parliament.semwebcentral.org/`
2. Beckett, D., Berners-Lee, T.: Turtle — Terse RDF Triple Language, `http://www.w3.org/TeamSubmission/turtle/`
3. Biron, P.V., Malhotra, A. (eds.): XML Schema Part 2: Datatypes Second Edition. W3C (October 2004), `http://www.w3.org/TR/xmlschema-2/`

4. Boag, S., Chamberlin, D., Fernández, M.F., Florescu, D., Robie, J., Siméon, J. (eds.): XQuery 1.0: An XML Query Language (Second Edition). W3C (December 2010), http://www.w3.org/TR/xquery/
5. Carroll, J.J., Dickinson, I., Dollin, C.: Jena: Implementing the Semantic Web Recommendations. Tech. rep., HP Laboratories Bristol (December 2003), http://www.hpl.hp.com/techreports/2003/HPL-2003-146.pdf
6. Carroll, J.J., Pan, J.Z. (eds.): XML Schema Datatypes in RDF and OWL. W3C (March 2006), http://www.w3.org/TR/swbp-xsch-datatypes/
7. Dean, M.: Toward a Science of Knowledge Base Performance Analysis. In: Invited Talk, 4th International Workshop on Scalable Semantic Web Knowledge Base Systems (SSWS2008). p. 20. Karlsruhe, Germany (October 2008), http://asio.bbn.com/2008/10/iswc2008/mdean-ssws-2008-10-27.ppt
8. Franz, Inc.: AllegroGraph, http://www.franz.com/products/allegrograph/
9. Garcia-Castro, R., Perez, A.G.: Interoperability results for Semantic Web technologies using OWL as the interchange language. Journal of Web Semantics: Science, Services and Agents in the World Wide Web November, 278–291 (2010)
10. Glimm, B., Ogbuji, C. (eds.): SPARQL 1.1 Entailment Regimes. W3C (May 2011), http://www.w3.org/TR/sparql11-entailment/\#DEntRegime
11. Hayes, P. (ed.): RDF Semantics. W3C (February 2004), http://www.w3.org/TR/2004/REC-rdf-mt-20040210
12. ter Horst, H.J.: Completeness, decidability and complexity of entailment for RDF Schema and a semantic extension involving the OWL vocabulary. Web Semantics: Science, Services and Agents on the World Wide Web 3, 79–115 (2005), http://www.websemanticsjournal.org/index.php/ps/article/download/66/64
13. HP Labs Semantic Web Research: Jena, http://www.hpl.hp.com/semweb/
14. Klyne, G., Carroll, J. (eds.): Resource Description Framework: Concepts and Abstract Syntax. W3C (February 2004), http://www.w3.org/TR/2004/REC-rdf-concepts-20040210/
15. Motik, B., Patel-Schneider, P.F., Parsia, B. (eds.): OWL 2 Web Ontology Language: Structural Specification and Functional-Style Syntax. W3C (October 2009), http://www.w3.org/TR/2009/REC-owl2-syntax-20091027/
16. Ontotext: OWLIM, http://www.ontotext.com/owlim/
17. Ontotext: Primer Introduction to OWLIM, http://owlim.ontotext.com/display/OWLIMv41/Primer+Introduction+to+OWLIM
18. OpenRDF: Sesame, http://openrdf.org
19. Polleres, A., Hogan, A., Harth, A., Decker, S.: Can we ever catch up with the web. Semantic Web Journal 1, 45–52 (2010)
20. Prud'hommeaux, E., Seaborne, A. (eds.): SPARQL Query Language for RDF. W3C (January 2008), http://www.w3.org/TR/rdf-sparql-query/
21. Revelytix, Inc.: Triple store evaluation analysis report. Tech. rep., Revelytix, Inc. (September 2010), http://www.revelytix.com/sites/default/files/TripleStoreEvaluationAnalysisResults.pdf
22. Rohloff, K., Dean, M., Emmons, I., Ryder, D., Sumner, J.: An evaluation of triple-store technologies for large data stores. In: On the Move to Meaningful Internet Systems 2007: OTM 2007 Workshops. pp. 1105–1114. Springer, Vilamoura, Portugal (2007), LNCS 4806
23. W3C: XSD Datatypes (June 2011), http://www.w3.org/2011/rdf-wg/wiki/XSD_Datatypes
24. W3C OWL Working Group (ed.): OWL 2 Web Ontology Language Document Overview. W3C (October 2009), http://www.w3.org/TR/owl2-overview/

The Quad Economy of a Semantic Web Ontology Repository

Manuel Salvadores, Paul R Alexander,
Mark A. Musen, and Natalya F. Noy

Stanford Center for Biomedical Informatics Research
Stanford University, US
{manuelso,palexander,musen,noy}@stanford.edu

Abstract. Quad stores have a number of features that make them very attractive for Semantic Web applications. Quad stores use Named Graphs to give contextual information to RDF graphs. Developers can use these contexts to incorporate meta-data such as provenance and versioning. The OWL language specification provides the *owl:imports* construct as one of the key ways to reuse the ontologies on the Semantic Web. When one ontology imports another through the *owl:imports* statement, all axioms from the imported ontology are brought to the source ontology. When implementing an online ontology repository, we have explored a number of different approaches to reflect these imports and contextual information in a quad store. These approaches have a direct impact on storage requirements, query articulation, and reusability.
This paper describes different models to represent contextual information, ontology imports and versioning. We have performed the experiments in the context of storing ontologies from the BioPortal ontology repository. This repository has 304 ontologies, with multiple versions for many of them. We extract metrics on the levels of imports, and the storage and performance requirements and discuss the trade-offs among query articulation, reusability and scalability.

Keywords: Ontology Repository, Scalability, Quad Stores, SPARQL

1 Introduction

Ontology repositories act as a gateway for users who need to find ontologies for their applications. Research institutions and companies submit their ontologies to these repositories in order to promote their vocabularies and to encourage inter-operation. In biomedicine, cultural heritage, and other domains, many of the ontologies and vocabularies are extremely large, with tens of thousands of classes. For example, SNOMED CT, one of the key terminologies in biomedicine, has almost 400,000 classes. The Gene Ontology (GO) has 34,000 classes. These ontologies and terminologies are updated on a regular basis, some very frequently. For example, a new version of the Gene Ontology is published daily.

In our laboratory, we have developed BioPortal, a community-based ontology repository for biomedical ontologies [1].[1] Users can publish their ontologies to BioPortal, submit new versions, browse the ontologies, and access the ontologies and their components through a set of REST services. BioPortal provides search across all ontologies in its collection, a repository of automatically and manually generated mappings between classes in different ontologies, ontology reviews, new term requests, and discussions generated by the ontology users in the community.

At the time of this writing, BioPortal has 304 ontologies, with 5.3 million classes among them. Many of these ontologies have multiple versions in the repository. Many of the ontologies import one another or import ontologies that are not in the BioPortal repository. Finally, the ontologies come in a number of different formats, and thus are persisted in different backends using various APIs. There is a single Lucene index of classes to enable search across ontologies and the BioPortal REST API provides uniform access to all the ontologies in the repository so that API users can be agnostic about the individual formats. For each ontology, we provide access to the latest version of the ontology, and partial access to the earlier versions.

We are currently working on creating a single database—a quad store—for all the ontologies and their associated metadata. As part of the design, we evaluated the cost, in terms of data size, of representing and materializing various aspects of the ontologies, including the import structure of the ontologies. In this paper, we report on our analysis and the trade-offs with respect to representing a set of ontologies and their versions in a quad store.

Specifically, this paper makes the following contributions:

- We analyze the anatomy of a large-scale ontology repository with respect to the frequency of updates and the structure of imports among the ontologies
- We compare the storage requirements for representing the ontologies and their versions in a quad store, analyzing the effect of materializing imports.

2 Background

In this section, we provide background on the quad store technology, and its relationship to the SPARQL query language. We discuss the use of these technologies to represent and query ontology metadata.

2.1 Quad Stores, Named-Graphs and RDF

The Resource Description Framework (RDF)[2] is the core standard for representing data on the Semantic Web. The Semantic Web community is promoting RDF stores (or *triple stores*) as the key data storage technology. In an RDF store, data are not bound to a schema. Furthermore, we can assert data into an

[1] `http://bioportal.bioontology.org`
[2] `http://www.w3.org/TR/2004/REC-rdf-concepts-20040210/`

RDF store directly from RDF sources where it is represented in native Semantic Web formats (e. g. RDF/XML or Turtle files). SPARQL is a query language designed to express queries for data viewed or stored as RDF. SPARQL introduces the notion of Named Graphs [4, 5] to represent and encapsulate distinct RDF Datasets.[3]

The notion of Named Graphs gave rise to the development of *quad stores*. A quad, unlike a triple, has a fourth component that often is referred as "named-graph," "context" or "model." In this paper, for the sake of consistency, we will always refer to the fourth component as named-graph. Today, there are numerous RDF databases that in essence are quad stores due to their native support for named-graphs. Examples of these systems include 4store [8], Virtuoso [7], Jena TDB [11] and Sesame [3]. These tools, and many others, have adopted SPARQL as the language to query RDF graphs. SPARQL 1.0 defines a standard mechanism for querying named-graphs. This type of query can be achieved with either `FROM NAMED` or `GRAPH` clauses. There are two widely accepted formats to represent named-graphs in RDF: N-Quads [6] and TriG [2]. Most quad stores are compatible with these two formats. However, neither of these formats is an RDF standard. Indeed, graph identification in RDF was listed as one of the core working items in the RDF Next Steps Workshop and, possibly, it will be standardized as part of the next RDF specification.[4]

2.2 Metadata, Versioning, Named-Graphs and SPARQL

Named graphs can be used in two ways. First, we can use them to represent statements about the same resource that were made in a different context. Each named graph contains statements that were made in a specific context (e.g., by a particular information source). Second, we can use named graphs to collect metadata and attach it to a document, in our case to attach metadata to an ontology.

BioPortal keeps several versions of each ontology. For each version, it holds metadata about its content, authorship, creation and modification dates, projects, categories, and so on. It is important that our backend infrastructure enables us to combine queries that filter information based on the metadata about both the ontologies and the content of the ontologies themselves. Quad store technology is the clear option to implement this kind of capability.

In quad stores, IRIs are used to identify graphs such that other statements can "say" things about graphs. This auto-referenceable mechanism allows us to incorporate metadata. As an example, the quads below—in TriG format—show how we can say things about a graph. In this example, we use the IRI `GraphX` to identify the named-graph that holds the ontology containing the term "Aromatic Amino Acids." Similarly, `GraphXMetadata` is the IRI where triples that say things about `GraphX` are held. This is just a simple example and is not necessarily the model that we will use to represent metadata for ontologies.

[3] `http://www.w3.org/TR/2008/REC-rdf-sparql-query-20080115/#rdfDataset`
[4] `http://www.w3.org/2011/rdf-wg/wiki/TF-Graphs`

```
:GraphX {
  :AromaticAminoAcid rdf:type owl:Class;
                     rdfs:subClassOf :SpecificAminoAcid, :AminoAcid;
                     rdfs:label "Aromatic Amino Acid" .
    (... some other ontological axioms in RDF/Turtle ...) }

:GraphXMetadata {
  :GraphX rdf:type owl:Ontology;
          dct:created "2001-09-01"^^xsd:date;
          rdfs:label "Amino Acid";
          dct:format "OWL";
          foaf:primaryTopic <http://bioportal.bioontology.org/ontologies/1054>;
    (... some other metadata statements ...) }
```

This technique has become very popular for RDF graph versioning. We could state things such as `X` is replaced by `Y` or that `Z` is a version of `T`. The `data.gov.uk` project uses a similar versioning model using named graphs [14].

As you can see, named-graphs are first-class objects in a quad store and the IRIs used to identify graphs become part of the global graph and can thus be retrieved with SPARQL queries. The SPARQL 1.0 specification defines two query clauses to filter and/or bind specific named-graphs. These clauses are `FROM NAMED` and `GRAPH`. For instance, with the following query, we would retrieve all the ontologies `?ont` where a triple has a binding on subject or object to `:SpecificAminoAcid`. The query also selects the predicates `?p` that match such bindings.

```
SELECT DISTINCT ?ont ?p WHERE {
    GRAPH ?g {
        { ?x ?p :SpecificAminoAcid . }
        UNION
        { :SpecificAminoAcid ?p ?x . }
    }
    ?g rdf:type owl:Ontology .
}
```

The `GRAPH` clause can be used both with query variables or constant IRIs depending on whether or not we need to match a specific named-graph or just filter out named-graphs that do not satisfy other conditions. `FROM NAMED` is used only to declare a set of graph IRIs where a set of triple patterns is going to be satisfied. Combining `GRAPH` and `FROM NAMED` is possible, and we can constrain queries to a set of graphs and still identify the graph which is the source of information.

As an example, the following query is applied over an RDF dataset that contains one default graph and two named graphs. Moreover, in the `WHERE` clause we obtain the graph where things of type `:SpecificAminoAcid` are bound. This query reflects the case where we can combine queries that retrieve both metadata and data about the ontologies.

```
SELECT ?g ?label_g ?s ?label_s
FROM :GraphMetadata
FROM NAMED :GraphX
FROM NAMED :GraphY
WHERE {
   GRAPH ?g {
        ?s rdf:type :SpecificAminoAcid .
        ?s rdfs:label ?label_s .
   }
   ?g rdfs:label ?label_g .
}
```

The next section describes BioPortal's architecture, its limitations and some reasons to motivate the replacement of the current backend technology with a more scalable solution.

3 Motivation

Two different motifs drive this research:

1. Scalability: The BioPortal backend interacts with several databases and APIs (Section 3.1). This variety of non-compatible sources makes it extremely hard to implement a uniform query interface across ontologies.
2. Provenance and Versioning: Currently, BioPortal treats every ontology as a whole, including the materialization of all the `owl:imports`. With this approach, it is very difficult to track provenance and to keep several versions of ontologies at the term/class level.

3.1 BioPortal Architecture

BioPortal is an open library of biomedical ontologies that may be accessed using a Web-based user interface or RESTful Web services.[5] The Web-based user interface allows users to browse, search, and visualize ontologies and also facilitates community participation in the ontology lifecycle, including providing reviews of ontologies, mappings between terms, comments, and new term proposals.

This Web-based user interface is driven by a variety of RESTful services, including services that expose information about terms in ontologies, mappings, notes, and metadata about the ontologies themselves. This metadata includes the information on who uploaded each ontology into the system, which authorities support it, the size of the ontology, and other information that is specific to BioPortal. These RESTful services are accessed using standard HTTP verbs (POST, GET, PUT, DELETE) that are roughly equivalent to create, retrieve, update, and delete operations.

Users of these services are able to access all the ontology formats using a standard XML output, which is one of the major advantages of using BioPortal

[5] `http://bioportal.bioontology.org`

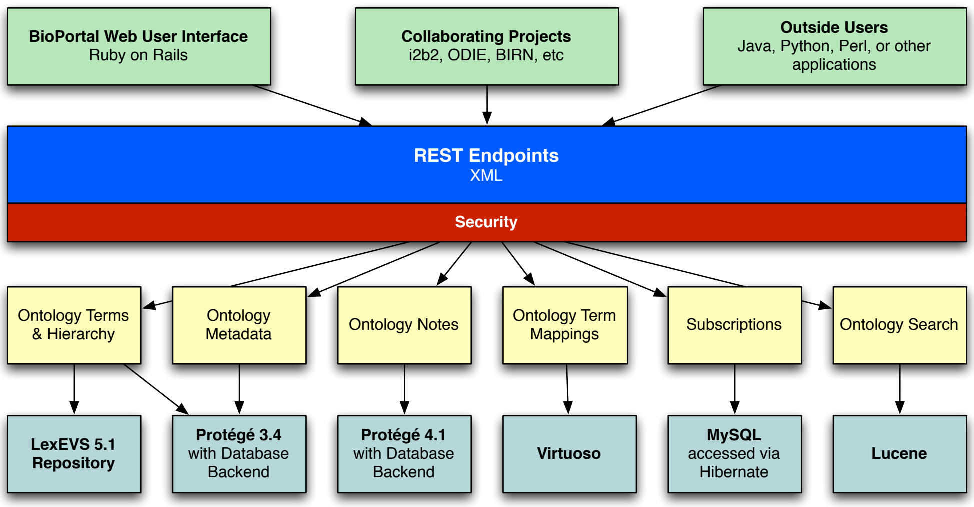


Fig. 1. BioPortal Architecture

over other systems that provide access to ontological content. Unfortunately, supporting the variety of ontology formats and related artifacts (mappings, notes, metadata) adds several layers of complexity over a traditional RDBMS solution.

In order to support multiple ontology formats, BioPortal currently utilizes two applications, LexEVS and Protégé. LexEVS is responsible for parsing and storing terminologies in formats that are primarily used in the biomedical domain: OBO Format and Rich Release Format (RRF). Protégé handles OWL, OWL2, and Protégé Frames (Figure 1). The systems use two entirely different models for storing information: Protégé relies on a RDBMS solution; LexEVS uses a combination of RDBMS and file-based storage.

In addition to ontology content, we also track a set of metadata related to each ontology in the system. We represent the metadata using an OWL ontology that we developed for this purpose, the BioPortal Metadata Ontology [10]. The metadata itself are a set of instances in this OWL ontology. Thus, we ultimately use the Protégé database backend as a store for metadata information. Protégé provides a flexible model for working with data and is well-suited for integrating and extending existing ontologies, such as the Ontology Metadata Vocabulary (OMV) [12]. Unfortunately, Protégé lacks some functionality that is generally accepted as standard on quad storage platforms, including the ability to query easily across records in different ontologies and scalability into the tens-of-million triples range.

Therefore, as you can see in Figure 1, there is no single uniform storage for the ontologies or their metadata. We implement this uniform layer through an API.

However, as the amount of information in BioPortal grew, we faced scalability issues with this approach.

We needed to represent 4.5 million term mappings between terms in BioPortal ontologies. Each mapping connects two terms from different ontologies, source and target. Each mapping also has several metadata properties associated with it. Protégé was not scalable enough for this dataset and thus we decided to represent the mappings in a quad store. Quad stores provided us with the same kind of flexibility that Protégé offers with the added ability to scale easily to support millions of triples and to provide opportunities for members of the Semantic Web community to query our data using a SPARQL endpoint.

However, using a quad store only for mappings turned out to have its own scalability problems: The triples that represent mappings are completely isolated from the storage of the ontologies themselves in Protégé and LexEVS. The mapping triples had only the URIs for the source and target terms. Thus, if the user needed labels, or any other information about these terms, a single query in the system that generated a result set with ten mappings needs at least 20 queries in order to correlate information about related terms. The BioPortal Web-based user interface shows 100 mappings on each page, and therefore at least 200 calls to the RESTful services are required to retrieve term information for a single page. While we do cache results, the experience can be less than ideal for users.

The same type of issues arise when examining systems for providing ontology notes, provisional terms, or any information about ontology content that isn't restricted to a single ontology, such as a query that looks for terms that share an ID across all ontologies. We implemented a Lucene-based index to overcome some of these issues, which will remain an integral part of the BioPortal architecture due to its speed in performing free-text searches. However, we clearly need a more comprehensive, scalable, and uniform solution.

Moreover, it is worth mentioning that both BioPortal's UI and REST API have been growing in traffic over the last three years. The number of page views has quadrupled between Q2 2009 and Q1 2011 (see Figure 2, top chart). Furthermore, BioPortal users proved to be very loyal and use the service frequently. Almost 45% of them visit the site at least 15 times per month and 18% return to the site 50 times or more (see Figure 2, bottom chart). The use of the REST services has experienced outstanding growth in 2011. The average number of hits per month grew from 3M hits in 2010 to 122M hits in 2011. [6] Our commitment to serve this growing demand is one of the primary motivations to undertake the project of migrating towards a more scalable infrastructure.

[6] We consider the average hits per month in order to be able to compare 12 months from 2010 versus 7 months in 2011.

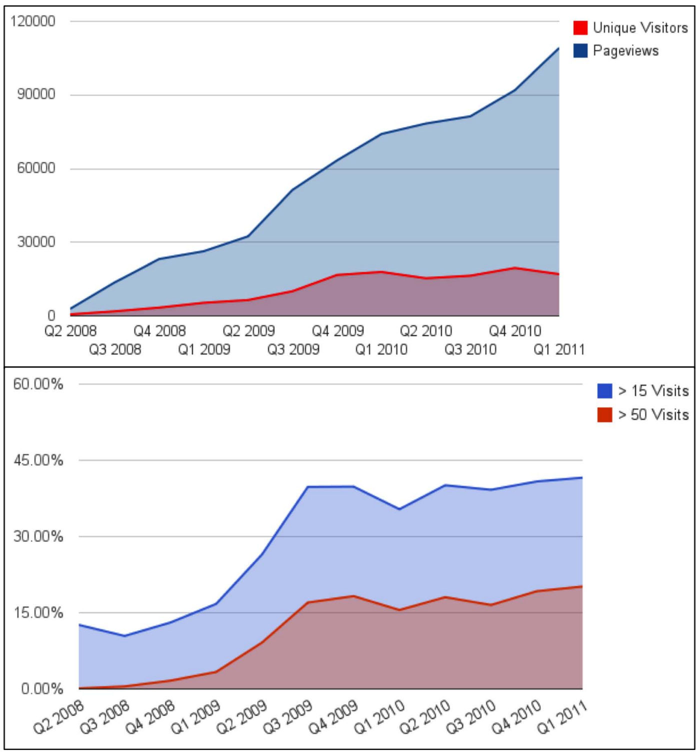


Fig. 2. BioPortal UI Traffic (per quarter, excluding Stanford) (top) and Returning visitors (bottom)

3.2 BioPortal Ontologies

At the time of this writing, there are 284 ontologies in BioPortal. As we mentioned in Section 3.1, the BioPortal ontology repository supports various ontology formats. Table 3 shows the number of ontologies per format. You can see that 90% of the ontologies are either in OBO or OWL formats. Of that 90%, 59% are in OWL and 41% in OBO.

OBO has been the format used for many of biomedical ontologies. Recently, however, there has been a shift and today OWL is the most popular format. This shift is due to the standardization of OWL and the rise of standard tools such as reasoners, query engines and editors. Moreover, there are tools that allow us to transform OBO ontologies into OWL—such as the OWL API [9]—and therefore use RDF serializations to store OBO ontologies in a RDF database. This study covers both OBO and OWL; we leave the rest of the formats for future analyses.

In BioPortal, the metadata associated with each ontology follows the BioPortal Metadata Ontology.[7] This ontology is used to declare projects, authorship,

[7] `http://www.bioontology.org/wiki/index.php/BioPortal_Metadata`

Table 1. Ontology Format Occurrences in BioPortal

Format	**N. Ontologies**
OWL	149
OBO	105
RRF	27
PROTEGE	2
UMLS-RELA	1

categories, coding schemes, users, etc. The details of classes and properties in this ontology are not relevant for this analysis; we will only look into the overall number of triples that we require for an ontology version. The average size of the metadata graph for the 284 ontologies is 38 triples without significant variability (94% of the ontologies are in the range [34,42]). In BioPortal, practically all the metadata are attached to an ontology version. Users can modify nearly every piece of metadata every time they submit a new ontology version. Thus, we will associate the cost of metadata triples to ontology versions rather than to the general concept of ontology.

There are large ontologies, such as the Gene Ontology (GO) with 73K terms in 738K triples, that get updated every day.[8] All of the versions of the Gene Ontology would expand to more than 350M triples. Today, we are not able to provide all the repository functions for every ontology version. Only ontology download is available for old versions. Capabilities such as term search and hierarchy visualization are accessible only for the latest version of each ontology.

For versioning, BioPortal deprecates ontologies using the Apple's "Time Machine" algorithm: keep all versions for the last 7 days; keep one most recent version per week for the last 30 days; keep one most recent version per 30 days from the beginning.[9] This logic applies only to ontologies automatically imported from remote locations. BioPortal keeps all manually submitted versions, because the number of versions of this kind is small.

Currently, BioPortal treats every ontology as a whole, including the materialization of all the `owl:imports`. Thus, if a small ontology imports a large one then the former becomes a large ontology. If we take into account that this problem gets reproduced for every version of that small ontology then we can end up with a very inefficient model that contains millions of duplicated items. Moreover, this model makes it very difficult to retrieve term provenance since many terms are duplicated in graphs where they do not originally belong.

Our hypothesis was that we could optimise the number of quads in the system by using a more granular model where `owl:imports` are not materialized and every ontology graph only contains its own RDF triples without the triples from the `owl:imports` ontologies. This type of granular model would economize the number of triples in our data store and, at the same time, enable provenance

[8] `http://bioportal.bioontology.org/ontologies/1070`
[9] `http://en.wikipedia.org/wiki/Time_Machine_(Mac_OS)`

retrieval. Figure 3 shows the difference between an **import-materialized** model versus an **ontology-per-graph** model.[10]

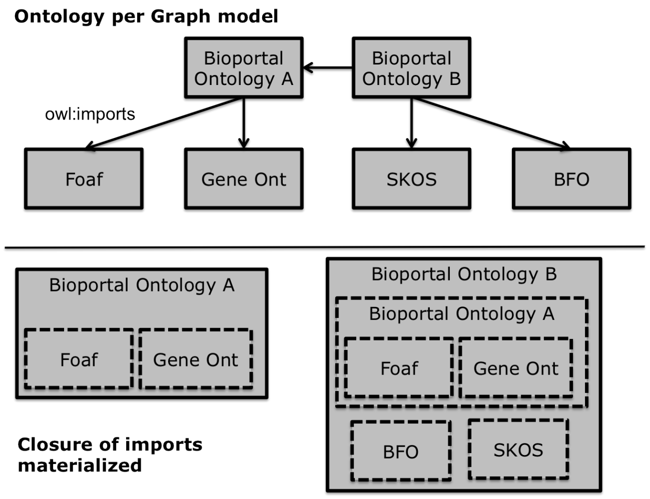


Fig. 3. Models: Per Graph Model vs Materialization of `owl:imports`

The upper part of Figure 3 depicts the import structure maintained by using named-graphs. The bottom part shows how ontologies are incorporated into other ontologies and how this model provokes a redundancy explosion. In order to see quantitatively the level of quad storage space optimisation that we can gain with an ontology-per-graph model we need to analyse the occurrences of `owl:imports` in the BioPortal repository.

In the next section, we analyse the size of BioPortal's data store in triples and the distribution of `owl:imports` in OWL ontologies.

4 Quad Economic Analysis

We estimate the cost, in number of quads, of asserting all the OWL ontologies into a quad store. We also look at the scalability numbers in case that maintaining all versions of each ontology becomes a requirement.

OWL and OBO formats use different mechanisms for import. In order to incorporate ontologies in the OBO format into the import analysis we need to study further the compatibility of OBO imports with OWL imports. Thus, the numbers presented in Figures 4, 5 and 6 include only statistics about imports for ontologies in OWL format.

[10] *Import-materialized* and *ontology-per-graph* are key terms and we shall refer to them in following sections.

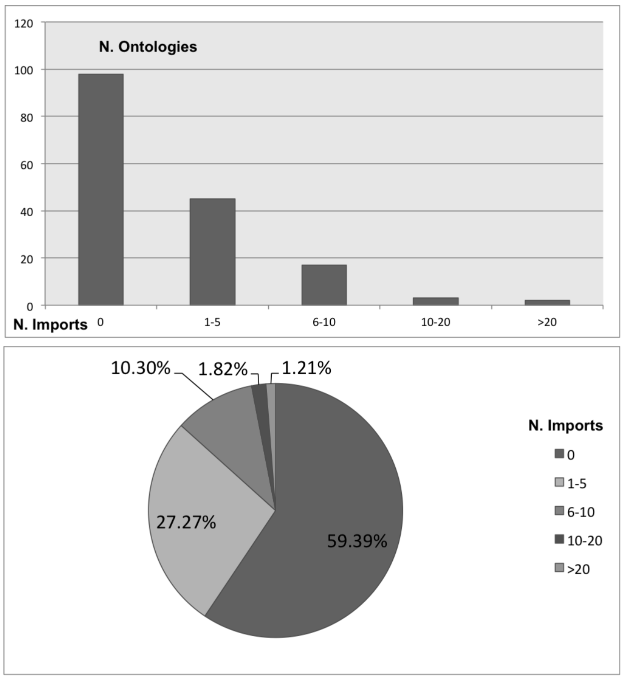


Fig. 4. Number of Imports per Ontology. Absolute numbers at the top, percentages at the bottom.

One of the questions to be answered is the optimisation ratio—in number of triples—when using an *ontology-per-graph* model rather than a *closure-materialized* model (see Section 2). This question is closely related to the number of `owl:imports` in our catalog. The `owl:imports` distribution gives us a mechanism to understand the level of reusability in the data store.

There are 299 ontologies in the import closure of the 149 OWL ontologies in BioPortal (i.e., if we follow all the `owl:imports` links from the 149 ontologies, we will create a set of 299 ontologies). These 299 OWL ontologies contain 303 `owl:imports`, the materialized import closure is a set of 495 `owl:imports`. Not all these 299 ontologies reside in BioPortal as first class citizens. Many of them are just pushed in the system as consequence of an `owl:import`. We do not have metadata for these ontologies. These ontologies could include FOAF or SKOS which are imported by several other ontologies but are not necessarily biomedical ontologies and thus they have not been submitted directly to BioPortal. From the 299 OWL ontologies in the study 165 are ontologies registered in BioPortal.

Figure 4 presents absolute numbers (top chart) and the percentages (bottom chart) of ontologies that have none, one, two or more than two imports. These numbers—from Figure 4—are not really conclusive by themselves, even though we can identify some tendencies. In order to extract conclusions, we also need

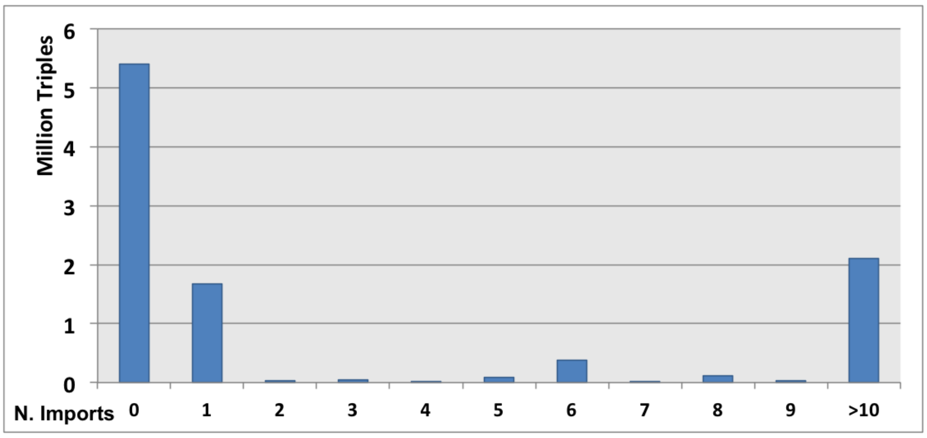


Fig. 5. Closure-Materialized Number of Triples Sum per Number of Imports. Number of imports on X Axis and number of triples on Y Axis

to know the ratio of reused triples. Figure 5 shows the distribution of number of triples, closure materialized, per number of imports. Ontologies that have no imports gather 5.4M triples in the system; ontologies with just one import 1.7M; none of the categories with imports from 2 to 9 reach 0.5M triples. For ontologies with more than 10 imports there is tendency change with a peak of 2.1M triples generated.

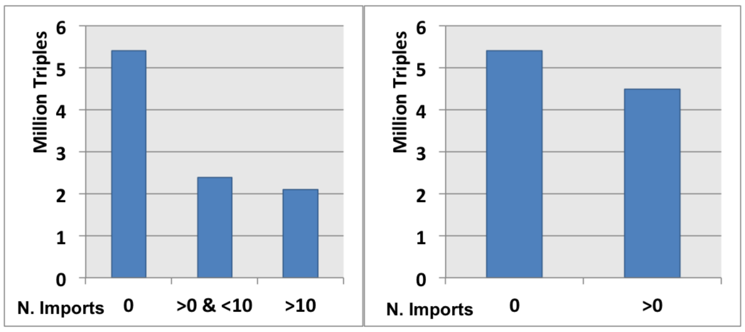


Fig. 6. Closure-Materialized Number of Triples Sum per Number of Imports Grouped. Number of imports on X Axis and number of triples on Y Axis.

Figure 6 groups the data presented in Figure 5 into two different charts. The left-hand side groups the number of triples into three categories: (a) no imports, (b) more than 1 and less than 10 and (c) more than 10. The right-hand side shows the same number grouped in two categories, the ones with imports and the ones without. These two graphs are used in Section 5 for qualitative discussion.

In addition to the previous statistics we also measured various indicators per ontology. Table 2 shows the number of triples without materialized imports (T. Without C.), number with materialized imports (T. With C.), number of imports in closure (N. Import C), depth of tree closure (Depth TI), triples import ratio (I Ratio) and number of versions (NV). For practical reasons we only show a few records in Table 2, but it is worth mentioning that similar numbers have been collected for the 299 OWL ontologies. Note that the last column "NV - Number of Versions" only applies to BioPortal ontologies and not to ontologies fetched from the Web as part of an import. To calculate the ontology import ratio we simply calculate the percentage of triples brought via `owl:imports` against the ones held directly in the ontology, such that:
$import\_ratio = 1 - (T.\ Without\ C./T.\ With\ C.)$

Table 2. Statistical Indicators Sample - Ontologies with more than 10 imports.

Ontology	T. Without C.	T. With C	N. Import C	Depth TI	I Ratio	NV
NIF.owl	6	1,795,951	58	4	99.9%	9
sopharm.owl	1,317	359,087	24	4	99.6%	4
vo.owl	40,804	45,557	16	2	10.2%	92
aero.owl	698	3,981	14	2	82%	8
OntoDM.owl	4,685	8,240	11	4	43%	1
ddi.owl	2,388	5,347	10	2	55%	1

Table 3. Number of Triples and Versions for OBO and OWL Ontologies

	I. Not Mat.	I. Mat.	N. Versions	All V. Not Mat.	All V. Mat.
OWL	9.75M	14.65M	817	41.2M	65.6M
OBO	N/A	5.9M	3,022	N/A	593M
TOTAL	N/A	20.55M	3,839	N/A	658.5M

The last indicative figures are shown in Table 3. This table includes:

– Number of triples without materialized imports (I. Not Mat.).
– Number of triples with materialized imports (I. Mat.).
– Number of versions (N. Versions).
– Number of triples for all the versions of all the ontologies without materialized imports (All V. Not Mat.).
– Number of triples for all the versions of all the ontologies import materialized (All V. Not Mat.).

After having presented several important aspects related to the data size in triples and the `owl:imports` distribution, we shall discuss how they impact the design and scalability of our quad storage in the next section.

5 Discussion

Our initial hypothesis was to assume that we could optimise the number of quads in the system by using a more granular model where we do not materialize `owl:imports` and every ontology graph only contains its own RDF triples without bringing the triples from the `owl:imports` ontologies. In that sense, for the 165 biomedical ontologies in OWL, Figure 4 shows a steep tail distribution where 59% of the ontologies do not make use of `owl:imports` at all. Therefore, they do not reuse any terms from other ontologies. The rest of the ontologies (40.6%) use imports but the majority of them (27.27%) contain a small number of imports, 5 imported ontologies or less. Ontologies with high levels of imports are rare in our catalog, just 3.03% have more than 10 imports and from those only 1.21% have more than 20. These figures let us conclude that the levels of reusability in OWL ontologies for the biomedical domain is low. We do not try to explain why this situation happens, but, arguably, the experts in the biomedical domain have adopted ontologies earlier than many others [13]; if do not often practice ontology reuse, works in other domains have an even longer way to go.

Next, we look more closely at what type of ontologies reuse other ontologies, in order to determine the effects in terms of storage requirements. Because most ontologies are self contained and they do not reuse other ontologies, even in the model where we do not materialize imports, not many ontology graphs will refer to other ontology graphs. However, if the 303 `owl:imports` are placed in the right place, then the *ontology-per-graph* model would optimize the dataset size. In an *ontology-per-graph* model, the ideal situation is to have the majority of the imports in the larger and most frequently updated ontologies. The rationale behind this conclusion is that, not all modules in a new version of an ontology would change. In those cases we only need to assert a new version for the updated modules and modify the metadata. Thus, we look at the relationship between ontology sizes, number of imports and update frequency.

Figures 5 and 6 give us the relationship between ontology size, in number of triples, and the level of modularity, in number of imports. Contrary to what one might expect, the larger the ontology, the fewer imports it has. In other words, larger ontologies that could benefit more from modularity actually are not modular in practice. The exception is the peak for "> 10" category in Figure 5, which has 2.1M triples. This peak is caused by two very modular ontologies that are also large, `NIF` and `sopharm`, with 58 and 24 imports respectively; and 1.7M and 300.5K triples respectively (see Table 2). However, ontologies like these are an exception. Within the 14 largest ontologies in our collection, all of them with more than 100K triples, `NIF` and `sopharm` are the only ontologies with more than 2 imports and with more than 10% of reused import ratio. Thus, we can conclude that the size of the ontology is not directly related to the number of imports.

Next, we look at the absolute numbers for ontology sizes (Table 3). We can calculate the percentage of saved triples for the latest ontology versions if we materialize the imports, $1 - (I.\ Not\ Mat./I.\ Mat.)$. The percentage obtained is 33.4% for the last snapshot and, for all versions, it goes up to 37.2%. These percentages show that, even though the level of re-usability in the OWL ontologies is very low, we can still save more than one third of the storage by moving to an *ontology-per-graph* model. Moreover, that model also provides the requirements to track provenance and implement versioning.

One of the columns in Table 3 also shows the size of the OBO ontologies, but only for the import-materialized case. The last snapshot of the OBO ontologies contains 5.9M triples. However, historically BioPortal has a larger number of versions (3,022) for OBO ontologies than for OWL because they get updated more frequently and also because OBO was the predominant format in the past. This large number of versions makes the total number of triples to be 593M. We hope that, once OBO ontologies use a more explicit import mechanism or their developers switch to OWL, we will be able to save on the storage at a ratio that is similar to the one that we observed for OWL Ontologies.

One of the motivations to maintain an *import-materialized* model is to facilitate SPARQL query construction. There will be cases in which we need to run queries on a set of ontologies —and not over the global graph. In an *import-materialized* model, one ontology is reflected by one graph only, and thus queries targeting specific ontologies are easy to write. In a *ontology-per-graph* model, one ontology is reflected by a closure of tree imports, which can potentially complicate the query construction. It is possible to argue that, depending on the complexity of the `owl:imports` network, it might not be feasible to construct queries with hundreds or thousands of `GRAPH` or `FROM NAMED` clauses. You could even argue that for a very large `owl:imports` network the size of the SPARQL query string could become problematic. Even though all these arguments are based on solid foundations, they would not be valid for BioPortal's case. The statistics collected in Section 4 show that:

1. 59% of the ontologies would be reflected by just one graph, because they do not have imports.
2. The largest import closure contains 58 items, which is far from being the potential cause of a problem. 58 graph clauses to construct an RDF data set should not be a problem for any of the quad stores listed in Section 2.
3. The maximum depth of the import tree is 4 levels, and thus calculating the import closure will never steal many computational cycles. Moreover, the number of imports in the system makes it feasible to pre-cache all of them at the application level. With this approach, the application would not have to go back and forth to the SPARQL endpoint in order to calculate the closures.

To conclude, our analysis shows that, on the one hand, ontology reuse is still far from being the norm, but, on the other, effective reuse is a goal worth pursuing: as ontologies become larger in size, the level of reuse can have significant implications for the scalability of the ontology storage systems.

6 Acknowledgments

This work was supported by the National Center for Biomedical Ontology, under grant U54 HG004028 from the National Institutes of Health.

References

1. Noy, N.F., Shah, N., Dai, B., Dorf, M., Griffith, N., Jonquet, C., Montegut, M., Rubin, D.L., Youn, C., Musen, M.A.: Bioportal: A Web Repository for Biomedical Ontologies and Data Resources. In: International Semantic Web Conference (Posters & Demos) (2008)
2. Bizer, C., Cyganiak, R.: The Trig Syntax. Tech. rep., FU Berlin (7 2007), `http://www.wiwiss.fu-berlin.de/suhl/bizer/TriG/Spec/TriG-20070730/`
3. Broekstra, J., Kampman, A., Harmelen, F.V.: Sesame: A Generic Architecture for Storing and Querying RDF and RDFS. pp. 54–68. Springer (2002)
4. Carroll, J.J., Bizer, C., Hayes, P.J., Stickler, P.: Named Graphs, Provenance and trust. In: WWW. pp. 613–622 (2005)
5. Chein, M., Mugnier, M., Simonet, G.: Nested Graphs: A Graph-Based Knowledge Representation Model with FOL Semantics. In: Proceedings of the 6th International Conference on Knowledge Representation (KR'98. pp. 524–534. Morgan Kaufmann (1998)
6. Cyganiak, R., Harth, A., Hogan, A.: N-Quads: Extending N-Triples with Context. Tech. rep. (2008), `http://sw.deri.org/2008/07/n-quads/`
7. Erling, O., Mikhailov, I.: RDF Support in the Virtuoso DBMS. In: CSSW. pp. 59–68 (2007)
8. Harris, S., Lamb, N., Shadbolt, N.: 4store: The Design and Implementation of a Clustered RDF Store. In: Scalable Semantic Web Knowledge Base Systems - SSWS2009. pp. (p. 94–109) (2009)
9. Horridge, M., Bechhofer, S.: The OWL API: A Java API for OWL Ontologies. Semantic Web 2(1), 11–21 (2011)
10. Noy, N.F., Dorf, M., Griffith, N., Nyulas, C., Musen, M.A.: Harnessing the Power of the Community in a Library of Biomedical Ontologies. In: Workshop on Semantic Web Applications in Scientific Discourse at the 8th International Semantic Web Conference (ISWC 2009). Chantilly, VA (2009)
11. Owens, A., Seaborne, A., Gibbins, N., schraefel, mc: Clustered TDB: A clustered triple store for jena (November 2008), `http://eprints.ecs.soton.ac.uk/16974/`
12. Palma, R., Hartmann, J., Haase, P.: OMV: Ontology Metadata Vocabulary for the Semantic Web. Tech. rep., http://ontoware.org/projects/omv/ (2008)
13. Rubin, D.L., Shah, N.H., Noy, N.F.: Biomedical Ontologies: a Functional Perspective. Briefings in Bioinformatics 9(1), 75–90 (2008)
14. Sheridan, J., Tennison, J.: Linking UK Government Data, pp. 1–4. ACM Press (2010), `http://events.linkeddata.org/ldow2010/papers/ldow2010_paper14.pdf`

CumulusRDF: Linked Data Management on Nested Key-Value Stores

Günter Ladwig and Andreas Harth

Institute AIFB, Karlsruhe Institute of Technology
76128 Karlsruhe, Germany
{guenter.ladwig,harth}@kit.edu

Abstract. Publishers of Linked Data require scalable storage and retrieval infrastructure due to the size of datasets and potentially high rate of lookups on popular sites. In this paper we investigate the feasibility of using a distributed nested key-value store as an underlying storage component for a Linked Data server which provides functionality for serving Linked Data via HTTP lookups and in addition offers single triple pattern lookups. We devise two storage schemes for our CumulusRDF system implemented on Apache Cassandra, an open-source nested key-value store. We compare the schemes on a subset of DBpedia and both synthetic workloads and workloads obtained from DBpedia's access logs. Results on a cluster of up to 8 machines indicate that CumulusRDF is competitive to state-of-the-art distributed RDF stores.

1 Introduction

Linked Data refers to graph-structured data encoded in RDF (Resource Description Framework[1]) and accessible via HTTP (Hypertext Transfer Protocol)[3]. Linked Data leverages the general web architecture for data publishing and has become increasingly popular for exposing data on the web. The Linking Open Data (LOD) cloud [2] lists over 200 datasets and comprises billions of RDF triples. A basic variant for publishing Linked Data is making an RDF file accessible via HTTP, e.g., by putting the file on a standard web server.

Many datasets on the Linked Data web cover descriptions of millions of entities. DBpedia [2], for example, describes 3.5m things with 670m RDF triples, and LinkedGeoData [13] describes around 380m locations with over 2bn RDF triples. Standard file systems are ill-equipped for dealing with these large amounts of files (each description of a thing would amount to one file). Thus, data publishers typically use full-fledged RDF triple stores for exposing their datasets online. Although the systems are in principle capable of processing complex (and hence expensive) queries, the offered query processing services are curtailed to guarantee continuous service[3].

[1] http://www.w3.org/RDF/
[2] http://lod-cloud.net/
[3] Restrictions on query expressivity are common in other large-scale data services, for example in the Google Datastore. See http://code.google.com/appengine/docs/python/datastore/overview.html\#Queries_and_Indexes.

At the same time, there is a trend towards specialised data management systems tailored to specific use cases [16]. Distributed key-value stores, such as Google Bigtable [5], Apache Cassandra [10] or Amazon Dynamo [6], sacrifice functionality for simplicity and scalability. These stores operate on a key-value data model and provide basic key lookup functionality only. Some key-value stores (such as Apache Cassandra and Googles Bigtable) also provide additional layers of key-value pairs, i.e. keys can be nested. We call these stores *nested* key-value stores. As we will see, nesting is crucial to efficiently store RDF data. Joins, the operation underlying complex queries, either have to be performed outside the database or are made redundant by de-normalising the storage schema. Given the different scope, key-value stores can be optimised for the requirements involving web scenarios, for example, high availability and scalability in distributed setups, possibly replicated over geographically dispersed data centres. Common to many key-value stores is that they are designed to work in a distributed setup and provide replication for fail-safety. The assumption is that system failures occur often in large data centres and replication ensures that components can fail and be replaced without impacting operation.

The goal of our work is to investigate the applicability of key-value stores for managing large quantities of Linked Data. We review Linked Data in Section 2 before we present our main contributions:

- We devise two RDF storage schemes on Google BigTable-like (nested) key-value stores supporting Linked Data lookups and atomic triple pattern lookups (Section 3).
- We compare the performance of the two storage schemes implemented on Apache Cassandra [10], using a subset of the DBpedia dataset with a synthetic and a real-world workload obtained from DBpedia's access log (Section 4).

We discuss related work in Section 5 and conclude with a summary and an outlook to future work in Section 6.

2 Linked Data

We first review basic Linked Data concepts. RDF is a data format for graph-structured data encoded as ($subject$, $predicate$,$object$) triples. These triples are composed of unique identifiers (URI references), literals (e.g., strings or other data values), and local identifiers called blank nodes as follows:

Definition 1. *(RDF Triple, RDF Term, RDF Graph) Given a set of URI references $\mathcal{U}$, a set of blank nodes $\mathcal{B}$, and a set of literals $\mathcal{L}$, a triple (s, p, o)* $\in (\mathcal{U} \cup \mathcal{B}) \times \mathcal{U} \times (\mathcal{U} \cup \mathcal{B} \cup \mathcal{L})$ *is called an* RDF triple. *We call elements of* $\mathcal{U} \cup \mathcal{B} \cup \mathcal{L}$ RDF terms. *Sets of RDF triples are called* RDF graphs.

The notion of graph stems from the fact that RDF triples may be viewed as labelled edges connecting subjects and objects.

There are various serialisation formats for RDF, for example normative RDF/XML and Notation3 (N3)[4]. We use the human-readable N3 in this paper. In N3, namespaces can be introduced to abbreviate full URIs as $namespaceprefix$:$localname$, e.g., a parser expands `foaf:name` in combination with the syntactic definition of the namespace prefix to `http://xmlns.com/foaf/0.1/name`.

The N-Triples syntax[4] is a subset of N3 (without namespaces declarations) where RDF triples are written as whitespace separated RDF terms with a trailing '.'. Brackets (<>) denote URIs and quotes (`""`) denote literals. Blank node identifiers start with '`_:`'.

Definition 2. *(Triple Pattern) A* triple pattern *is an RDF triple that may contain variables (prefixed with '?') instead of RDF terms in any position.*

Triple patterns can be used to express basic RDF queries. For example, a triple pattern `?s foaf:name ?n .` matches all triples with a `foaf:name` predicate. In total there are eight possible patterns for RDF triples (where s,p,o denote a constant and ? a variable): (spo), $(sp?)$, $(?po)$, $(s?o)$, $(?p?)$, $(s??)$, $(??o)$, $(???)$. Queries in SPARQL[5] can be translated to query plans involving triple pattern lookups. However, we exclude full SPARQL query processing capabilities as our goal is to provide scalable Linked Data access.

Linked Data refers to principles [3] that mandate that things (entities or concepts) are identified via HTTP URIs. When dereferencing the URI t denoting a thing, the web server shall return an RDF graph g describing t; g should contain links to other URIs to enable decentralised discovery of resources.

There is a correspondence (in URI syntax or via HTTP redirects) between t and the information resource s which represents that data source of a graph. For an example of a syntactic correspondence consider the resource URI `http://harth.org/andreas/foaf#ah` which is described at the information resource URI `http://harth.org/andreas/foaf`. The former denotes a person and the latter denotes the physical data source which contains the RDF graph. HTTP redirects are another way for ensuring the correspondence; a user agent performs a lookup on URI t, and the web server answers with a new location for the data source s. User agents specify their preferred content format via HTTP's `Accept` header for content negotiation. For example, a lookup on `http://xmlns.com/foaf/0.1/knows` redirects to a HTML file `http://xmlns.com/foaf/spec/index.html` when dereferenced via a Web browser, or to an RDF/XML file `http://xmlns.com/foaf/spec/index.rdf` when accessed via an RDF-aware client.

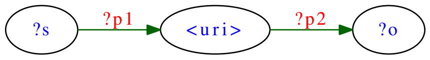


Fig. 1. Illustration of object-subject lookups with `<uri>` as constant.

What should be included in the graph g is only lightly specified[6]. A method which can be regarded as current best practice is employed by DBpedia, which, given a URI,

[4] `http://www.w3.org/TR/rdf-testcases/#ntriples`
[5] `http://www.w3.org/TR/rdf-sparql-query/`
[6] [3] advises to "provide useful information"

returns (i) all triples with the given URI (or its associated information URI) as subject and (ii) some triples with the given URI as object are returned (Figure 1). Such a lookup translates to a union of two triple pattern lookups (or four when including the associated information URI lookups): (i) a $(s??)$ lookup on the SPO index and (ii) a $(??o)$ lookup.

Another method called "Concise Bounded Description" [15] proposes to return all triples which contain the given URI on the subject position with provisions to ensure that connected blank nodes and reified statements are returned. How to include the handling of blank nodes and reified statements in our method is subject to further work.

3 Storage Layouts

In the following we describe indexing schemes for RDF triples on top of nested key-value stores. The goals for the index scheme are:

- To cover all six possible RDF triple pattern with indices to allow for answering single triple patterns directly from the index. In other words, we aim to provide a complete index on RDF triples [8].
- To employ prefix lookups so that one index covers multiple patterns; such a scheme reduces the number of indices and thus index maintenance time and amount of space required.

Ultimately, the index should efficiently support basic triple pattern lookups and (as a union of those) Linked Data lookups which we evaluate in Section 4.

3.1 Nested Key-Value Storage Model

Since Google's paper on Bigtable [5] in 2006, a number of systems have been developed that mimic Bigtable's mode of operation. Bigtable pioneered the use of the key-value model in distributed storage systems and inspired systems such as Amazon's Dynamo [6] and Apache's Cassandra [10]. We illustrate the model in Figure 2.

We use the notation { `key:value` } to denote a key-value pair. We denote concatenated element with, e.g., `sp`, constant entries with `'constant'` and an empty entry with `-`. The storage model thus looks like the following:

```
{ row key : { column key : value } }
```

Systems may use different strategies to distribute data and organise lookups. Cassandra uses a distributed hashtable structure for network access: storage nodes receive a hash value; row keys are hashed and the row is stored on a node which is closest in numerical space to the row key's hash value. Only hash lookups thus can be performed. Although there is the possibility for configuring an order preserving partitioner for row keys, we dismiss that option, as skewed data distribution easily can lead to hot-spots among the storage nodes.

Another restriction is that entire rows are stored on a single storage node - data with the same row key always ends up on the same machine. Columns, on the other hand, are stored in order of their column keys, which means that the index allows for range scans and therefore prefix lookups.

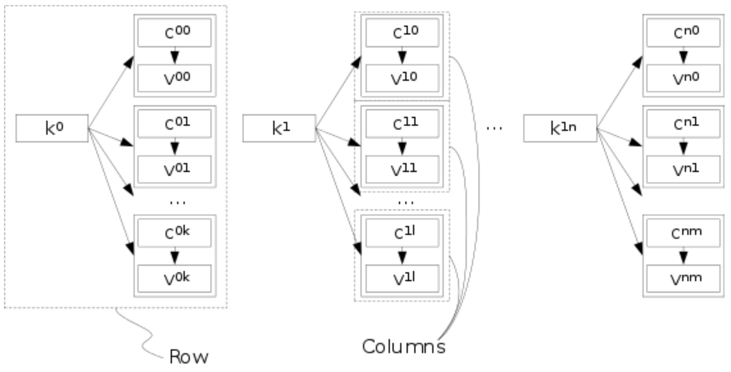


Fig. 2. Illustration of key-value storage model comprising rows and columns. A lookup on the row key (k) returns columns, on which a lookup on column keys (c) returns values (v).

Please note that keys have to be unique; both row keys and column keys can only exist in the index once, which has implications on how we can store RDF triples.

Cassandra has two further features which our index schemes use: supercolumns and secondary indices. Supercolumns add an additional layer of key-value pairs; a storage model with supercolumns looks like the following:

```
{ row key : { supercolumn key : { column key : value }}}
```

Supercolumns can be either stored based on the hash value of the supercolumn key or in sorted order. In addition, supercolumns can be further nested.

Secondary indices, another feature of Cassandra we use for one of the storage layouts, allow to map column values to row keys:

```
{ value : row key }
```

Applications could use the regular key-value layout to also index values to row keys, however, secondary indices are "shortcuts" for such indices. In addition, secondary indices are built in the background without requiring additional maintenance code.

For an introduction to more Cassandra-specific notation we refer the interested reader to the Cassandra web site[7].

We identify two feasible storage layouts: and one based on supercolumns ("Hierarchical Layout"), and one based on a standard key-value storage model ("Flat Layout"), but requiring a secondary index given restrictions in Cassandra.

3.2 Hierarchical Layout

Our first layout scheme builds on supercolumns.

[7] `http://wiki.apache.org/cassandra/DataModel`

The first index is constructed by inserting (s, p, o) triples directly into a supercolumn three-way index, with each RDF term occupying key, supercolumn and column positions respectively, and an empty value. We refer to that index as SPO, which looks like the following:

```
{ s : { p : { o : - } }
```

For each unique `s` as row key, there are multiple supercolumns, one for each unique `p`. For each unique `p` as supercolumn key, there are multiple columns, one for each `o` as column key. The column value is left empty.

Given that layout, we can perform (hash-base) lookups on `s`, (sorted) lookups on `p` and (sorted) lookups on `o`.

We construct the POS and OSP indices analogously. We use three indices to satisfy all six RDF triple patterns as listed in Table 1. The patterns (spo) and $(???)$ can be satisfied by any of the three indices.

Triple Pattern	Index
(spo)	SPO, POS, OSP
$(sp?)$	SPO
$(?po)$	POS
$(s?o)$	OSP
$(?p?)$	POS
$(s??)$	SPO
$(??o)$	OSP
$(???)$	SPO, POS, OSP

Table 1. Triple patterns and respective usable indices to satisfy triple pattern

3.3 Flat Layout

We base our second storage layout on the standard key-value data model. As columns are stored in a sorted fashion, we can perform range scans and therefore prefix lookups on column keys. We thus store (s, p, o) triples as

```
{ s : { po : - } }
```

where s occupies the row-key position, p the column-key position and o the value position.

We use a concatenated `po` as column key as column keys have to be unique. Consider using `{ s : { p : { o } } }` as layout, which `p` as column key and `o` as value. In RDF, the same predicate can be attached to a subject multiple times, which violates the column key uniqueness requirement. We would like to delegate all low-level index management to Cassandra, hence we do not consider maintaining lists of `o`'s manually.

The SPO index satisfies the triple patterns $(s??)$, $(sp?)$ and (spo) with direct lookups. To perform a $(sp?)$ lookup on the SPO index, we look for the column keys matching p (via prefix lookup on the column key `po`) on the row with row key s.

We also need to cover the other triple patterns. Thus, two more indices are constructed by inserting (p, o, s) and (o, s, p) re-orderings of triples, leading to POS and OSP indices. In other words, to store the SPO, POS and OSP indices in a key-value store, we create a column family for each index and insert each triple three times: once into each column family in different order of the RDF terms.

There is a complication with the POS index though, because RDF data is skewed: many triples may share the same predicate [9]. A typical case are triples with `rdf:type` as predicate, which may represent a significant fraction of the entire dataset. Thus, having P as the row key will result in a very uneven distribution with a few very large rows. As Cassandra is not able to split rows among several nodes, large rows lead to problems (at least a very skewed distribution, out of memory exceptions in the worst case). The row size may exceed node capacity; the uneven distribution makes load balancing problematic. We note that skewed distribution may also occur on other heavily used predicates (and possibly objects, for example in case of heavily used class URIs), depending on the dataset. For an in-depth discussion of uneven distribution in RDF datasets see [7].

To alleviate the issue, we take advantage of Cassandra's secondary indexes.

First, we use PO (and not P) as a row key for the POS index which results in smaller rows and also better distribution, as less triples share predicate and object than just the predicate. The index thus looks like the following:

```
{ po : { s : - } }
```

In each row there are is also a special column 'p' which has P as value:

```
{ po : { 'p' : p } }
```

Second, we use a secondary index which maps column values to row keys, resulting in an index which allows for retrieving all PO row keys for a given P. Thus, to perform a $(?p?)$ lookup, the secondary index is used to retrieve all row keys that contain that property. For $(?po)$ lookups, the corresponding row is simply retrieved.

In effect, we achieve better distribution at the cost of a secondary index and an additional redirection for $(?p?)$ lookups.

4 Evaluation

We now describe the experimental setup and the results of the experiments. We perform two set of experiments to evaluate our approach:

- first, we measure triple pattern lookups to compare the hierarchical (labelled Hier) and the flat storage (labelled Flat) layout; and
- second, we select the best storage layout to examine the influence of output format on overall performance.

4.1 Setting

We conducted all benchmarks on four nodes in a virtualised infrastructure (provided via OpenNebula[8], a cloud infrastructure management system similar to EC2). Each virtualised node has 2 CPUs, 4 GB of main memory and 40 GB of disk space connected to a networked storage server (SAN) via GPFS[9]. The nodes run Ubuntu Linux. We used version 1.6 of Sun's Java Virtual Machine and version 0.8.1 of Cassandra. The Cassandra heap was set to 2GB, leaving the rest for operating system buffers.. We deactivated the Cassandra row cache and set the key cache to 100k. The Tomcat[10] server was run on one of the cluster nodes to implement the HTTP interface.

Apache JMeter[11] was used to simulate multiple concurrent clients.

4.2 Dataset and Queries

For the evaluation we used the DBpedia 3.6 dataset[12] excluding pagelinks; characteristics of that DBpedia subset are listed in Table 2. We obtained DBpedia query logs from 2009-06-30 to 2009-10-25 consisting of 87,203,310 log entries.

Name	Value
distinct triples	120,436,315
distinct subject	18,324,688
distinct predicates	42,004
distinct objects	40,531,020

Table 2. Dataset characteristics

We constructed two query sets:

– for testing single triple pattern lookups, we sampled 1 million S, SP, SPO, SO, O patterns from the dataset;
– for Linked Data lookups we randomly selected 2 million resource lookup log entries from the DBpedia logs (which, due to duplicate lookups included in the logs, amount to 1,241,812 unique lookups).

The triple pattern lookups were executed on CumulusRDF using its HTTP interface. The output of such a lookup is the set of triples matching the pattern, serialised as RDF/XML.

The output of a Linked Data lookup on URI u is the union of two triple pattern lookups: 1) all triples matching pattern $(u??)$ and 2) a maximum of 10k triples matching $(??u)$. The number of results for object patterns is limited in order to deal with the

[8] `http://opennebula.org/`
[9] `http://www-03.ibm.com/systems/software/gpfs/`
[10] `http://tomcat.apache.org/`
[11] `http://jakarta.apache.org/jmeter/`
[12] `http://wiki.dbpedia.org/Downloads36`

skewed distribution of objects, which may lead to very large result sets. A similar limitation is used by the official DBpedia server (where a maximum of 2k triples in total are returned).

4.3 Results: Storage Layout

Table 3 shows the size and distribution of the indices on the four machines. The row size of a Cassandra column family is the amount of data stored for a single row-key. For example, for the OSP index, this is the number of triples that share a particular object.

Index	Node 1	Node 2	Node 3	Node 4	Std.Dev.	Max. Row
SPO Hier	4.41	4.40	4.41	4.41	0.01	0.0002
SPO Flat	4.36	4.36	4.36	4.36	0.00	0.0004
OSP Hier	5.86	6.00	5.75	6.96	0.56	1.16
OSP Flat	5.66	5.77	5.54	6.61	0.49	0.96
POS Hier	4.43	3.68	4.69	1.08	1.65	2.40
POS Sec	7.35	7.43	7.38	8.05	0.33	0.56
POS Flat	-	-	-	-	-	-

Table 3. Index size per node in GB. POS Flat is not feasible due to skewed distribution of predicates. The size for POS Sec includes the secondary index.

The load distribution shows several typical characteristics of RDF data and directly relates to the dataset statistics from Table 2. The small maximum row size for SPO shows that there are very few triples that share a subject. The large maximum row size of the OSP index indicates that there are a few objects that appear in a large amount of triples, i.e., the distribution is much more skewed (this is usually due the small number of classes that appear as objects in the many rdf:type triples).

Here, we can also see the difference between the hierarchical and secondary POS indices. The hierarchical POS index only uses the predicate as key, leading to a very skewed distribution (indicated by the maximum row size) and a very uneven load distribution among the cluster nodes (indicated by the high standard deviation). The POS index using a secondary index fares much better as it uses the predicate and object as row key. This validates the choice of using secondary index for POS over the hierarchical layout.

4.4 Results: Queries

The performance evaluation consists of two parts: first, we use triple pattern lookups to compare two CumulusRDF storage layouts and, second, we examine the influence of output formats using Linked Data lookups.

Triple Pattern Lookups Fig. 3 shows the average number of requests/s for the two CumulusRDF storage layouts (flat and hierarchical) for 2, 4, 8, 16 and 32 concurrent

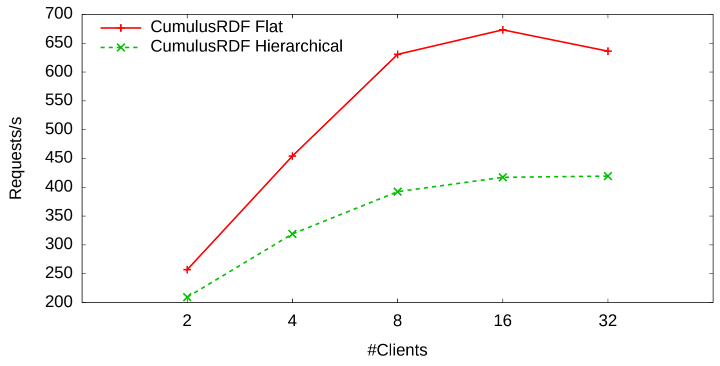


Fig. 3. Requests per second for triple pattern lookups with varying number of clients.

clients. Overall, the flat layout outperforms the hierarchical layout. For 8 concurrent clients, the flat layout delivers 1.6 times as many requests per second as the hierarchical layout. For both layouts the number of requests per second does not increase for more than 8 concurrent clients, indicating a performance limit. This may be due to the bottleneck of using a single HTTP server (as we will see in the next section).

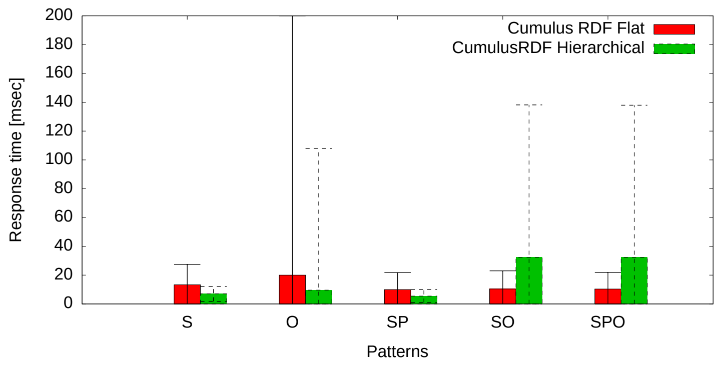


Fig. 4. Average response times per triple pattern (for 8 concurrent clients). Please note we omitted patterns involving the POS index (P and PO). Error bars indicate the standard deviation.

Fig. 4 shows the average response times from the same experiment, broken down by pattern type (S, O, SP, SO and SPO). This shows the differences between the two CumulusRDF storage layouts. While the hierarchical layout performs better for S, O and SP patterns, it performs worse for SO and SPO patterns. The worse performance for SO and SPO is probably due to inefficiencies of Cassandra super columns. For example, Cassandra is only able to de-serialise super columns as a whole, which means for SPO lookups *all* triples matching a particular subject and predicate are loaded. This is not the case for the flat layout (which does not use super columns): here, Cassandra is able to load only a single column, leading to much better response times.

Linked Data Lookups Fig. 5 shows the average number of requests per second of the flat layout of CumulusRDF with two different output formats: RDF/XML and N-Triples. As CumulusRDF stores RDF data in N3 notation the N-Triples output is cheaper to generate (for example, it does not require the escaping that RDF/XML does). This is reflected in the higher number of lookups per second that were performed using N-Triples output. The difference between the two output format is more pronounced for a higher number of concurrent clients: for 4 clients N-Triples is 12% faster than RDF/XML, whereas for 8 clients the difference is 26%. This indicates that the performance is mainly limited by the Tomcat server, whose webapp performs the output formatting.

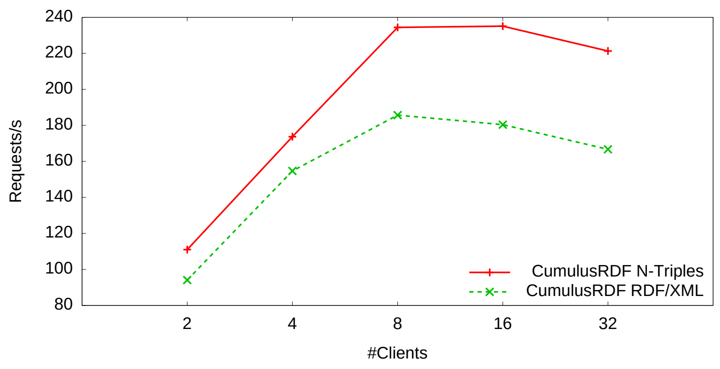


Fig. 5. Requests per second Linked Data lookups (based on flat storage layout) with varying number of clients (including measure of effect of serialisation).

4.5 Conclusion

Overall, the evaluation shows that the flat layout outperforms the hierarchical layout. We also see that Apache Cassandra is a suitable storage solution for RDF, as indicated by the preliminary results. As the different layouts of CumulusRDF perform differently for different types of lookups, the choice of storage layout should be made considering the expected workload. From the experiments we can also see that the output format also has a large impact on performance. It may be beneficial to store the RDF data in an already escaped form if mainly RDF/XML output is desired.

5 Related Work

A number of RDF indexing schemes have been devised. YARS [8] described the idea of "complete" indices on RDF with context (quads)[13], with an index for each possible triple pattern. Similar indices are used by RDF-3X [11] and Hexastore [17]. All of these systems are built to work on single machines. In our work we use complete indices for RDF triples, implemented in three indices over a distributed key-value store.

Abadi et al. introduced an indexing scheme on C-Store for RDF [1] called "vertical partitioning", however, with only an index on the predicate while trying to optimise access on subject or object via sorting of rows. The approach is sub-optimal on datasets with many predicates as subject or object lookups without specified predicate require a lookup on each of the predicate indices. For a follow-up discussion on C-Store we refer the interested reader to [12].

The skewed distribution for predicates on RDF datasets and the associated issues in a distributed storage setting have been noted in [9].

Stratustore is an RDF store implemented over Amazon's SimpleDB [14]. In contrast to Stratusstore, which only makes use of one index on subject (and similarly the RDF adaptor for Ruby[14]), we index all triple patterns. In addition, Cassandra has to be set up on a cluster while SimpleDB is a service offering, accessible via defined interfaces.

6 Conclusion and Future Work

We have presented and evaluated two index regimen for RDF on nested key-value stores to support Linked Data lookups and basic triple pattern lookups. The flat indexing scheme has given best results; in general, our implementation of the flat indexing scheme on Apache Cassandra can be seen as a viable alternative to full-fledged RDF stores in scenarios where large amounts of small lookups are required. We are working on packaging the current version of CumulusRDF for publication as open source at `http://code.google.com/p/cumulusrdf/`. We would like to add functionality for automatically generating and maintaining dataset statistics to aid dataset discovery or distributed query processors. Future experiments include measures on workloads involving inserts and updates.

Acknowledgements

We thank Knud Möller and Chris Bizer for providing access to the dbpedia.org logs and Aidan Hogan and Andreas Wagner for comments. We gratefully acknowledge the support of the Open Cirrus team at KIT's Steinbuch Centre for Computing which provided the necessary infrastructure for conducting the experiments. The authors acknowledge the support of the European Commission's Seventh Framework Programme FP7/2007-2013 (PlanetData, Grant 257641).

[13] See also `http://kowari.sourceforge.net/`.
[14] `http://rdf.rubyforge.org/cassandra/`

References

1. D. J. Abadi, A. Marcus, S. R. Madden, and K. Hollenbach. Scalable semantic web data management using vertical partitioning. In *Proceedings of the 33rd International Conference on Very Large Data Bases*, VLDB '07, pages 411–422. VLDB Endowment, 2007.
2. S. Auer, C. Bizer, G. Kobilarov, J. Lehmann, R. Cyganiak, and Z. G. Ives. Dbpedia: A nucleus for a web of open data. In *Proceedings of the 6th International Semantic Web Conference*, ISWC '07, pages 722–735, 2007.
3. T. Berners-Lee. Linked Data - Design Issues, year = 2006. `http://www.w3.org/DesignIssues/LinkedData`.
4. T. Berners-Lee. Notation3 (N3) A readable RDF syntax year = 2000. `http://www.w3.org/DesignIssues/Notation3`.
5. F. Chang, J. Dean, S. Ghemawat, W. C. Hsieh, D. A. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. E. Gruber. Bigtable: a distributed storage system for structured data. In *Proceedings of the 7th USENIX Symposium on Operating Systems Design and Implementation*, OSDI '06, pages 15–15, 2006.
6. G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels. Dynamo: Amazon's highly available key-value store. In *Proceedings of 21st ACM SIGOPS Symposium on Operating Systems Principles*, SOSP '07, pages 205–220. ACM, 2007.
7. S. Duan, A. Kementsietsidis, K. Srinivas, and O. 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. ACM, 2011.
8. A. Harth and S. Decker. Optimized Index Structures for Querying RDF from the Web. In *Proceedings of the 3rd Latin American Web Congress*, pages 71–80. IEEE Computer Society, 2005.
9. A. Harth, J. Umbrich, A. Hogan, and S. Decker. Yars2: A federated repository for querying graph structured data from the web. In *6th International Semantic Web Conference*, ISWC '07, pages 211–224, 2007.
10. A. Lakshman and P. Malik. Cassandra: a structured storage system on a p2p network. In *Proceedings of the 21st Annual Symposium on Parallelism in Algorithms and Architectures*, SPAA '09, pages 47–47. ACM, 2009.
11. T. Neumann and G. Weikum. RDF-3X: a RISC-style Engine for RDF. *Proceedings of the VLDB Endowment*, 1(1):647–659, 2008.
12. L. Sidirourgos, R. Goncalves, M. Kersten, N. Nes, and S. Manegold. Column-store support for rdf data management: not all swans are white. *Proceedings of the VLDB Endowment*, 1(2):1553–1563, 2008.
13. C. Stadler, J. Lehmann, K. Höffner, and S. Auer. Linkedgeodata: A core for a web of spatial open data, 2011. Under review, preliminary version at `http://www.semantic-web-journal.net/sites/default/files/swj173_1.pdf`.
14. R. Stein and V. Zacharias. Rdf on cloud number nine. In *Workshop on NeFoRS: New Forms of Reasoning for the Semantic Web: Scalable & Dynamic*, 2010.
15. P. Stickler. CBD - concise bounded description, June 2005. W3C Member Submission, `http://www.w3.org/Submission/CBD/`.
16. M. Stonebraker and U. Cetintemel. "one size fits all": An idea whose time has come and gone. In *Proceedings of the 21st International Conference on Data Engineering*, ICDE '05, pages 2–11. IEEE Computer Society, 2005.
17. C. Weiss, P. Karras, and A. Bernstein. Hexastore: sextuple indexing for semantic web data management. *Proceedings of the VLDB Endowment*, 1(1):1008–1019, 2008.

Optimizing Unbound-property Queries to RDF Views of Relational Databases

Silvia Stefanova, Tore Risch

Uppsala University, Department of Informational Technology
Box 337, SE-751 05 Uppsala, Sweden
{Silvia.Stefanova, Tore.Risch}@it.uu.se

Abstract. SAQ (Semantic Archive and Query) is a system for querying and long-term preservation of relational data in terms of RDF. In SAQ relational data in a back-end DBMS is exposed as an RDF view, called the *RD-view*. SAQ can process arbitrary SPARQL queries to the RD-view. In addition long-term preservation as RDF of selected parts of a relational database is specified by SPARQL queries to the RD-view. Such queries usually select sets of RDF properties and thus in the query definition a property p is unknown. We call such queries *unbound-property queries*. This class of queries is also present in the SPARQL benchmarks. We optimize unbound-property queries by introducing a query transformation algorithm called *Group Common Terms, GCT*. It pulls out from a DNF normalized query those common terms that can be translated to SQL predicates accessing the relational database. Our experiments using the Berlin SPARQL benchmark show that GCT improves substantially the query execution time to a back-end commercial relational DBMS for both selective and unselective unbound-property queries. We compared the performance of our approach with the performance of other systems processing SPARQL queries over views of relational databases and showed that GCT improves scalability compared to the approaches used by the other systems.

Keywords: SPARQL queries, RDF views of relational databases, query optimization, query rewrites, unbound property queries

1 Introduction

Semantic Web technology and, in particular, RDF and RDFS seem promising for both search and long-term preservation of any kind of data including data currently stored in relational databases. In order to investigate the use of RDF for both search and archival of existing relational databases we have developed the SAQ (Semantic Archive and Query) system. In SAQ relational data in a back-end DBMS is exposed as RDF by a view, called the *RD-view,* represented in a Datalog dialect. The RD-view is automatically generated by accessing the database schema. SAQ can process arbitrary SPARQL queries to the RD-view. Long-term preservation as RDF of the contents of selected parts of a relational database is specified by SPARQL queries to the RD-view.

In the RD-view tables are represented as RDFS classes and attributes as RDF properties. Each data value in the relational database is viewed as a triple *(s, p, v)*, where the subject *s* is a URI identifying a row in a relational table, *p* is an RDF property representing the column (i.e. attribute) where a value is stored, and *v* is the data value of the attribute for the row. In SAQ a SPARQL query to the RD-view is transformed into an execution plan containing SQL calls to the back-end relational DBMS followed by post-processing.

Queries to archive database contents typically select sets of attributes of tables to archive. This corresponds to selecting sets of RDF properties in the RD-view to be archived, for example all properties of the class representing the table *offer*. Therefore, in such SPARQL queries a property *p* in some triple pattern is not known. We call such queries *unbound-property queries*. Moreover, unbound-property queries are also present in SPARQL benchmarks. For example, in the SP2Bench benchmark [18] queries *Q9* and *Q10* are unbound-property queries, and in the Berlin SPARQL benchmark [2] *Query 9* and *Query 11* are unbound-property queries, while *Query 9* is a DESCRIBE query that can be expressed as an unbound-property query. Since *p* is unknown in unbound-property queries, the translation from SPARQL to SQL is not trivial and can "easily result in large unions" of sub-queries [7] and therefore "using of variables in the predicate position is discouraged" [8].

In this paper we present a novel approach for optimizing unbound-property queries by implementing a predicate rewrite rule called *group common terms (GCT)*. GCT is shown to substantially improve SPARQL query execution time. It partially denormalizes disjunctive normal form (DNF) predicates to form query fragments doing select-project-joins over the back-end relational tables. The reason for the performance improvement is that GCT generates execution plans that access data in row-order, which is substantially more efficient to process than without GCT, where data is accessed column-wise. In the performance section it will be shown that our approach improves query performance compared to a naïve approach without using GCT. Furthermore, we show that SAQ with GCT executes unbound-property queries substantially faster than other systems able to process SPARQL queries to views of relational databases [4][8]. By investigating the SQL queries emitted by the other systems, we show that they do not employ query transformations similar to GCT.

The rest of this paper is organized as follows. First, in Section two unbound-property queries are defined and exemplified. In Section three the architecture of the SAQ system is presented, the RD-view is defined, and the steps of the query processing in SAQ are explained. In Section four the GCT algorithm is presented. Section five analyzes the performance of SAQ for unbound-property queries and compares it with related systems. Section six describes related work and finally Section seven summarizes.

2 Unbound-property Queries

A *bound-property triple pattern* is a SPARQL triple pattern *(s, P, v)* where the property *P* is a URI representing an RDF property, e.g. *(?s1* *saq:product#review/person ?s.)*. For SPARQL queries to an RD-view the property must match a URI representing a relational column, otherwise the result is empty.

An *unbound-property triple pattern* is a triple pattern *(s, u, v)* where *u* is a variable, e.g. *(?s ?p ?v)* or *(%offerXYZ% ?property ?hasValue)*.

A *bound-property query* is a SPARQL query having only bound-property triple patterns. An *unbound-property query* is a SPARQL query having one or several unbound-property triple patterns. Finally, a *simple unbound-property query* is an unbound-property query with a single unbound-property pattern.

Fig. 1 shows a small relational database *product* having three tables *offer*, *person,* and *review*. The tables are parts of the relational database generated by the Berlin SPARQL benchmark data generator [1]. In the example, they are populated with two offers, two persons, and a review made by each of them. In the scalability experiments later, we use the full Berlin benchmark relational database with the tables *product*, *offer*, *person*, *producer*, *productfeature*, *productfeatureproduct*, *producttype*, *producttypeproduct*, *review*, and *vendor*.

The columns *onr*, *nr,* and *rnr* are the primary keys in the tables, while the column *person* in table *review* references the column *nr* in table *person* as foreign key.

Table *offer*

onr	**price**	**deliveryDays**
5	854.18	3
7	440.9	5

Table *review*

rnr	**person**	**reviewDate**
10	1	2007-09-16
166	8	2006-05-12

Table *person*

nr	**name**	**country**	**publisher**
1	Caryn	KR	9
8	Linda-Nada	AT	3

Fig. 1. *product* database

The following queries are examples of SPARQL queries to the RD-view for preservation and search of the *product* database. They represent different kinds of unbound-property queries with varying selectivity.

Query Q1

```
SELECT ?s ?p ?v
FROM <product>
WHERE {?s rdf:type %offerType%.
       ?s ?p        ?v        }
```
(1)

Query Q2

```
SELECT ?property ?hasValue ?isvalueOf
FROM <product>
WHERE {
        {%offerXYZ% ?property ?hasValue }
        UNION
        {?isValueOf ?property %offerXYZ%} }
```
(2)

Query Q3	(3)
```
SELECT ?s ?p ?v
FROM <product>
WHERE{ ?s  saq:product#person/name %nameXYZ%.
       ?s  ?p                         ?v       }
```
Query Q4	(4)
```
SELECT ?s ?p ?v
FROM <product>
WHERE { ?s1 saq:product#review/person  ?s  .
        ?s  saq:product#person/country 'JP'.
        ?s  ?p                          ?v }
```

In the queries *<product>* denotes the URI for the RD-view of the database product.

Q1, Q3, and *Q4* are simple unbound-property queries, while *Q2* is a union of simple unbound-property queries.

Query *Q1* converts the entire table *offer* to RDF triples. *%offerType%* is the URI associated with table *offer*. *Q1* is a very unselective unbound-property query.

Q2 is a highly selective query that retrieves all information about an offer *%offerXYZ%*, i.e. a single row from the table *offer*. *Q2* is called 'Query 11' in the Berlin SPARQL Benchmark [1][2]. *Q2* searches for both the properties and the inverse properties of an explicitly given offer.

Q3 retrieves all the properties of a person for a given name *%nameXYZ%*. Like *Q2* query *Q3* is highly selective, it selects one row from the table *person*. The difference from *Q2* is that the subject of the unbound-property is not explicitly specified, so the row in the relational database cannot be identified by the URI as in *Q2*.

Query *Q4* retrieves all properties of the 10% of all reviewers coming from country 'JP'. Unlike *Q3* the name of a described person is not explicitly specified, but are retrieved by joining the tables *person* and *review*. It is an unbound-property query with a join. *Q4* is more selective than *Q1* but much less selective than *Q2* and *Q3* since it retrieves more rows when the database grows.

It is shown in section five that the performance of the unbound-property queries *Q1…Q4* is substantially improved by applying the proposed GCT rule. Queries like *Q2* defined with a 'UNION' clause are handled as a union of several non-disjunctive queries, where GCT is applied on each sub-query in the union. The unusual case of unbound-property queries with several unbound-property patterns in a conjunction are also handled by SAQ, but are outside the scope of this paper. The processing of bound-property queries to an RD-view is also outside the scope and was described in [15][16].

3 SAQ

The architecture of the SAQ system is presented in Fig. 2. The *source DB* is the underlying relational database, which can be queried and preserved by SAQ. The *RD-view generator* generates the *RD-view* over the source DB from the database schema

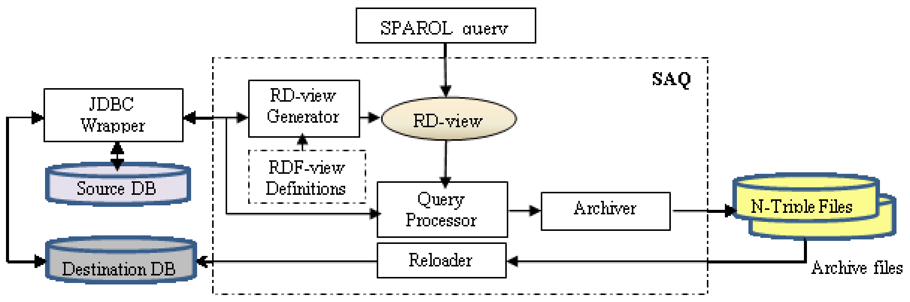


Fig. 2. SAQ Architecture

by using the *RD-view definitions*. The *query processor* executes arbitrary SPARQL queries to the RD-view by accessing the source DB through the *JDBC wrapper*.

When the source DB or part of it is to be preserved as RDF a SPARQL query extracting the desired content is sent to the query processor. The query result is transformed by the *archiver* into RDF triples and stored in a repository of *archive files* as N-Triples [14]. The archiver stores two N-triple files – one with the transformed query result, called the *data archive* and another one with schema information, called the *schema archive*. For instance, when all instances of class *%offerType%* representing the content of the relational table *offer* are unloaded as RDF, *Q1* is executed by the query processor in SAQ.

Later on, when a preserved database is to be restored, the *reloader* reads the archive files from the repository and makes it live again by reloading it into a *destination DB*. It first reads the schema archive to generate the relational database schema and then loads the data by reading the data archive.

3.1 RD-view Definition

In the RD-view each relational table is represented as an RDFS class, while each column is represented as an RDF property, as prescribed in [22]. For example, the table *person* is represented by the RDFS class URI *saq:product#person.* The URIs *saq:product#person/name* and *saq:product#person/country* are the RDF properties associated with the columns *name* and *country* of table *person*.

SAQ contains four meta-tables *cMap*, *pMap, fkMap,* and *rmmMap* providing mappings between RDF resources and the relational schema. The *class mapping table, cMap(T, ClassID),* maps 1-1 each relational table named *T* to the corresponding RDFS class *ClassID*. Here *T* is primary key and *ClassID* is secondary key. The *property mapping table, pMap(T,* A_j*, PropID),* maps each column (attribute) named A_j in table *T* into an RDF property *PropID*. The composite primary key is $T + A_j$, and *PropID* is secondary key. The foreign key mapping table, *fkMap(T,* A_i*, T', ResID)* maps a foreign key attribute A_i in a table *T* referencing table *T'* into an RDF resource *ResID*. The composite primary key of *fkMap* is $T + A_i + T'$, and *ResID* is secondary key. Finally, the many-to-many relationship mapping table, *rmmMap(T,* A_m*,* A_n*, T', T'', RmmID)* maps the attribute pair (A_m, A_n) of the composite primary key in *T*, where A_m and A_n are foreign key attributes referencing tables *T'* and *T''* into an RDF resource *RmmID*.

SAQ populates the meta-tables by accessing the catalogue of the relational database to construct identities of *ClassID, PropID, ResID,* and *RmmID*. Furthermore, the user can update the mapping tables to override default mappings in order to match some ontology or to limit data access.

In the following Datalog definitions we use capital letters to denote constants and small letters to denote variables.

The RD-view, defined in Datalog is a union of the following sub-views: the *relational column views* $C_{T,A}$, *the foreign key views*, $FK_{T,A}$, the *many-to many relationship views*, MM_T, and the *row class views,* RC_T. For instance, the RD-view for the *product* database, *RD-view*$_{product}$ is a union of:

- the relational column views for attributes *price* and *deliveryDays* of table *offer*
- the relational column views for attributes *name*, *country,* and *publisher* of table *person*
- the relational column view for attribute *reviewDate* of table *review*
- the foreign key view for attribute *person* of table *review*
- the row class views for tables *offer*, *person,* and *review*

The RD-view for the *product* database is

$$\begin{array}{ll}
\texttt{RD-view}_{\texttt{product}}\texttt{(s,p,v) :-} & \\
\texttt{C}_{\texttt{offer.price}}\texttt{(s,p,v)} & \texttt{OR} \\
\texttt{C}_{\texttt{offer.deliveryDays}}\texttt{(s,p,v)} & \texttt{OR} \\
\texttt{C}_{\texttt{person.name}}\texttt{(s,p,v)} & \texttt{OR} \\
\texttt{C}_{\texttt{person.country}}\texttt{(s,p,v)} & \texttt{OR} \\
\texttt{C}_{\texttt{person.publisher}}\texttt{(s,p,v)} & \texttt{OR} \\
\texttt{C}_{\texttt{review.reviewDate}}\texttt{(s,p,v)} & \texttt{OR} \\
\texttt{FK}_{\texttt{review.person}}\texttt{(s,p,v)} & \texttt{OR} \\
\texttt{RC}_{\texttt{offer}}\texttt{(s,p,v)} & \texttt{OR} \\
\texttt{RC}_{\texttt{person}}\texttt{(s,p,v)} & \texttt{OR} \\
\texttt{RC}_{\texttt{review}}\texttt{(s,p,v)} &
\end{array} \tag{5}$$

In general, an RD-view in SAQ has the following structure:

$$\begin{array}{ll}
\texttt{RD-view(s,p,v) :-} & \\
\underset{\texttt{T}}{\texttt{OR}}\ \texttt{(}\underset{\texttt{A}}{\texttt{OR}}\ \texttt{(C}_{\texttt{T.A}}\ \texttt{(s,p,v)))} & \texttt{OR} \\
\underset{\texttt{T}}{\texttt{OR}}\ \texttt{(}\underset{\texttt{A}}{\texttt{OR}}\ \texttt{(FK}_{\texttt{T.A}}\texttt{(s,p,v)))} & \texttt{OR} \\
\underset{\texttt{T}}{\texttt{OR}}\ \texttt{(MM}_{\texttt{T}}\texttt{(s,p,v))} & \texttt{OR} \\
\underset{\texttt{T}}{\texttt{OR}}\ \texttt{(RC}_{\texttt{T}}\texttt{(s,p,v))} &
\end{array} \tag{6}$$

where $\underset{B}{OR}\ (P)$ denotes a disjunction over all *P* for each possible value of *B*.

In general, a relational column view $C_{T,A}$ (7a) is defined for each attribute (column) A that is neither primary key nor a foreign key attribute of each table *T*.

$C_{T.A}$ (s,p,v) :-		$C_{review.reviewDate}$(s,p,v):		
R_T(a_1,..,a_i,..,a_r)	AND	R_{review}(rnr,person,reviewDate)	AND	
cMap(T,cid)	AND	cMap('review',cid)	AND	
pMap(T,A_i,p)	AND	pMap('review,'reviewDate',p)	AND	(7)
rowid(cid,(a_1,..,a_k),s)	AND	rowid(cid,(rnr),s)	AND	
v= a_i		v = reviewDate		
a)		b)		

In (7a) R_T is the *source predicate* representing the relational database table *T*, and A_j is the name of the attribute *A* in *T*. (a_1, ...,a_j, ..., a_r) is a tuple representing a row in *T*. The primary key of *T* is represented by the tuple (a_1,...,a_k). The variable *cid* is bound to the RDFS class associated with table *T*. The *rowid* predicate maps row identifiers to relational rows. It creates a unique URI *s* representing a row identifier in *T* by concatenating the class associated with a table and the primary key of a row. The predicate *rowid* is implemented in SAQ as a multidirectional foreign predicate [12], which implements both i) the construction of a new row identifier as described above and ii) its inverse to access the primary key based on a known row identifier. The URI *p* represents a relational attribute as an RDF property. The *attribute variable* a_j (and its alias *v*) holds the value of attribute A_j in a row.

Example (7b) shows the relational column view $C_{review,reviewDate}$ that represents attribute *reviewDate* in table *review*. R_{review} is the source predicate of table *review* and *cid* is bound to its associated class.

A *foreign key view* $FK_{T,A}$ for non-composite foreign key *A* in table *T* has the structure in (8a). $FK_{T,A}$ is defined in terms of the class URI *cid* associated with *T*, the class URI *cid'* associated with the table *T'* owning the foreign key, and the name of the foreign-key attribute, A_i. For example, (8b) shows the foreign key view $FK_{review,person}$. SAQ supports composite key foreign key views as well, which is not elaborated here.

$FK_{T.A}$(s,p,v) :-		$FK_{review.person}$(s,p,v) :-		
R_T(a_1,..,a_i,..,a_r)	AND	R_{review}(rnr,person,reviewDate)	AND	
cMap(T,cid)	AND	cMap('review',cid)	AND	
rowid(cid,(a_1,..,a_k),s)	AND	rowid(cid,(rnr),s)	AND	
fkMap(T,A_i,T',p)	AND	fkMap('review','person',	AND	(8)
		'person',p)		
cMap(T',cid')	AND	cMap('person',cid')	AND	
rowid(cid',(a_i),v)		rowid(cid',(nr),v)		
a)		b)		

A *many-to-many relationship view* $MM_{T,A,B}$ defined a many-to-many relationship between two other tables *T'* and *T''* represented by a connection table *T* of foreign keys (*A*, *B*) [22]. This is not further elaborated here.

Finally, a *row class view* RC_T (9a) represents the RDF classes of the row identifiers in a relational table *T*. For example, (9b) shows the row class view for table *review,* RC_{review}.

$$
\begin{array}{ll|ll}
\texttt{RC}_{\texttt{T}}\texttt{(s,p,v) :-} & & \texttt{RC}_{\texttt{review}}\texttt{(s,p,v) :-} & \\
\texttt{R}_{\texttt{T}}\texttt{(a}_{\texttt{1}}\texttt{,...,a}_{\texttt{r}}\texttt{)} & \texttt{AND} & \texttt{R}_{\texttt{review}}\texttt{(rnr,person,reviewDate)} & \texttt{AND} \\
\texttt{cMap(T,cid)} & \texttt{AND} & \texttt{cMap('review',cid)} & \texttt{AND} \\
\texttt{rowid(cid,(a}_{\texttt{1}}\texttt{,..,a}_{\texttt{k}}\texttt{),s)} & \texttt{AND} & \texttt{rowid(cid,(rnr),s)} & \texttt{AND} \\
\texttt{p = rdf:type} & \texttt{AND} & \texttt{p = rdf:type} & \texttt{AND} \\
\texttt{v = cid} & & \texttt{v = cid} & \\
\text{a)} & & \text{b)} &
\end{array}
\qquad (9)
$$

3.2 Query Processing

The main steps of the query processing in SAQ are illustrated by Fig. 3. The *SPARQL parser* transforms the SPARQL query into a Datalog expression where each SPARQL triple pattern becomes a reference to the RD-view. The *RD-view expander* recursively expands each RD-view reference in the query into a disjunctive expanded RD-view. It thereby looks up the mapping tables *cMap, pMap*, *fkMap,* and *rmmMap* to replace the variables in the expanded RD-view with corresponding URIs. Then it simplifies the query by unifying terms [9]. Each bound-property triple pattern is thereby simplified into a single conjunction [15] since the property URI determines the accessed table's attribute. However, each unbound-property triple pattern will remain a disjunction. The *DNF-normalizer* transforms the simplified query to a disjunctive normal form (DNF) predicate. The *GCT transformer* applies the GCT rewrite rule to transform the DNF predicate into a more efficient representation. It groups those common terms in different disjuncts of the DNF predicate that can be translated to SQL. The *SQL generator* generates calls to SQL for each grouped query fragment. The *post-processor* transforms the result from the SQL queries sent to the relational DBMS, e.g. it constructs URI objects and forms SPARQL result tuples. All processing in the system is streamed so that no large intermediate collections are generated.

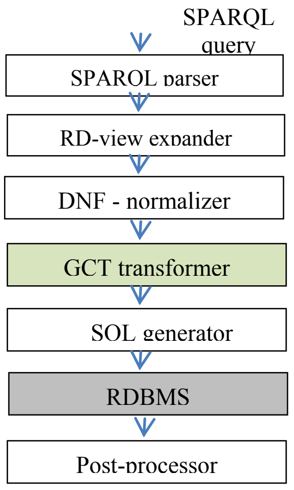


Fig. 3. Query processing in SAQ

4 The GCT Rule

The GCT rule is applied on a DNF predicate. It extracts from the disjuncts common terms that can be translated to SQL queries, i.e. source predicates R_T and SQL comparisons predicates. After GCT the DNF predicate becomes a disjunction of conjunctions between grouped terms and disjunctions of the remaining terms with the grouped terms removed. The remaining terms cannot be expressed in SQL and must be post-processed. For example, (10) shows the view expanded, simplified, and DNF normalized unbound-property query *Q4* where the tables *cMap*, *pMap,* and *fkMap* have been looked up by the RDF-view expander to obtain for each table *T* its associated RDFS class C_T, and the RDF properties $P_{T,Aj}$=*saq:product#T/*A_j representing the attributes A_j in *T*.

```
Q4(s,p,v) :-
(Rperson(nr,v,'JP',publisher)           AND
 Rreview(rnr,nr,reviewDate)             AND
 rowid(Cperson,(nr),s)                  AND
 rowid(Creview,(rnr),s1)                AND
 p = saq:product#person/name)                OR
(Rperson(nr,name,'JP',publisher)        AND
 Rreview(rnr,nr,reviewDate)             AND
 rowid(Cperson,(nr),s)                  AND
 rowid(Creview,(rnr),s1)                AND
 p = saq:product#person/country)             OR
 v = 'JP' )                                                  (10)
(Rperson(nr,name,'JP',v)                AND
 Rreview(rnr,nr,reviewDate)             AND
 rowid(Cperson,(nr),s)                  AND
 rowid(Creview,(rnr),s1)                AND
 p = saq:product#person/publisher)           OR
(Rperson(nr,name,'JP',publisher)        AND
 Rreview(rnr,nr,reviewDate)             AND
 rowid(Cperson,(nr),s)                  AND
 rowid(Creview,(rnr),s1)                AND
  p = rdf:type                          AND
  v = Cperson )
```

We notice that bold marked terms, representing source predicates, can be pulled out and later translated to a single SQL join query. In the example GCT will produce the predicate:

```
Q4(s, p, v) :-
(Rperson(nr,name,'JP',publisher)        AND
 Rreview(rnr,nr,reviewDate))            AND
  (rowid(Cperson,(nr),s)                AND
   rowid(Creview,(rnr),s1)              AND
   p = saq:product#person/name          AND
   v = name)                                 OR
  (rowid(Cperson,(nr),s)                AND
   rowid(Creview,(rnr),s1)              AND
   p = saq:product#person/country       AND              (11)
   v = 'JP')                                 OR
  (rowid(Cperson,(nr),s)                AND
   rowid(Creview,(rnr),s1)              AND
   p = saq:product#person/publisher     AND
   v = publisher)                            OR
  (rowid(Cperson,(nr),s)                AND
   rowid(Creview,(rnr),s1)              AND
  p = rdf:type                          AND
  v = Cperson)
```

In (11) the bold marked accesses to the source predicates are broken out from the disjunction and referenced only once, while in (10) the source predicates are referenced in each disjunct.

Without GCT the SQL generator will construct several SQL queries, one for each conjunction that can be translated to SQL. For example, in (10) the four bold marked conjunctions would produce these four SQL queries:

```
1) SELECT name from person p, review r WHERE p.country='JP'
   AND p.nr=r.person
2) SELECT   country   from   person   p,   review   r   WHERE
   p.country='JP' AND p.nr=r.person
3) SELECT   publisher   from   person   p,   review   r   WHERE
   p.country='JP' AND p.nr=r.person
4) SELECT nr from person p, review r WHERE p.country='JP'
   AND p.nr=r.person
```
(12)

The four queries would be sent to the relational DBMS and their result tuples postprocessed in sequence. Binding of *s*, *p* and *v* has to be performed by the not bold marked post-processing predicates in (10) after the rows are retrieved since object construction and result variable bindings cannot be expressed in SQL.

With GCT applied in (11) a single SQL query is produced from the grouped terms:

```
SELECT  nr,  name,  country,  publisher  from  person     p,
review r WHERE p.country='JP' AND p.nr=r.person
```
(13)

In (11) the values from the relational tables are retrieved in row-order, while in (10) they are retrieved column-by-column. Therefore the execution plan generated from (11) is more efficient than the execution plan from (10).

In general, the steps of the GCT algorithm applied on a DNF predicate are the following:

(i) In a pre-step, normalize the variable names of the DNF predicate so that the same variable names are used in each disjunct.

(ii) Allocate a hash table that for each predicate group maintains mappings to the disjuncts from which its predicates have been extracted.

(iii) For each disjunct, extract the source predicates and SQL comparison predicates to form a conjunctive predicate group to extract. Use the entire extracted conjunction as key in the hash table.

(v) After the entire DNF predicate is scanned, go through the hash table and form for each extracted conjunction *c* a conjunction between *c* and the remaining terms in the disjuncts where *c* occurs. Finally, form a disjunction of all constructed conjunctions. In the example this transforms (10) into (11).

The pseudo code of the GCT algorithm is the following:

```
Function GCT(P, gf) -> GP
Input:P:  a DNF predicate with normalized variable names
      gf: a function that extracts a conjunction of specific
          terms, e.g. $R_T$, SQL comparisons from a conjunction
Output:GP: P grouped on the common terms
1. Allocate a hash table Ht for the common terms in disjuncts
2. GP:=null
3. for each disjunct D in P do
```

```
4.   if D is an atom then GP:= orify(GP,D)
5.   else if D is not a conjunction then GP:=orify(D,GP)
6.   else if D has only one term then GP:=orify(D,GP)
7.   else CT:=gf(D)   /*CT is a list of common terms)*/
8.        if CT=null then GP:=orify(D,GP)
9.        else put in Ht(key=CT):=
               orify((D with CT removed),
               (existing value for CT in Ht ))
10. for each (CT' and valueCT') in Ht do
11.   GP:=orify(andify(CT', valueCT'),GP)
14. return GP
```

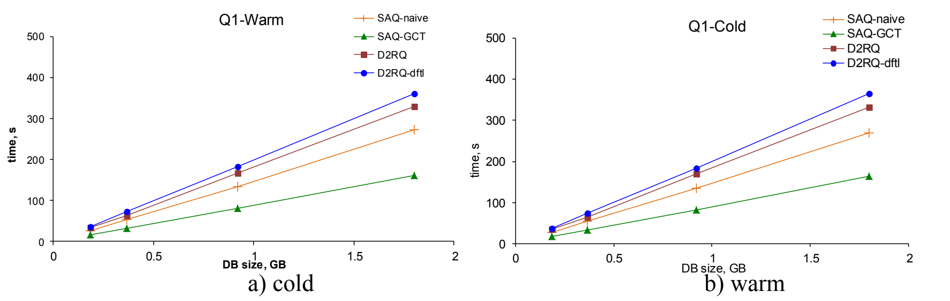
The function *orify(x,y)* forms a disjunction between predicates *x* and *y*, and *andify(x,y)* forms a conjunction.

We notice that the processing is done in one pass and is therefore *O(N)* where *N* is the number of disjuncts in the DNF predicate.

5 Performance

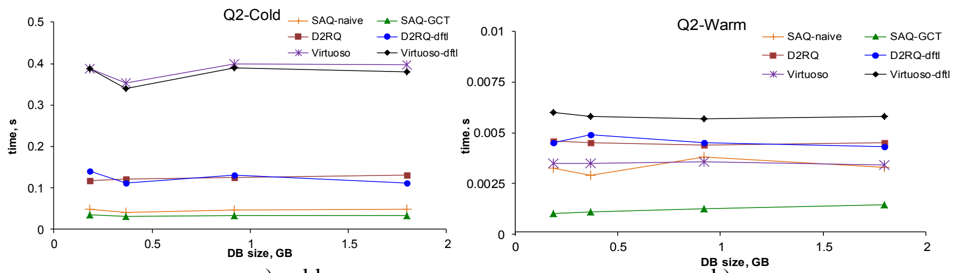
The measurements were made on a PC Intel(R) Core(TM), 2Quad CPU Q9400 with 2.67 GHz and 8 GB RAM running 64-bits Windows 7 Professional. The impact of GCT was evaluated for the unbound-property queries *Q1, Q2, Q3,* and *Q4*. We compare the performance of SAQ with Virtuoso RDF Views [8] and D2RQ [4], all systems accessing the same back-end MS SQL Server database. The experiment configuration was the following:

1. MS SQL server 2008 was configured with the default settings for the min and max server memory.
2. The SQL data was generated by the Berlin benchmark data generator [1][2] and loaded into the relational database.
3. Non-clustered, non-unique indexes were put on the columns *country* and *name* in the *person* table to speed up queries *Q3* and *Q4*.
4. For Virtuoso RDF Views, the RDF view to the underlying relational database was generated on the Virtuoso server (ver. 06.02.3128, Windows-64) by using the *Virtuoso Conductor* tool. The SPARQL queries to the RDF view were run from a Java program, implementing a *Jena Provider* [23], which allows users to query Virtuoso RDF views from Java. The Java heap size was set to 1 GB.
5. For D2RQ (v.07), the RDF view of the underlying RDBMS was generated by D2RQ's auto-generated mapping script [3]. In the generated script we inserted the option "*d2rq:useAllOptimizations true*" to guarantee that we use full optimization in D2RQ. The SPARQL queries were run from a Java program calling the D2RQ Engine through Jena2 [3]. The Java heap size was set to 1 GB.
6. All measurements were made five times and the mean values plotted. The standard deviation was less than 10% in all measurements.
7. The default mappings of the analyzed systems SAQ, Virtuoso RDF Views, and D2RQ all produce different results. For example, some redundant labels and inverse properties are produced by Virtuoso RDF Views and D2RQ. To make fair comparisons we configured the systems so that they all generated the same query



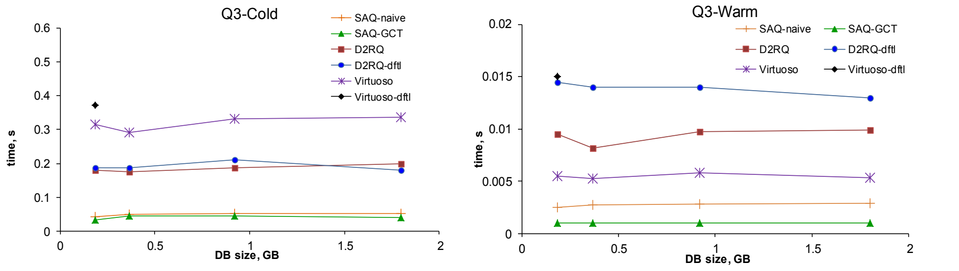
a) cold b) warm

Fig. 4. Execution times for Q1 up to 1.8 GB database



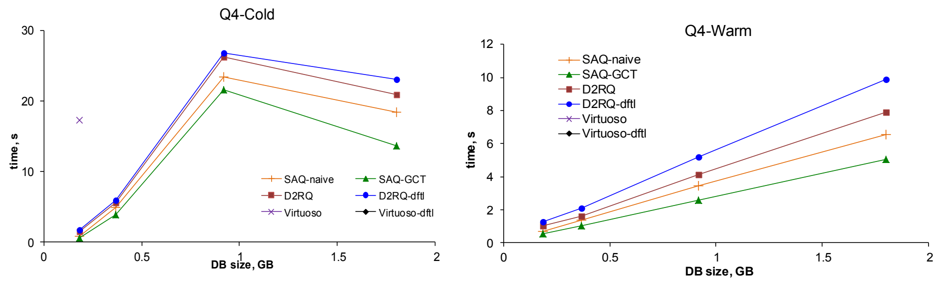
a) cold b) warm

Fig. 5. Execution times for Q2 up to 1.8 GB database



a) cold b) warm

Fig. 6. Execution times for Q3 up to 1.8 GB database



a) cold b) warm

Fig. 7. Execution times for Q4 up to 1.8 GB database

result. To investigate whether the performance is better with the default mappings we also measured Virtuoso RDF Views and D2RQ with their default mappings.

The following notation is used in the performance diagrams:

- ***Virtuoso***: Virtuoso RDF Views configured with the SAQ mappings.
- ***Virtuoso-dflt***: Virtuoso RDF Views configured with the system default mappings.
- ***D2RQ***: D2RQ configured with the SAQ mappings.
- ***D2RQ-dflt***: D2RQ configured with the system default mappings
- ***SAQ-naive***: SAQ without GCT
- ***SAQ-GCT*****:** SAQ with GCT

In all cases the time spent in executing the query by the relational database followed by post-processing is measured, thus not including the time for preparing the SPARQL query by the respective system. The cold execution measurements were made immediately after flushing the buffer pool, while the warm ones were made by re-executing the query immediately after a cold query was run. The cold execution times include reading data from disk and SQL query optimization in the DBMS server. Since the back-end DBMS has a statement cache a same SQL query executed twice will be optimized the first time it is received. Therefore, the warm executions do not include back-end DBMS query optimization time.

Table 1 Speed-up of ***SAQ-GCT*** compared to other approaches

Q1

system	***SAQ-GCT***	***SAQ-naive***	***D2RQ***	***D2RQ-dftl***	***Virtuoso***	***Virtuoso-dftl***
cold	1	1.65	2	~ 2.2	>8 hours	>8hours
warm	1	1.6	2	~ 2.2	>8hours	>8hours
SQL queries	1	11	11	13	>1000	>1000

Q2

system	***SAQ-GCT***	***SAQ-naive***	***D2RQ***	***D2RQ-dftl***	***Virtuoso***	***Virtuoso-dftl***
cold	1	1.3 - 1.5	3.5 - 4	3.5 - 4	11.4 - 12.4	11 - 12
warm	1	2.2 - 3.2	3 - 4.5	3 - 4.5	2.3 - 3.5	4 - 6
SQL queries	1	10	4	4	11	18

Q3

system	***SAQ-GCT***	***SAQ-naive***	***D2RQ***	***D2RQ-dftl***	***Virtuoso***	***Virtuoso-dftl***
cold	1	1.2 - 1.3	4.2 - 5	4 - 5.5	6.6 - 9	11 - 480
warm	1	2.5 - 3	8 - 9.7	13 - 14	5.3 - 5.7	15 - 380
SQL queries	1	6	6	8	9	12

Q4

system	***SAQ-GCT***	***SAQ-naive***	***D2RQ***	***D2RQ-dftl***	***Virtuoso***	***Virtuoso-dftl***
cold	1	1.2 - 1.3	1.2 - 1.5	1.2 - 2.8	28 - 800	100 - 3000
warm	1	1.3	1.6 - 2	2	30 - 2200	120 - 8000
SQL queries	1	6	6	8	>1000	>1000

The results from the measurements are presented in Fig. 4-7. The figures show the execution times for *Q1, Q2, Q3,* and *Q4* while scaling the generated Berlin benchmark dataset from 10M to 100M [1][2], which corresponds to scaling the relational database from 312 MB to 1.8 GB. The number of RDF triples in the RD-view varied from 3 949 935 to 38 771 340.

Table 1 summarizes the speed-up of the different approaches for *Q1-Q4* compared to ***SAQ-GCT***. In particular, the ***SAQ-naive*** column shows the speed-up of GCT.

The performance of ***SAQ-GCT*** for simple unbound-property queries is better than all compared implementations. Furthermore, GCT always improves performance substantially (20-65%) for queries to cold databases, where the execution time is dominated by disk accesses on the database server. For the queries *Q2* and *Q3,* which select a single row from the buffer pool in a warm database, the improvement is even better (220-320%). The reason is that without GCT more SQL requests are sent to the server and therefore the communication overhead dominates when the server time is insignificant.

To analyze how the other systems process unbound-property queries we measured their performance and investigated what SQL queries were sent to the relational database. For ***D2RQ*** we used the profiling tool of MS SQL Server 2008 to obtain the SQL queries sent to the DBMS. Normally ***D2RQ*** sends exactly the same SQL queries as ***SAQ-naive*** so GCT is not used. For *Q2* ***D2RQ*** makes a special optimization when the subject of a triple pattern is a constant URI so fewer queries are sent.

Virtuoso RDF Views translates unbound-property queries to SQL using an unknown algorithm [7][8]. The debug logging of Virtuoso was used to investigate what SQL queries were sent to the relational database. For *Q2* and *Q3* ***Virtuoso*** also sends exactly the same queries as **SAQ-naive** plus a number of additional queries. For the non-selective queries *Q1* and *Q4* more than 1000 additional SQL queries were sent and the processing did not scale for large databases.

6 Related Work

Virtuoso RDF Views [7][8] and D2RQ [3][4][5] are other systems that allow mapping of relational tables and views into RDF to make them queryable by SPARQL. These systems implement compilers that translate SPARQL directly to SQL. By contrast, SAQ first generates Datalog queries to a declarative RD-view of the relational database, and then transforms the SPARQL queries to SQL based on logical transformations. We have shown that the particular query transformation GCT significantly improves performance for unbound-property queries.

We did not find any publication of how D2RQ compiles unbound-property SPARQL queries into SQL. The documentation for Virtuoso is very limited [7][8]. However, by using the profiling tool of the DBMS and the debug logging of Virtuoso we were able to analyze what queries were actually sent to the DBMS, showing that neither of those systems uses anything similar to GCT. SquirrelRDF also allows SPARQL queries to relational tables, but does not support unbound-property SPARQL queries [19] [20].

Work on optimizing disjunctive database queries in general is described in [6][11][13]. The closest work to GCT is the combinatorial algorithm [13], which merges disjuncts with common sub-expressions in general disjunctive logical expression in order to avoid repeated evaluation of the same predicate on the same tuple. By contrast, the purpose of GCT is to group in a DNF predicate query fragments that can be translated to SQL, and therefore the simpler linear GCT algorithm can be used.

The idea of bypass evaluation of disjunctive queries in [6][11] is based on implementing specialized operators that produce two output streams: the true-stream of the tuples that fulfill the operator's predicate and the false-stream of the tuples that do not match. The main profit of the technique of bypass evaluation is in eliminating duplicates by avoiding unnecessary join operators. The purpose of GCT is not duplicate elimination, but to rewrite complex disjunctive queries for faster execution.

7 Conclusions

We have presented an approach to optimize simple unbound-property SPARQL queries to RDF views over back-end relational databases in a system called SAQ for querying and archiving relational databases as RDF. Simple unbound-property queries retrieve dynamic sets of properties for given subjects, which is important for archiving selected parts of a database with SPARQL. Such queries are optimized by the presented GCT (Group Common Terms) query transformation rule, which groups those common terms from a DNF predicate that can be translated to SQL.

By using data from the Berlin SPARQL benchmark, GCT was shown to improve query execution time compared to naïve processing. Compared to not using GCT, it reduces the number of SQL queries to execute and retrieves data in relational row order rather than column order. The performance of SAQ was compared to other systems that support SPARQL queries to views of existing relational databases. It was shown experimentally that SAQ with GCT performs better than those systems, since they do not use any similar transformation strategy.

Future work includes investigating the impact of GCT and other rewrite rules on the performance of other kinds of queries, such as queries with multiple unbound-property triple patterns and other kinds of archival queries.

References

1. Bizer, C. and Schultz, A.: The Berlin SPARQL Benchmark (BSBM) Specification – V3.0, http://www4.wiwiss.fu-berlin.de/bizer/BerlinSPARQLBenchmark/spec/index.html (2010).
2. Bizer, C. and Schulz, A.: The Berlin SPARQL Benchmark, Journal of Semantic Web and Information Systems, special issue on scalability and performance of semantic web systems, Vol. 5, Issue 2, pp 1-24 (2009)
3. Bizer, C., Cyganiak R., Garbers, G., Maresch, O., and Becker C.: The D2RQ Platform v0.7 - Treating Non-RDF Relational Databases as Virtual RDF Graph. http://www4.wiwiss.fu-berlin.de/bizer/d2rq/spec/ (2009)
4. Bizer, C. and Seaborne A.: D2RQ-Treating Non-RDF Databases as Virtual RDF Graphs, Poster at 3rd International Semantic Web Conference (2004)
5. Bizer, C. and Cyganiak, R.: D2R Server-Publishing Relational Databases on the Semantic Web, Poster at the 5th International Semantic Web Conference (2006)
6. Claussen, J., Kemper, A., Peithner, K., and Steinbrunn, M.: Optimization and Evaluation of Disjunctive Queries, IEEE Transactions on Knowledge and Data Engineering, Vol. 12, No 12, March/April (2000)

7. Erling, O.: Declaring RDF views of SQL Data, W3C Workshop on RDF Access to Relational Databases, 25-26 October, Cambridge, MA, USA (2007)
8. Erling, O and Mikhailov, I.: RDF Support in the Virtuoso DBMS, Springer, ISSN: 1860-949X (Print) 1860-9503 (Online), in Studies in Computational Intelligence, Vol. 221(2009)
9. Fahl, G. and Risch, T.: Query Processing over Object Views of Relational Data, The VLDB Journal , Vol. 6, No. 4, pp 261-281 (1997)
10. Garcia-Molina, H., Ullman, J. D. and Widom, J.: Database Systems, the complete book, Prentice Hall (2002).
11. Kemper, A., Moerkotte, G., Pethner, K., and Steinbrunn, M.: Optimizing Disjunctive Queries with Expensive Predicates, in Proc. ACM SIGMOD, Conf. Management of Data, pp. 336-347 (1994)
12. Litwin, W. and Risch, T.: Main Memory Oriented Optimization of OO Queries Using Typed Datalog with Foreign Predicates, IEEE Transactions on Knowledge and Data Engineering, Vol. 4, No. 6 (1992)
13. Muralikrishna, M. and Witt, D. J De.: Optimization of Multiple-Relation Multiple-Disjunct Queries, PODS '88 Proceedings of the seventh ACM SIGACT-SIGMOD-SIGART, pp. 263-275 (1988)
14. N-triples, W3C RDF Core WG Internal Working Draft, http://www.w3.org/2001/sw/RDFCore/ntriples/
15. Petrini, J. and Risch, T.: Processing queries over RDF views of wrapped relational databases, in Proceedings of the 1st International Workshop on Wrapper Techniques for Legacy Systems, WRAP 2004, Delft, Holland (2004)
16. Petrini, J.: Querying RDF Schema Views of Relational Databases, PhD Thesis, Uppsala University, Department of IT, ISSN 1104-2516, http://www.it.uu.se/research/group/udbl/Theses/JohanPetriniPhD.pdf (2008)
17. Risch, T. and Josifovski, V.: Distributed Data Integration by Object-Oriented Mediator Servers, Concurrency and Computation: Practice and Experience, J. 13(11), John Wiley & Sons (2001)
18. Schmidt, M., Hornung, T., Lausen, G. and Pinkel, C.: SP2Bench: A SPARQL Performance Benchmark, ICDE 2009, pp. 222-233 (2009)
19. Seaborne, A., Steer, D., and Williams, S.: SQL-RDF, http://www.w3.org/2007/03/RdfRDB/papers/seaborne.html (2007)
20. SqirrelRDF, http://jena.sourceforge.net/SquirrelRDF/
21. Stuckenschmidt, H. and Harmelen, F.: Information Sharing on the Semantic Web, Springer, ISBN 3-540-20594-2 (2005)
22. Sören, A., Feigenbaum, L., Miranker, D., Fogarolli, A., and Sequeda J.: Use Cases and Requirements for Mapping Relational Databases to RDF, W3C Working Draft 8, http://www.w3.org/TR/rdb2rdf-ucr/ (2010)
23. Virtuoso Jena Provider, OpenLink Virtuoso Universal Server: Documentation, http://docs.openlinksw.com/virtuoso/rdfnativestorageproviders.html#Rdfnativestorageprovidersjena (2009)

Query Rewriting Under Query Extensions for OWL 2 QL Ontologies

Tassos Venetis[1], Giorgos Stoilos[2], and Giorgos Stamou[1]

[1] School of Electrical and Computer Engineering
National Technical University of Athens
Zographou Campus, 15780, Athens, Greece

[2] Department of Computer Science, University of Oxford
Wolfson Building, Parks Road, Oxford, UK

Abstract. Conjunctive query answering is a key reasoning service for many ontology-based applications. With the advent of lightweight ontology languages, such as OWL 2 QL, several query answering systems have been proposed which compute the so called *UCQ rewriting* of a given query. It is often the case in realistic scenarios, that users refine their original queries, by e.g., extending them with new constraints and making them more precise. To the best of our knowledge, in such cases, all OWL 2 QL systems would need to recompute the rewriting of the refined query from scratch. In this paper we study the problem of computing the rewriting of the refined query by 'extending' the pre-computed rewriting and avoiding re-computation. We study the problem from a theoretical point of view and present a practical algorithm. Finally, we evaluate our implementation experimentally by comparing it against many state-of-the-art query rewriting systems, obtaining encouraging results.

1 Introduction

A key application of OWL ontologies is ontology-based data access (OBDA) [15], where an ontology is used to support query answering against distributed and/or heterogeneous data sources. A typical scenario would involve the use of an OWL ontology to answer conjunctive queries over RDF datasets. Due to the high complexity of answering conjunctive queries over OWL 2 DL ontologies [11, 6], prominent languages such as OWL 2 QL have been developed. OWL 2 QL (a well-known OWL 2 profile[3]) is based on the well-known Description Logic (DL) DL-Lite$_R$ [3, 1]. DL-Lite$_R$ is a member of the DL-Lite family [3, 1], a family of 'lightweight' ontology languages specifically designed to feature low theoretical complexity, and hence imply the existence of efficient query answering algorithms.

Query answering in the DL-Lite family is usually performed via a technique called *query rewriting*. According to this technique, given a *query* and a DL-Lite ontology, the query is 'rewritten' into a set of queries such that, the union

[3] `http://www.w3.org/TR/owl2-profiles/`

of the answers of the queries in the set over the input data and by discarding the input ontology is equal to the answers of the original query over the data and the ontology. In recent years, query rewriting over DL-Lite ontologies has drawn significant attention, and several different algorithms and systems have been proposed [3, 13, 4, 16].

Quite often in realistic data-access scenarios, users do not immediately ask the query that they want. More precisely, as has been shown in the Web literature [9, 8, 12], users usually first ask some 'general' query and then, according to the results they get back, refine it by adding further constraints, making their request more specific each time. Consequently, the actual (final) query might only be known after several refinements of the initial one. In a different (Semantic Web) motivating scenario, in order to assist users in constructing their queries, iterative and incremental techniques have also been proposed [18, 5, 17]. However, to the best of our knowledge, none of the query rewriting approaches that have been proposed in the literature is designed to work well in such scenarios. More precisely, in such cases all algorithms will (re)compute the entire rewriting of each of the refined queries from scratch.

In the current paper we study the following problem: Given a DL-Lite$_R$ ontology, a query and its rewriting (computed previously), and a new constraint to be added to the query, compute the rewriting of the new query by 'extending' the input rewriting and avoid computing it from scratch through the standard algorithms. First, we study the problem at a theoretical level and investigate whether it is possible to compute the rewriting of the extended query given any input rewriting. Unfortunately, the answer is negative and we explain how optimisation techniques employed by nearly all modern systems might compute rewritings that are not suitable for our task. Then, we present an algorithm for computing query rewritings under query extensions. Our algorithm is based on the well-known PerfectRef algorithm [3], which in its original version did not include any optimisations and hence is suitable for our purposes. For our algorithm to behave well we propose several optimisations that are specific to our case. Finally, we have implemented the algorithm and have conducted an experimental evaluation using the evaluation framework proposed in [13]. We have compared our techniques with many of the available query rewriting systems and our first results are encouraging given our preliminary implementation.

Our problem is also highly relevant to the field of databases, where it has been studied under the term *view adaptation* [7, 10]—that is, computing the materialisation of a re-defined materialised view. However, view adaptation has not been studied in the presence of database constraints—that is, under the presence of logical axioms. We also feel that this new approach and view of query rewriting opens an interesting line of research for scalable OBDA.

2 Preliminaries

Description Logics We assume that the reader is familiar with the basics of DL syntax, semantics and standard reasoning problems [2]. We next recapitulate the syntax of DL-Lite$_R$ [3] a prominent DL language that consists of the logical underpinnings of the QL profile of OWL 2[3] and is widely used in ontology-based data access.

Let $\mathbf{C}$, $\mathbf{R}$, and $\mathbf{I}$ be countable, pairwise disjoint sets of *atomic concepts*, *atomic roles*, and *individuals.* A DL-Lite$_R$-*role* is either an atomic role P or its *inverse* P^-. DL-Lite$_R$-*concepts* are defined inductively by the following grammar, where $A \in \mathbf{C}$ and R is a DL-Lite$_R$-role:

$$B := A \mid \exists R$$

A DL-Lite$_R$-TBox is a finite set of axioms of the form $B_1 \sqsubseteq B_2$ or $B_1 \sqcap B_2 \sqsubseteq \bot$, with $B_{(i)}$ DL-Lite$_R$-concepts and $\bot$ the *bottom* concept that is empty in all interpretations, or of the form $R_1 \sqsubseteq R_2$ with $R_{(i)}$ DL-Lite$_R$-roles. An ABox is a finite set of assertions of the form $A(c)$ or $P(c,d)$ for $A \in \mathbf{C}$, $P \in \mathbf{R}$ and $c, d \in \mathbf{I}$. A DL-Lite$_R$-ontology $\mathcal{O} = \mathcal{T} \cup \mathcal{A}$ consists of a TBox and an ABox.

Queries We use standard notions of (function-free) term and variable. A concept atom is of the form $A(t)$ with A an atomic concept and t a term. A role atom is of the form $R(t,t')$ for R an atomic role, and t, t' terms. A conjunctive query (CQ) q is an expression of the form:

$$\{\vec{x} \mid \{\alpha_1, \ldots, \alpha_m\}\}$$

where each α_i is a concept or role atom and $\vec{x} = (x_1, \ldots, x_n)$ is a tuple of variables called the *distinguished* (or answer) variables, each appearing in at least some atom α_i. The remaining variables of q are called *undistinguished.* We use $\mathsf{var}(q)$ to denote all the variables appearing in q and $\mathsf{avar}(q)$ to denote all its distinguished variables. Finally, a variable that is either distinguished or appears in at least two different atoms α_i, α_j with $i \neq j$ in q is called *bound*, otherwise it is called *unbound.* We often abuse notation and use q to refer to the set of its atoms, i.e., $\{\alpha_1, \ldots, \alpha_m\}$. For the rest of the paper, and without loss of generality, we will asume that queries are *connected* [6]. Finally, a union of conjunctive queries (UCQ) is a set of conjunctive queries.

A certain answer to a CQ q w.r.t. $\mathcal{O}$ is a tuple $\vec{c} = (c_1, \ldots, c_n)$ of individuals s.t. $\mathcal{O}$ entails the FOL formula obtained by building the conjunction of all atoms α_i in q, replacing each distinguished variable x_j with c_j and existentially quantifying over undistinguished variables. We denote with $\mathsf{cert}(q, \mathcal{O})$ the set of all certain answers to q w.r.t. $\mathcal{O}$. Given q_1, q_2 with distinguished variables $\vec{x}$ and $\vec{y}$, we say that q_2 *subsumes* q_1, if there exists a substitution θ from the variables of q_2 to the variables of q_1 such that the set $[\{ans(\vec{y})\} \cup q_2]_\theta$ is a subset of the set $\{ans(\vec{x})\} \cup q_1$, where ans is in each case a predicate of the same arity as $\vec{x}$ ($\vec{y}$) not appearing in q_1 (q_2). Finally, q_1 is *equivalent* to q_2 if they subsume each other.

Query Answering in DL-Lite$_R$ Query answering in DL-Lite$_R$ is performed with a technique known as *query rewriting* which, given a DL-Lite$_R$-TBox $\mathcal{T}$ and query q, computes a UCQ u, called a *UCQ rewriting* for $q, \mathcal{T}$, with the following property: for each ABox $\mathcal{A}$ s.t. $\mathcal{O} = \mathcal{T} \cup \mathcal{A}$ is consistent, the following holds:

$$\mathsf{cert}(q, \mathcal{O}) = \bigcup_{q' \in u} \mathsf{cert}(q', \mathcal{A}).$$

For a DL-Lite$_R$-TBox, a UCQ rewriting u for $q, \mathcal{T}$ can be computed using the perfect reformulation algorithm (PerfectRef) described in [3]. The algorithm applies exhaustively a *reformulation* and a *condensation* step that generate new CQs; the process terminates when no new CQ is generated.

In the reformulation step the algorithm picks a CQ q, an atom in the CQ $\alpha \in q$ and an axiom $I \in \mathcal{T}$ and *applies* the axiom on the atom α of q replacing it with a new atom and hence, creating a new CQ. This is performed with the function $\mathsf{gr}(\alpha, I)$, that takes as input an atom α and an axiom $I \in \mathcal{T}$ and returns a new atom. For a CQ q, $\alpha \in q$ and $I \in \mathcal{T}$, $\mathsf{gr}(\alpha, I)$ is defined as follows:

- if $\alpha = A(x)$ and
  - i) $I = B \sqsubseteq A$, then $\mathsf{gr}(\alpha, I) = B(x)$;
  - ii) $I = \exists P \sqsubseteq A$, then $\mathsf{gr}(\alpha, I) = P(x, y)$ for y a new variable in q;
  - iii) $I = \exists P^- \sqsubseteq A$, then $\mathsf{gr}(\alpha, I) = P(y, x)$ for y a new variable in q.
- if $g = P(x, z)$, z is unbound in q and
  - i) $I = A \sqsubseteq \exists P$, then $\mathsf{gr}(\alpha, I) = A(x)$;
  - ii) $I = \exists S \sqsubseteq \exists P$, then $\mathsf{gr}(\alpha, I) = S(x, y)$ for y a new variable in q;
  - iii) $I = \exists S^- \sqsubseteq \exists P$, then $\mathsf{gr}(\alpha, I) = S(y, x)$, for y a new variable in q.
- if $g = P(z, x)$, with z unbound, then
  - i) $I = A \sqsubseteq \exists P^-$, then $\mathsf{gr}(\alpha, I) = A(x)$;
  - ii) $I = \exists S \sqsubseteq \exists P^-$, then $\mathsf{gr}(\alpha, I) = S(x, y)$, where y is new in q;
  - iii) $I = \exists S^- \sqsubseteq \exists P^-$, then $\mathsf{gr}(\alpha, I) = S(y, x)$, where y is new in q.
- if $g = P(x, y)$ and
  - i) $I = S \sqsubseteq P$ or $I = S^- \sqsubseteq P^-$, then $\mathsf{gr}(\alpha, I) = S(x, y)$;
  - ii) $I = S \sqsubseteq P^-$ or $I = S^- \sqsubseteq P$, then $\mathsf{gr}(\alpha, I) = S(y, x)$.

If for some axiom I and some atom α one of the above conditions holds, then we say that I is *applicable* to α, and applying I to some α in some CQ q creates a new CQ of the form $q[\alpha/\mathsf{gr}(\alpha, I)]$—that is, the new CQ contains the atom $\mathsf{gr}(\alpha, I)$ instead of α.

In the condensation step a new query is generated from a query q by applying to q the most general unifier between two atoms α_1, α_2 of its body; the application of the condensation step is denoted by $\mathsf{reduce}(q, \alpha_1, \alpha_2)$.

3 Query Rewriting Under Query Extensions

In this section we present the design and implementation of an algorithm for computing the UCQ rewriting of a refined query from the UCQ rewriting of the initial query by avoiding the computation of the UCQ rewriting of the refined query from scratch as in any of the standard query rewriting algorithms. In the current paper we focus on refinements that involve additions of new atoms to the queries, which we call *extensions*. We leave other types of refinements like deletion of atoms or changes in the set of distinguished variables for future work.

In the following, we first study the problem at a theoretical level and give examples that highlight issues and difficulties and which also explain several of its technical parts; then, we present the algorithm in detail.

3.1 Algorithm Design

Example 1. Consider the following TBox about an academic domain and the CQ which retrieves all individuals that are students:

$$\begin{aligned}\mathcal{T} &= \{\mathsf{GradStudent} \sqsubseteq \mathsf{Student}, \mathsf{TennisPlayer} \sqsubseteq \mathsf{Athlete}\}\\ q &= \{x \mid \{\mathsf{Student}(x)\}\}\end{aligned}$$

The set $u = \{q, q_1\}$, where $q_1 = \{x \mid \{\mathsf{GradStudent}(x)\}\}$ is a UCQ rewriting for $q, \mathcal{T}$ and can be computed by any state-of-the-art query rewriting algorithm.

Suppose, now that a user wants to extend the initial query and retrieve only those students that are also athletes—that is, issue the new query $q' = \{x \mid \{\mathsf{Student}(x), \mathsf{Athlete}(x)\}\}$. It can be easily checked that the UCQ $u' = \{q', q'_1, q'_2, q'_3\}$, with q'_i defined as follows, is a UCQ rewriting for $q', \mathcal{T}$:

$$\begin{aligned}q'_1 &= \{x \mid \{\mathsf{Student}(x), \mathsf{TennisPlayer}(x)\}\},\\ q'_2 &= \{x \mid \{\mathsf{GradStudent}(x), \mathsf{Athlete}(x)\}\},\\ q'_3 &= \{x \mid \{\mathsf{GradStudent}(x), \mathsf{TennisPlayer}(x)\}\}\end{aligned}$$

which can again be computed using any rewriting algorithm for $q', \mathcal{T}$. ♢

Although u' from the above example can be computed by any state-of-the-art query rewriting algorithm and system, when such an algorithm is applied over q' it will 'repeat' all the work previously done for the atom $\mathsf{Student}(x)$, when it computed the UCQ rewriting for query q. Our motivation is that since all the work for q has already been done, perhaps it is possible to compute the UCQ rewriting of the extended query, by computing a UCQ rewriting only for the new atom (i.e., for the atom $\mathsf{Athlete}(x)$) and then, by appropriately 'combining' the two rewritings. Using this approach we will perform only the additional work left to compute a UCQ rewriting for the extended query, modulo the overhead for combining the rewritings which we anticipate to be small.

To design a correct algorithm several important technical issues need to be resolved. To mention a few, firstly, we need to figure out what is the appropriate CQ of the new atom for which a UCQ rewriting should be computed (especially regarding the choice of distinguished variable) while then, how to appropriately combine the two UCQ rewritings.

Definition 1. *Let q be a CQ and α an atom containing at least a variable of q. The* atom-query *for α w.r.t. q is the CQ defined as follows $q_\alpha := \{\mathsf{var}(\alpha) \cap \mathsf{var}(q) \mid \{\alpha\}\}$.*

The distinguished variables of the atom-query for α w.r.t. q are all the variables of α that also appear in q. The intuition is that, in the extended query $(q \cup \{\alpha\})$ these variables are bound and hence, when computing the UCQ rewriting for α in isolation, these should be treated as such. In our previous example, the atom-query for α w.r.t. q is the CQ $q_\alpha = \{x \mid \{\mathsf{Athlete}(x)\}\}$ and its UCQ rewriting is the UCQ $u_\alpha = \{q_\alpha, q'_\alpha\}$, where $q'_\alpha = \{x \mid \{\mathsf{TennisPlayer}(x)\}\}$.

Having computed a UCQ rewriting u_α for the atom-query, we can use the two UCQs to compute a UCQ rewriting for the extended query. At this point it is important to see how the two rewritings should be combined. The obvious choice is to take the pair-wise union of the CQs for which there is an overlap between their variables. More precisely, for u and u_α two UCQ rewritings and for $q_1 \in u, q_2 \in u_\alpha$ such that $\mathsf{avar}(q_2) \subseteq \mathsf{var}(q_1)$, a CQ of the form $\{\mathsf{avar}(q) \mid q_1 \cup q_2\}$ can be constructed. Again, the intuition behind the condition on the variables is that there must exist 'join'-points between the queries that are unified. Indeed, one can construct the UCQ rewriting u' for $q', \mathcal{T}$ from Example 1 by following this procedure: from queries $q_1 \in u$ and $q'_\alpha \in u_\alpha$ we can obtain the CQ q'_3, while from $q_1 \in u$ and $q_\alpha \in u_\alpha$ we can obtain the CQ q'_2. However, as the following example shows this operation between the UCQs is not enough to give a complete UCQ rewriting for the extended query.

Example 2. Consider the following TBox and CQ:

$$\mathcal{T} = \{A \sqsubseteq \exists R, R \sqsubseteq S, \exists S^- \sqsubseteq B\} \quad q = \{x \mid \{R(x,y)\}\}.$$

The set $u = \{q, q_1\}$, where $q_1 = \{x \mid \{A(x)\}\}$ is a UCQ rewriting for $q, \mathcal{T}$. Consider now the addition of the atom $\alpha = B(y)$. The atom-query for α w.r.t. q is the query $q_\alpha = \{y \mid B(y)\}$ and the UCQ $u_\alpha = \{q_\alpha, q^1_\alpha, q^2_\alpha\}$, where $q^1_\alpha = \{y \mid \{S(z,y)\}\}$ and $q^2_\alpha = \{y \mid \{R(z,y)\}\}$, is a UCQ rewriting for $q_\alpha, \mathcal{T}$.

Using the procedure described above we can compute the UCQ $u_\cup := \{q \cup q_\alpha, q \cup q^1_\alpha, q \cup q^2_\alpha\}$, which is indeed a sound UCQ rewriting for the query $q^+ := q \cup \{\alpha\}$. However, it is not complete; more precisely, any complete UCQ rewriting for $q^+, \mathcal{T}$ must contain the query $\{x \mid \{A(x)\}\}$; but, for all CQs $q' \in u_\alpha$ $\mathsf{avar}(q') \not\subseteq \mathsf{var}(q_1)$, hence q_1 is never used. ♢

In the previous example we observe that the missing query (q_1) does exist in the UCQ rewriting of the initial query, but it cannot be added to the target UCQ using the union operation. This suggests that there is probably another type of interaction between the UCQ rewritings that we should consider. More precisely, we observe that apart from points where the UCQ rewritings should be unified, there exist points where the UCQs should be 'merged'. For example, in the previous case we can observe that the formula implied by the CQ $q^2_\alpha \in u_\alpha$ is in some sense already 'contained in' $q \in u$. This represents a point where the two UCQ rewritings actually 'merge'. Hence, the construction of the UCQ rewriting of the extended query should proceed by copying q and all the CQs that are generated in the UCQ of the initial query 'after' q. Thus, in our previous example, q_1 should be copied (as is) to the computed UCQ. This in turn implies that the rewriting algorithm used to compute the UCQ rewriting of the initial query should keep track of the dependencies between the generated queries. As we will show in the next section, the aforementioned union (join) and merge operations are the two operations used to compute a UCQ rewriting of the extended query.

Another important open question is whether the above process can be performed using any computed UCQ rewriting for the initial query. Unfortunately, as the following example shows, this is not always possible. The problem is that optimisation techniques like subsumption checking, employed by many modern state-of-the-art query rewriting systems, can prune queries that are not going to be redundant in UCQ rewritings of the extended query.

Example 3. Consider the following TBox and CQ:

$$\mathcal{T} = \{A \sqsubseteq \exists R\} \quad q = \{x \mid \{A(x), R(x,y)\}\}.$$

The UCQ $u := \{q, q_1\}$, where $q_1 = \{x \mid \{A(x)\}\}$, is a UCQ rewriting for $q, \mathcal{T}$. However, q_1 subsumes q, hence q can be removed and the UCQ $\{q_1\}$ is also a UCQ rewriting for $q, \mathcal{T}$. Most modern systems are likely to return the latter UCQ rewriting.

Now suppose that we extend the original query by adding the new atom $B(y)$. Then, the new query is of the form $q^+ = \{x \mid \{A(x), R(x,y), B(y)\}\}$ and its UCQ rewriting consists of the set $\{q^+\}$. Unfortunately, it is not possible to compute this UCQ rewriting from the UCQ $\{q_1\}$. Intuitively, the problem is that query q_1, which is used to prune q, is no longer generated in the UCQ rewriting of the extended query; hence, in that context q is not redundant. A UCQ rewriting for $q^+, \mathcal{T}$ can, however, be generated from u (the UCQ without the subsumed query removed) and a UCQ rewriting for $q_\alpha = \{y \mid \{B(y)\}\}$, which consists of the UCQ $\{q_\alpha\}$. More precisely, from the union of $q \in u$ and $q_\alpha \in u_\alpha$ one obtains the query q^+. $\diamondsuit$

The previous example suggests that we should use an algorithm that does not employ such optimisation techniques. One such algorithm is the original PerfectRef algorithm. However, the absence of optimisation techniques compromises the practicality of the approach. More precisely, as has been shown by experimental evaluations [14], systems that do not use optimisations tend to compute very large UCQ rewritings. Hence, performing a pair-wise union of two large UCQ rewritings can be impractical. However, our intuition is that on the one hand, the UCQ rewriting of the atom-query is going to be rather small, while on the other hand, the two UCQ rewritings would have many 'merge' and few 'join' points, as the following example shows.

Example 4. Consider the following TBox and CQ:

$$\mathcal{T} = \{A_n \sqsubseteq A_{n-1}, \ldots, A_2 \sqsubseteq A_1, A_1 \sqsubseteq B, A_1 \sqsubseteq C\} \quad q = \{x \mid \{B(x)\}\}.$$

The set $u = \{q, q_1, \ldots, q_n\}$ where q_i is a CQ of the form $\{x \mid \{A_i(x)\}\}$ is a UCQ rewriting for $q, \mathcal{T}$. Now suppose that we extend the query with atom $C(x)$ obtaining the new CQ $q^+ = \{x \mid \{B(x), C(x)\}\}$. Following our previous discussion, we can compute a UCQ rewriting for $q^+, \mathcal{T}$ by combining u with a UCQ rewriting for $q_\alpha = \{x \mid \{C(x)\}\}$ w.r.t. $\mathcal{T}$, which in this case is the UCQ $u_\alpha = \{q_\alpha, q_1, \ldots, q_n\}$. However, after computing the union of $q \in u$ and $q_\alpha \in u_\alpha$ we immediately see that q_1 appears in both UCQ rewritings. Hence, at this point the two UCQs merge and all queries q_i with $1 \leq i \leq n$ can be copied to the final UCQ and can be discarded from further processing. $\diamondsuit$

Concluding our analysis and design, we present yet another technical issue in the construction of a correct algorithm.

Example 5. Consider the following TBox and CQ:

$$\mathcal{T} = \{A \sqsubseteq \exists R\} \quad q = \{x \mid \{R(x, y), R(z, y)\}\}.$$

The set $u = \{q, q_1, q_2\}$, where $q_1 = \{x \mid \{R(x, y)\}\}$ and $q_2 = \{x \mid \{A(x)\}\}$ is a UCQ rewriting for $q, \mathcal{T}$. Consider now the addition of the atom $\alpha = B(z)$. The atom-query for α w.r.t. q is the query $q_\alpha := \{z \mid \{B(z)\}\}$ and its UCQ rewriting is $u_\alpha := \{q_\alpha\}$. We can observe that the only query with which q_α joins is the query q, however, a UCQ rewriting for $q \cup \{\alpha\}$ must contain the queries $q'_1 = \{x \mid \{R(x, y), B(x)\}\}$ and $q'_2 = \{x \mid \{A(x), B(x)\}\}$. The issue is that query q_1 is produced from q by unifying $R(x, y)$ and $R(z, y)$ through a condensation step, and z, the common variable, is renamed to x. $\diamondsuit$

The above example suggests that in order to be able to compute a UCQ rewriting of an extended query from the UCQ rewriting of an initial one, the algorithm used to create the UCQ rewriting of the initial one should keep track of variable renamings preformed during the condensation step. If this is the case, then in the previous example, q_α can be joined with q_1 and q_2 in order to produce queries q'_1 and q'_2.

3.2 The UCQ Extension Algorithm

As detailed in the previous section, in order to produce a correct UCQ rewriting for an extended query, first and foremost, the algorithm that is used to compute the UCQ rewriting of the initial query must, on the one hand keep track of the dependencies between the generated queries while on the other hand, keep track of variable changes in the condensation step.

These changes are detailed in Algorithm 1, which presents ex-PerfectRef, an extended version of the standard PerfectRef algorithm. Unlike PerfectRef, ex-PerfectRef maintains a binary-relation G over queries. A pair $\langle q, q' \rangle$ is in G if q' is generated from q by an application of either a single reformulation or condensation step. Since G can contain cycles, ex-PerfectRef also extracts and returns a *hierarchy* out of the computed dependency relation—that is, for each cycle a representative query is selected and then a transitively-reduced strict partial order of all the representative elements is constructed. The formal definition of the hierarchy function is given next.

The function hierarchy. Let U be a set, let $K \subseteq U \times U$ be a binary relation over U and let S be a subset of U.

Algorithm 1 ex-PerfectRef$(q, \mathcal{T})$

```
    Input: A CQ $q$ and a DL-Lite$_R$-TBox $\mathcal{T}$
 1: Initialise a UCQ $u := \{q\}$
 2: Initialise a binary relation $G := \emptyset$
 3: Initialise a mapping $\mu$ from CQs to unifications and set $\mu(q) := \emptyset$
 4: repeat
 5:     $u' := u$
 6:     for all $q \in u'$ do
 7:         for all $\alpha \in q$ do
 8:             for all PI $I \in \mathcal{T}$ do
 9:                 if $I$ is applicable to $\alpha$ then
10:                     $q' := q[\alpha/\mathsf{gr}(\alpha, I)]$
11:                     $\mu(q') := \mu(q)$
12:                     $u := u \cup \{q'\}$;
13:                     $G := G \cup \{\langle q, q' \rangle\}$
14:                 end if
15:             end for
16:         end for
17:         for all $\alpha_1, \alpha_2$ in $q$ do
18:             if there is a most general unifier $\sigma$ for $\alpha_1$ and $\alpha_2$ then
19:                 $q' := \mathsf{reduce}(q, \alpha_1, \alpha_2)$
20:                 $\mu(q') := \mu(q) \cup \{\sigma\}$
21:                 $u := u \cup \{q'\}$
22:                 $G := G \cup \{\langle q, q' \rangle\}$
23:             end if
24:         end for
25:     end for
26: until $u' = u$
27: if $G = \emptyset$ then
28:     return $(\mathsf{hierarchy}(u, \{\langle q, \{\mathsf{var}(q) \mid \{\}\}\rangle\}), \mu)$
29: end if
30: return $(\mathsf{hierarchy}(u, G), \mu)$
```

– $D \in U$ is *reachable* in K from $C \in U$, written $C \rightsquigarrow_K D$, if $E_0, \ldots, E_n$ with $n \geq 0$ exist where $E_0 = C$, $E_n = D$ and $\langle E_i, E_{i+1} \rangle \in K$ for each $0 \leq i < n$.[4]
– A *hierarchy* of S w.r.t. K is a pair $\langle \mathcal{H}, \rho \rangle$ defined as follows:
  • Let $V \subseteq S$ be a minimal (w.r.t. set inclusion) set such that, it contains exactly one element from each set $\{C \mid C, D \in S,\ C \rightsquigarrow_K D$ and $D \rightsquigarrow_K C\}$. Then, $\mathcal{H}$ is the reflexive–transitive reduction of the relation $\{\langle C, D \rangle \in V \times V \mid C \rightsquigarrow_K D\}$.
  • $\rho : V \to 2^S$ is the function on V such that $D \in \rho(C)$ if and only if $C \rightsquigarrow_K D$ and $D \rightsquigarrow_K C$.
– $\mathsf{hierarchy}(S, K)$ is a function that returns one arbitrarily chosen but fixed hierarchy of S w.r.t. K.

[4] Note that, according to this definition, each $C \in U$ is reachable from itself.

Algorithm 2 ExtendRewriting$(q, \alpha, \mathcal{T}, \langle \mathcal{H}, \rho \rangle, \mu)$

```
   Input: A CQ $q$, an atom $\alpha$, a DL-Lite$_R$-TBox $\mathcal{T}$ and a hierarchy $\langle \mathcal{H}, \rho \rangle$ and
   mapping $\mu$ computed using Algorithm 1.
1: $u_\alpha :=$ PerfectRef$(\{\mathsf{var}(\alpha) \cap \mathsf{var}(q) \mid \{\alpha\}\}, \mathcal{T})$
2: Initialise a queue $Q$ with $Q := \{q_0\}$, where $q_0$ is the root in $\mathcal{H}$
3: $u := \emptyset$
4: while $Q \neq \emptyset$ do
5:     Remove the head $q_H$ from $Q$
6:     for all $q_{eq} \in \rho(q_H)$ do
7:         for all $q_\alpha \in u_\alpha$ do
8:             if isContainedIn$(q_\alpha, q_{eq})$ then
9:                 for all $q''$ such that $q_{eq} \rightsquigarrow_{\mathcal{H}} q''$ do
10:                    Add $q''$ and all CQs in $\rho(q'')$ to $u$
11:                end for
12:            else
13:                $\mu_{eq} := \mu(q_{eq})$
14:                if containsAllVars$(q_\alpha, q_{eq}, \mu_{eq})$ then
15:                    Add $\{\mathsf{avar}(q) \mid q_{eq} \cup (q_\alpha)_{\mu_{eq}}\}$ to $u$
16:                    Add to the end of $Q$ each $q''$ such that $\langle q_{eq}, q'' \rangle \in \mathcal{H}$
17:                end if
18:            end if
19:        end for
20:    end for
21: end while
22: $u := u \setminus \{\{\mathsf{var}(q) \mid \{\}\}\}$
23: return removeSubsumed$(u)$
```

Finally, the algorithm uses a mapping μ from CQs to variable mappings in order to keep track of the variable unifications that are conducted during the condensation step (Line 20). These are also copied to newly created CQs in the reformulation step (Line 11).

Having computed a UCQ rewriting for some query q and TBox $\mathcal{T}$ in the form of a hierarchy $\langle \mathcal{H}, \rho \rangle$ using Algorithm 1, and tracked variable unifications using μ, one can compute a UCQ rewriting for any extension of query q with an atom α. The algorithm uses the following functions to check that the two UCQ rewritings should be merged or to check using the variable mappings in μ computed by Algorithm 1 that two CQs can be joined (cf. Example 5).

The function isContainedIn. Let q, q' be two CQs. Then, isContainedIn(q', q) returns true if the CQ $\{\mathsf{avar}(q) \mid q \cup q'\}$ subsumes q; otherwise it returns false. The intuition is that all atoms in q' already exist in q.

The function containsAllVars. Let q, q' be two CQs and μ a set of variable mappings. Then, containsAllVars(q', q, μ) returns true if, for each $z \in \mathsf{avar}(q')$ there exists $x \in \mathsf{var}(q)$ such that, when considering the mappings in μ as a graph, we have $z \rightsquigarrow_\mu x$.

Algorithm 2 presents the algorithm in detail. The algorithm accepts as an input a CQ q, a new atom α, a DL-Lite$_R$-TBox $\mathcal{T}$ and a hierarchy $\langle \mathcal{H}, \rho \rangle$ and a mapping μ computed using Algorithm 1 and it returns a UCQ for the query $\{\mathsf{var}(q) \mid q \cup \{\alpha\}\}$. It first computes a UCQ rewriting u_α for the atom-query q_α of α w.r.t. q (Line 1). The UCQ rewriting for q_α explicates all implied information of the $\mathcal{T}$ about the atom α, which is essential for the correctness of the algorithm.

Having a UCQ rewriting for the initial query and the atom-query for α w.r.t. q, the algorithm proceeds in combining the UCQs; it uses a queue Q to perform a breadth-first search over the queries in $\mathcal{H}$ and either compute the union of the queries or copy queries from the UCQ rewriting of the initial query. More precisely, it picks a query q_H from Q, a query q_{eq} in the equivalence class $\rho(q_H)$ and a query q_α from the UCQ u_α. If $\mathsf{isContainedIn}(q_\alpha, q_{eq}) = \mathsf{true}$, then the two UCQs merge and hence, all queries q'' that are reachable in $\mathcal{H}$ from q_{eq} and all those queries in the equivalence class of q'' can be added to the target UCQ (Lines 9–11). Otherwise, the algorithm checks if the two CQs can be unified using function containsAllVars. If the function returns true then the union of the queries is sound, and is thus added to the target UCQ after appropriately renaming the variables of q_α if necessary; then, the successor query of q_{eq} in $\mathcal{H}$ is added to Q and the process continues. Finally, the algorithm applies subsumption checking in order to remove all the redundant queries and return a minimal UCQ rewriting for the extended query.

It can be shown that Algorithm 2 correctly computes a UCQ rewriting for an extended query, given a hierarchy computed using Algorithm 1.

4 Evaluation

We have developed a prototype tool for computing the rewriting of an extended conjunctive query based on Algorithms 2 and 1. Our implementation uses the implementation of PerfectRef that was developed and used in the experimental evaluation in [14].

We have compared our implementations with a number of available query rewriting systems. More precisely, our set of tools include the aforementioned implementation of PerfectRef,[5] Requiem [14], a resolution-based rewriting algorithm that uses subsumption to reduce the number of generated queries and Rapid [4], a recently developed highly-optimised DL-Lite$_R$ UCQ rewriting algorithm. For the evaluation we used the framework proposed in [14]. It consists of nine ontologies, namely V that captures information about European history,[6] P1 and P5 two hand-crafted artificial ontologies, S that models information about European Union financial institutions, U that is a DL-Lite$_R$ version of the well-known LUBM[7] ontology and A that is an ontology capturing information about abilities, disabilities and devices. Moreover, we also used the ontologies P5X, UX and AX that consist of normalised versions of the ontologies P5, U

[5] `http://www.cs.ox.ac.uk/projects/requiem/C.zip`
[6] `http://www.vicodi.org/`
[7] `http://swat.cse.lehigh.edu/projects/lubm/`

Table 1. Comparison between PerfectRef and ex-PerfectRef

$\mathcal{O}$	Q	PerfectRef		ex-PerfectRef		$\mathcal{O}$	Q	PerfectRef		ex-PerfectRef		$\mathcal{O}$	Q	PerfectRef		ex-PerfectRef	
		$\sharp u$	t	$\sharp u$	t			$\sharp u$	t	$\sharp u$	t			$\sharp u$	t	$\sharp u$	t
P1	1	2	1	2	4	S	1	6	2	6	4	V	1	15	4	15	6
	2	3	2	3	5		2	202	175	202	158		2	11	9	11	9
	3	7	9	7	9		3	1005	1113	1005	1109		3	72	43	72	38
	4	16	21	16	24		4	1548	945	1548	1920		4	185	82	185	121
	5	32	55	32	54		5	8693	8589	8693	23521		5	150	167	150	181
P5	1	14	4	14	6	U	1	5	3	5	5	A	1	783	561	783	277
	2	86	32	86	40		2	286	166	286	173		2	1812	1217	1812	710
	3	530	273	530	304		3	1248	479	1248	593		3	4763	1506	4763	2074
	4	3476	1576	3476	2663		4	5359	1628	5359	3919		4	7251	2336	7251	5288
	5	23744	26498	23744	188456		5	9220	4038	9220	14451		5	7885	347798	-	-
P5X	1	14	3	14	5	UX	1	5	3	5	4	AX	1	783	335	738	269
	2	86	34	86	38		2	286	187	286	154		2	1812	1249	1812	713
	3	530	282	530	301		3	1248	484	1248	587		3	4763	1403	4763	2089
	4	3476	1452	3476	2660		4	5358	1615	5358	3995		4	7251	2717	7251	5407
	5	23744	33248	23744	195478		5	9220	4041	9220	14513		5	7885	356756	-	-

and A. For each ontology, a set of five hand-crafted queries is proposed [14]. All experiments were conducted on a MacBook Pro with a 2.66GHz processor and 4GB of RAM with a time-out of 600 seconds.

In our first experiment we compared our extended ex-PerfectRef (i.e., Algorithm 1) against the standard implementation of PerfectRef. The goal is to assess the extent to which our extensions and changes affect the performance of the original algorithm.

Table 1 presents the results, where $\sharp u$ and t denote the size of the computed UCQ and the execution time (in milliseconds). As can be seen from that table, the performance of ex-PerfectRef is generally worse than that of PerfectRef. This was not surprising due to the extensions that have been applied to the original algorithm. This difference is relatively small in queries such as $Q1$–$Q4$, while it is usually more acute in query $Q5$. Notably, for query Q5 in ontologies A and AX, ex-PerfectRef failed to terminate within the set time-out.

In our second experiment, we evaluated Algorithms 1 and 2 against other query rewriting algorithms. In the case of our system, we proceeded as follows: for each of the test ontologies and for each of the queries Qi, $1 \leq i \leq 5$ we removed an arbitrary selected atom α to obtain a (hypothetical) initial query Qi^{-}. Then, we first run the method ex-PerfectRef$(Qi^{-}, \mathcal{T})$ to compute a UCQ rewriting for $Qi^{-}, \mathcal{T}$ in the form of a hierarchy $\langle \mathcal{H}, \rho \rangle$ together with the variable unification mappings μ, and then run the method ExtendRewriting$(Qi^{-}, \alpha, \mathcal{T}, \langle \mathcal{H}, \rho \rangle, \mu)$ to compute the UCQ rewriting for $Qi, \mathcal{T}$, as detailed in Algorithm 2. Since this process requires a query that contained at least two atoms, we did not consider query $Q1$ for some ontologies.

Table 2 presents the results from our second experiment. In that table, ex-PR refers to algorithm ex-PerfectRef executed for Qi^{-} and $\mathcal{T}$, Ref refers to Algorithm 2 without the final redundancy elimination step, while *sub* refers to that step (Line 23 of Algorithm 2). Hence, $\sharp u_{\star}$ and $t_{\star}$ denote the size of the computed UCQ and the execution time (in milliseconds) for the respective code $\star$. Also P-Ref refers to algorithm PerfectRef. Note that, after the final redundancy elimination

Table 2. Results of Algorithms 1 and 2 compared with other UCQ rewriting systems

$\mathcal{O}$	Q	Algorithms 1 & 2							P-Ref	Requiem	Rapid
		$\sharp u_{\text{ex-PR}}$	$\sharp u_{\text{Ref}}$	$t_{\text{ex-PR}}$	t_{Ref}	t_{sub}	$t_{\text{Ref}}+t_{sub}$	t_{all}			
	2	1	10	3	35	3	38	41	14	15	44
	3	3	72	4	82	45	127	131	124	63	66
V	4	37	185	30	39	95	134	164	274	173	116
	5	120	30	184	7	11	18	202	392	93	108
	2	2	2	5	2	1	3	8	4	6	10
	3	3	2	5	3	0	3	8	9	11	12
P1	4	7	2	11	3	0	3	14	31	27	17
	5	16	2	30	4	1	5	35	102	69	37
	2	14	10	5	5	2	7	12	40	26	15
	3	86	13	47	15	3	18	65	299	245	26
P5	4	530	18	322	35	1	36	358	1328	1131	39
	5	3476	32	2685	105	3	108	2793	26781	7722	104
	2	14	25	7	6	7	13	20	134	152	30
	3	86	79	59	17	35	52	111	630	1380	161
P5X	4	530	399	406	73	61	134	540	6327	4161	1230
	5	3476	2649	2911	225	770	995	3906	342133	93234	6533
	2	34	29	19	11	1	12	31	400	180	12
	3	193	33	255	15	4	19	274	1514	1095	16
S	4	404	57	464	20	5	25	489	1490	1142	15
	5	2296	154	2551	52	3	55	2606	28829	8202	18
	1	24	2	10	4	0	4	14	4	11	9
	2	41	18	19	10	0	10	29	427	158	9
U	3	180	8	161	11	1	12	173	650	256	11
	4	205	59	99	51	4	55	154	2133	1234	14
	5	225	59	179	19	6	25	204	6453	3307	18
	1	24	5	8	5	0	5	13	3	13	8
	2	41	20	18	10	1	11	29	471	212	8
UX	3	180	39	183	13	6	19	202	1169	778	20
	4	205	35	105	54	3	57	162	7677	6975	17
	5	225	113	215	19	30	49	264	20863	20466	33
	1	27	52	14	80	8	88	102	676	181	20
	2	783	71	305	22	7	29	334	1184	162	42
A	3	4763	104	2072	117	11	128	2200	1476	231	97
	4	783	323	313	95	80	175	488	2774	340	179
	5	4763	624	2141	262	195	457	2598	342897	576	316
	1	27	67	15	81	15	96	111	527	233	31
	2	783	1490	356	93	648	741	1097	2210	1561	1269
AX	3	4763	4752	11296	188	10428	10616	21921	9774	12097	2132
	4	783	3355	338	153	3451	3604	3942	16608	9140	2846
	5	4763	36013	11459	934	-	-	-	-	-	60492

step, all systems returned UCQ rewritings of the same size (the same as the ones reported in [14]) and hence the numbers are not presented.

As we can observe from the table, compared to PerfectRef, the process of extending the UCQ rewriting of a query (column $t_{\mathsf{Ref}}+t_{sub}$) is much more efficient than computing the UCQ rewriting of the new query from scratch (column for PerfectRef). Even more interestingly, even when considering Algorithms 1 and 2 together (i.e., t_{all}), the process is much more efficient than PerfectRef. In cases of queries containing a few atoms (usually queries Q1 and Q2) and having small rewritings (less than 30 queries), the total time is comparable, however in queries with large UCQ rewritings and large number of atoms, Algorithms 1 and 2 combined, manage to be several times and sometimes even 1 or 2 orders of magnitude faster than PerfectRef in computing the UCQ rewriting for $Qi, \mathcal{T}$. Such notable cases are queries Q3–Q5 in ontology P5 and P5X, all the queries in ontology S, queries Q3-Q5 in ontology U and UX, queries Q1, Q2, Q4 and Q5 in ontology A and finally queries Q1, Q2, and Q4 in ontology AX. An intuition behind this large improvement is that the brute-force (blind) application of the reformulation and condensation steps of PerfectRef is bound to be inefficient and not scale well in such cases. In our case though, Algorithm 1 first computes a UCQ rewriting for a *smaller* CQ (i.e., Qi) and then Algorithm 2 performs a much more guided breadth-first search, applying simple operations like set-union.

However, there are also two exceptions. Firstly, PerfectRef is faster in query Q3 ontology A. The reason is that the UCQ rewriting of the 'reduced' query Qi^- is much larger (4763 CQs) than the UCQ rewriting of Qi (104 CQs). That is, the extra atom in Qi helps PerfectRef stop computation earlier and compute the small target UCQ rewriting fast, while Algorithm 2 begins the refinement process with a large number of CQs most of which are not going to produce CQs for $Qi, \mathcal{T}$. Finally, like PerfectRef, Algorithm 2 failed to terminate in query Q5 ontology AX. The reason is that the size of the UCQ computed by the Ref part of the algorithm, i.e., $\sharp u_{\mathsf{Ref}}$, is quite large and the final redundancy elimination method fails to terminate within the set time-out.

Interestingly, a similar good behaviour for Algorithms 1 and 2 combined can be observed even when compared to the much more optimised system Requiem. There are a few cases that Requiem is more efficient, especially for ontology A which, as mentioned above, seems to be problematic for Algorithm 2, however, we can observe that in most cases the behaviour of Algorithms 1 and 2 is much more robust and scales better in queries with a UCQ rewriting of increasing size. Again, this is due to the guided nature of the refinement algorithm, while Requiem, although it uses subsumption internally to remove redundant queries, applies the resolution rule in an unguided brute-force way.

Finally, even when compared to Rapid, a highly optimised and DL-Lite$_R$-tuned algorithm, although Rapid is in most cases faster than the overall execution time of our strategy, there are several cases that the performance of the two algorithms is comparable. Actually, in ontology P5X and ontology A query 2, it manages to be notably faster than Rapid. Furthermore, when restricted only to the refinement step (Algorithm 2), algorithm manages to be even closer to the performance of Rapid.

5 Conclusion

In the current paper we studied the following problem: Given a query, a UCQ rewriting for the query and some atom, can we compute a UCQ rewriting for the query extended with the additional atom by "extending" the input UCQ rewriting without computing a UCQ rewriting of the new query from scratch?

We studied the problem at a theoretical level and investigated whether it is possible to compute such a UCQ rewriting from any given UCQ rewriting for the initial query. Our results showed that this is not possible in general, especially when optimisations are used to prune queries from the UCQ rewriting of the initial query. Hence, we designed our refinement algorithm by using the PerfectRef algorithm, which, in its original version, did not include any optimisations. Although it is commonly accepted that an unoptimised rewriting algorithm would compute large UCQ rewritings, and hence, compromise the practicality of our method, we continued by developing new optimisation strategies and a careful strategy for computing the refinement. Subsequently, we implemented the proposed algorithm and evaluated it experimentally, obtaining several encouraging and interesting results. On the one hand, the refinement process is much more efficient than computing the UCQ rewriting of the extended query from scratch using most (if not all) state-of-the-art rewriting algorithms. On the other hand, even when considering the overall time of computing the UCQ rewriting of the initial query together with the time for the refinement, the method was more efficient and robust compared to PerfectRef and Requiem, the latter of which also employs several optimisations.

There are many interesting challenges for future work. Firstly, one could study a similar problem under different types of query refinements, such as, after removing an atom or after adding and/or removing distinguished variables. Secondly, our initial relatively naive and preliminary algorithm is definitely open for further optimisations. More precisely, it is currently unknown whether some redundant queries from the UCQ rewriting of the initial query can actually be removed. Finally, investigating whether such an approach can also be applied to optimised systems such as Rapid or to more expressive DLs like $\mathcal{EL}$ and $\mathcal{ELHI}$ using systems such as Requiem are also interesting issues.

References

1. Artale, A., Calvanese, D., Kontchakov, R., Zakharyaschev, M.: The DL-Lite family and relations. Journal of Artificial Intelligence Research 36, 1–69 (2009)
2. Baader, F., Calvanese, D., McGuinness, D.L., Nardi, D., Patel-Schneider, P.F. (eds.): The Description Logic Handbook: Theory, Implementation, and Applications. Cambridge University Press (2003)

3. Calvanese, D., De Giacomo, G., Lembo, D., Lenzerini, M., Rosati, R.: Tractable reasoning and efficient query answering in description logics: The DL-Lite family. Journal of Automated Reasoning 39(3), 385–429 (2007)
4. Chortaras, A., Trivela, D., Stamou, G.: Optimized query rewriting in OWL 2 QL. In: Proceedings of the 23rd International Conference on Automated Deduction (CADE 23), Polland. pp. 192–206 (2011)
5. Demidova, E., Zhou, X., Nejdl, W.: A probabilistic scheme for keyword-based incremental query construction. IEEE Transactions on Knowledge and Data Engineering 99 (2011)
6. Glimm, B., Horrocks, I., Lutz, C., Sattler, U.: Conjunctive query answering for the description logic $\mathcal{SHIQ}$. In: Proc. of International Joint Conference on Artificial Intelligence (IJCAI 2007) (2007)
7. Gupta, A., Mumick, I.S., Ross, K.A.: Adapting materialized views after redefinitions. In: Proceedings of ACM SIGMOD International Conference on Management of Data. pp. 211–222 (1995)
8. Jansen, B.J., Spink, A., Blakely, C., Koshman, S.: Defining a session on web search engines: Research articles. Journal of the American Society for Information Science and Technology 58, 862–871 (2007)
9. Jansen, B.J., Spink, A., Pedersen, J.: A temporal comparison of altavista web searching: Research articles. Journal of the American Society for Information Science and Technology 56, 559–570 (2005)
10. Mohania, M.: Avoiding re-computation: View adaptation in data warehouses. In: In Proceedings of 8 th International Database Workshop. pp. 151–165 (1997)
11. Ortiz, M., Calvanese, D., Eiter, T.: Data complexity of query answering in expressive description logics via tableaux. Journal of Automated Reasoning 41(1), 61–98 (2008)
12. Pass, G., Chowdhury, A., Torgeson, C.: A picture of search. In: Proceedings of the 1st international conference on Scalable information systems (InfoScale 06). ACM (2006)
13. Pérez-Urbina, H., Motik, B., Horrocks, I.: Tractable query answering and rewriting under description logic constraints. Journal of Applied Logic 8, 186–209 (2009)
14. Pérez-Urbina, H., Horrocks, I., Motik, B.: Efficient query answering for OWL 2. In: Proceedings of the International Semantic Web Conference (ISWC 09). pp. 489–504 (2009)
15. Poggi, A., Lembo, D., Calvanese, D., De Giacomo, G., Lenzerini, M., Rosati, R.: Linking data to ontologies. Journal on Data Semantics X, 133–173 (2008)
16. Rosati, R., Almatelli, A.: Improving query answering over DL-Lite ontologies. In: Proceedings of the International Conference on Principles of Knowledge Representation and Reasoning (KR-10) (2010)
17. Tran, T., Cimiano, P., Rudolph, S., Studer, R.: Ontology-based interpretation of keywords for semantic search. In: Proceedings of the International Semantic Web Conference (ISWC 2007). vol. 4825, pp. 523–536 (2007)
18. Zenz, G., Zhou, X., Minack, E., Siberski, W., Nejdl, W.: From keywords to semantic queries-incremental query construction on the semantic web. Journal of Web Semantics 7, 166–176 (2009)

Sound and Complete SHI Instance Retrieval for 1 Billion ABox Assertions

Sebastian Wandelt, Ralf Möller

Institute for Software Systems,
Hamburg University of Technology
`wandelt@tuhh.de, moeller@tuhh.de`

Abstract. In the last years, reasoning over very large ontologies became more and more important. In our contribution, we propose a reasoning infrastructure, which can perform instance checking and instance retrieval over ontologies with semi-expressive terminological knowledge and large assertional parts.
The key idea is to 1) use modularizations of the assertional part, 2) use some kind of intermediate structure to find similarities between individual modules, and 3) store information efficiently on external memory. For evaluation purposes, experiments on benchmark and real world ontologies were carried out. We show that our reasoning infrastructure can handle up to 1 billion ABox assertions. To the best of our knowledge this is the first system to provide sound and complete reasoning over SHI ontologies with such a huge number of assertions.

1 Introduction

The Semantic Web is intended to bring structure to the meaningful content of web pages and to create an accessible environment for software agents. Ontologies are one way of representing the knowledge of these agents. The idea to represent datasets on the Internet with ontologies was first widely made public in [BLHL01]. Since then the Semantic Web became a widely used buzzword.

There is increased interest in the development of Semantic Web applications, e.g. digital libraries, community management, and health-care systems. As the Semantic Web evolves, the amount of data available in these applications is growing with an incredible speed. Since the size of the Semantic Web is expected to further grow in the coming years, scalability and performance of Semantic Web systems become increasingly important. Usually, such systems deal with information described in description-logic based ontology languages such as OWL [HKP$^+$09], and provide services for storing, querying, and updating large numbers of facts.

Decidability results for many expressive description logics and for query answering over these description logics have been shown. However, early tableau-based description logic reasoning systems, e.g. Racer [HMW04] and Pellet [SPG$^+$07], do not perform well with large ontologies since the implementation of tableau algorithms is built based on efficient main memory data structures.

There exists a lot of research to identify tractable description logics. For example the descriptions logic $\mathcal{EL}$ and extensions up to $\mathcal{EL}^{++}$, introduced in [BBL08], admit reasoning in polynomial time for classification and instance checking. Another lightweight description logic (family) is *DL-LITE* [CDGL$^+$05]. *DL-LITE* allows the use of relational database management systems for query answering. Another tractable fragment is the rule-based language OWL-R, introduced in [HKP$^+$09]. All tractable fragments have in common that the set of constructors in the ontology language is restricted in order to obtain efficient reasoning algorithms for query answering. However, in practical applications, users often need more expressive languages.

The increasing growth of Semantic Web applications also led to the development of a new class of external memory-based retrieval systems, so called triple stores. Originally motivated to store RDF schema information, see [Bec04], a general architecture to store triples was proposed in [BKvH03]. In the recent years, the number of these stores substantially increased, see for instance Franz AllegroGraph [Fra11] or OWLIM [Kir06]. Although the creators of triple stores continuously come up with more impressive performance evaluation results, there are two basic problems with these statistics. First, in general, it is not clear what kind of reasoning takes place inside the triple store during retrieval - it can be anything from pure lookup to complex description logic reasoning. Second, the hardware test configurations used by triple stores creators seem to be a little over the line. For instance, if one uses four computers with 48 GB of main memory each, then it is not a big surprise that the system is able to handle datasets in the order of several GB. This scenario seems to be at odds with the original intention of triple stores - managing data in external memory.

Another approach to overcome the problem of reasoning over large ontologies is to approximate the ontology by a more compact representation or in a weaker description logic. In [PTZ09], the authors propose to reuse the idea of knowledge compilation to approximate ontologies in a weaker ontology language. For the ontology language of their choice, i.e. *DL-LITE*, efficient query answering algorithms with polynomial data complexity exist. Reasoning on the approximated ontology allows to include/reject potential answers with respect to the original ontology. A similar direction was taken in [RPZ10], where the terminology part of an ontology is approximated to the description logic $\mathcal{EL}^{++}$. The results from the approximated ontology are used for more efficient classification over the original ontology. The classification results can then be used for more efficient retrieval as well.

Another approach focusing on reasoning over instances in large ontologies is presented in [TRKH08]. The algorithms in [TRKH08] are based on KAON2 [Mot08] algorithms, which transform the terminological part of an ontology into

Datalog [MW88]. Depending on the transformation strategy, the obtained Datalog program can be used for sound or complete reasoning over instances in the source ontology. The preceding approximation approaches rely on expressivity reduction of the ontology language.

A different approach is proposed in [DFK$^+$07], based on summarization and refinement. First, a summarization of the assertional part is created by aggregating individuals. This is part of a setup step that can be performed offline, i.e. before query answering takes place. Queries are then executed over the summarization. During the summarization process, one has to take care of inconsistencies. If the summarization leads to inconsistencies, previously merged individuals have to be broken up again.

In our work, we propose a system which can handle (i.e. perform sound and complete instance retrieval) more than 1 billion ABox assertions. For the syntax and semantics of the description logic $\mathcal{SHI}$ please refer to [Baa99]

2 Ingredients for Efficient Instance Retrieval

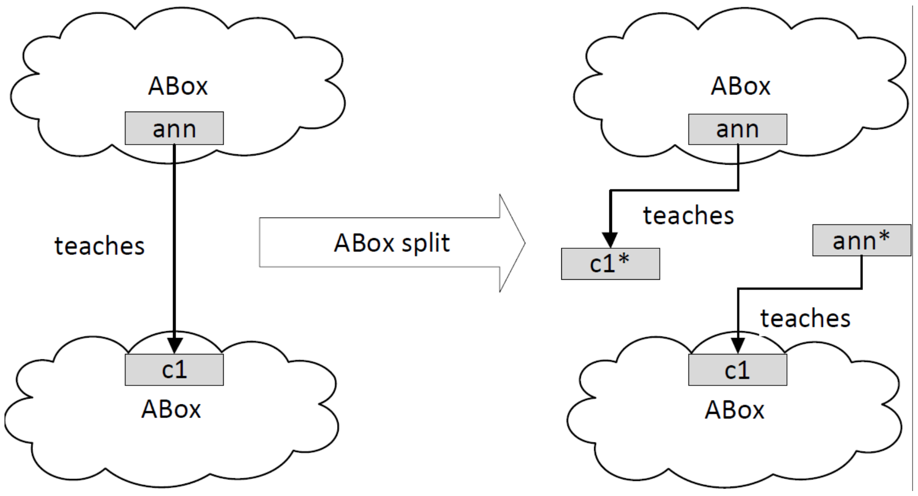
2.1 ABox Modularization

In [WM08], a method is proposed to identify the relevant information (assertions) to reason about an individual. The main motivation is to enable in-memory reasoning over large ontologies, i.e. ontologies with a large ABox, for traditional tableau-based reasoning systems.

Inspired by graph partitioning approaches, we developed techniques to break down an ABox into smaller chunks (modules), such that decision problems can be solved by considering these smaller parts only.

Naive modularization techniques (based on connectedness of individuals) are usually not sufficient for ABox modularizations, since most individuals are somehow connected to each other. We extend the naive modularization technique by introducing so-called *ABox splits*. Informally speaking, an ABox split breaks up a role assertion in an ABox, while preserving the semantics (this is formalized below). The idea is depicted in Figure 1. The clouds in Figure 1 indicate a set of ABox assertions. We split up the role assertion $teaches(ann, c1)$, create two new individuals (ann^* and $c1^*$), and keep the concept assertions for each fresh individual copy. After applying all possible ABox splits to an ABox of an ontology, a graph-based ABox modularization becomes more fine-grained, i.e. one obtains more (and smaller) modules.

Fig. 1 Intuition of an ABox split



Definition 1 **($\mathcal{SHI}$-splittability of Role Assertions).** *Given a $\mathcal{SHI}$-ontology $\mathcal{O} = \langle \mathcal{T}, \mathcal{R}, \mathcal{A} \rangle$ and a role assertion $R(a_1, a_2)$, we say that $R(a_1, a_2)$ is $\mathcal{SHI}$-*splittable with respect to $\mathcal{O}$ *if*

1. *there exists no transitive role R_2 with respect to $\mathcal{R}$, such that $\mathcal{R} \models R \sqsubseteq R_2$,*
2. *for each C that can be propagated over the role description R*
   - *$C = \bot$ or*
   - *there exists a concept description C_2, such that $C_2(b) \in \mathcal{A}$ and $\mathcal{T} \models C_2 \sqsubseteq C$ or*
   - *there exists a concept description C_2, such that $C_2(b) \in \mathcal{A}$ and $\mathcal{T} \models C \sqcap C_2 \sqsubseteq \bot$*

   and
3. *for each $C \in extinfo^{\forall}_{\mathcal{T},\mathcal{R}}(R^-)$*
   - *$C = \bot$ or*
   - *there exists a concept description C_2, such that $C_2(a) \in \mathcal{A}$ and $\mathcal{T} \models C_2 \sqsubseteq C$ or*
   - *there exists a concept description C_2, such that $C_2(a) \in \mathcal{A}$ and $\mathcal{T} \models C \sqcap C_2 \sqsubseteq \bot$.*

It can be shown that ABox splits are consistency preserving, if the role assertion is $\mathcal{SHI}$-splittable. The idea is that each tableau proof which makes use of propagated concept descriptions (via a $\forall$-tableau rule application) can be rewritten into a tableau proof without using the $\forall$-tableau rule application.

An individual island can be computed following the approach in [WM08]: given a start (root) individual, one can perform a graph search following all $\mathcal{SHI}$-unsplittable role assertions. This individual island is then sound and complete for instance checking w.r.t. the root individual.

2.2 One-Step nodes

We introduce a specialization of individual islands next. The basic idea is to define a notion of so-called pseudo node neighbors, which represent the directly asserted successors of a named individual in an ABox. Then, for each individual in the ABox, the information about all pseudo node successors plus the information about the original individual is combined, to obtain so-called one-step nodes. In addition to similarity detection, these one-step nodes can be used to answer instance checking and instance retrieval queries directly (always sound, and possible in a complete manner).

First, in Definition 2, we formally define a pseudo node successor for an individual with respect to an ABox.

Definition 2 (Pseudo Node Successor). *Given an ABox $\mathcal{A}$, a* pseudo node successor *of a named individual a is a pair $pns^{a,\mathcal{A}} = \langle \mathbf{rs}, \mathbf{cs} \rangle$, such that $\exists a_2 \in Ind(\mathcal{A})$ with*

1. $\forall R \in \mathbf{rs}.(R(a, a_2) \in \mathcal{A} \vee R^-(a_2, a) \in \mathcal{A})$,
2. $\forall C \in \mathbf{cs}.C(a_2) \in \mathcal{A}$, *and*
3. $\mathbf{rs}$ *and* $\mathbf{cs}$ *are maximal.*

Next, we combine all pseudo node successors of a named individual a in an ABox $\mathcal{A}$, the reflexive role assertions for a, and the directly asserted concepts of a, in order to create a summarization representative, called one-step node.

Definition 3 (One-Step Node). *Given an ontology $\mathcal{O} = \langle \mathcal{T}, \mathcal{R}, \mathcal{A} \rangle$ and an individual $a \in NInd(\mathcal{A})$, the* one-step node of a for $\mathcal{A}$, *denoted $osn^{a,\mathcal{A}}$, is a tuple $\langle \mathbf{rootconset}, \mathbf{reflset}, \mathbf{pnsset} \rangle$, such that $\mathbf{rootconset} = \{C | C(a) \in \mathcal{A}\}$, $\mathbf{reflset} = \{R | R(a, a) \in \mathcal{A} \vee R^-(a, a) \in \mathcal{A}\}$, and $\mathbf{pnsset}$ is the set of all pseudo node successors of individual a. The set of all possible one-step nodes is denoted* $\mathbf{OSN}$.

Definition 4 (One-Step Node Similarity). *Two individuals a_1 and a_2 are called* one-step node similar *for an ABox $\mathcal{A}$ if $osn^{a_1,\mathcal{A}} = osn^{a_2,\mathcal{A}}$.*

Every one-step node can be used for sound instance checking, since it represents a subset of the ABox assertions from the input ABox. It is clear that not every one-step node is complete for instance checking. However, in case the one-step node coincides with the individual island, then we can show that instance checking over the one-step node is even complete. For this, we define so-called splittable one-step nodes, for which each role assertion to a direct neighbor is $\mathcal{SHI}$-splittable.

Definition 5 (Splittable One-Step Node). *Given an ontology $\mathcal{O} = \langle \mathcal{T}, \mathcal{R}, \mathcal{A} \rangle$, an named individual a, and a one-step node $osn^{a,\mathcal{A}} = \langle \mathbf{rootconset}, \mathbf{reflset}, \mathbf{pnsset} \rangle$, we say that $osn^{a,\mathcal{A}}$ is* splittable *if for each $\langle \mathbf{rs}, \mathbf{cs} \rangle \in \mathbf{pnsset}$, a fresh individual $a_2 \notin Ind(\mathcal{A})$, and for each $R \in \mathbf{rs}$, the role assertion axiom $R(a, a_2)$ is $\mathcal{SHI}$-splittable with respect to ontology $\mathcal{O}_2 = \langle \mathcal{T}, \mathcal{R}, \mathcal{A}_2 \rangle$ with*

$$\mathcal{A}_2 = \{C(a) \mid C \in \mathbf{rootconset}\} \cup \{C(a_2) \mid C \in \mathbf{cs}\} \cup \{R(a, a_2)\}.$$

It is easy to see, that splittable one-step nodes can be directly used for sound and complete instance checking. Furthermore, two similar one-step nodes only need to be checked one time during instance retrieval. An example is shown below.

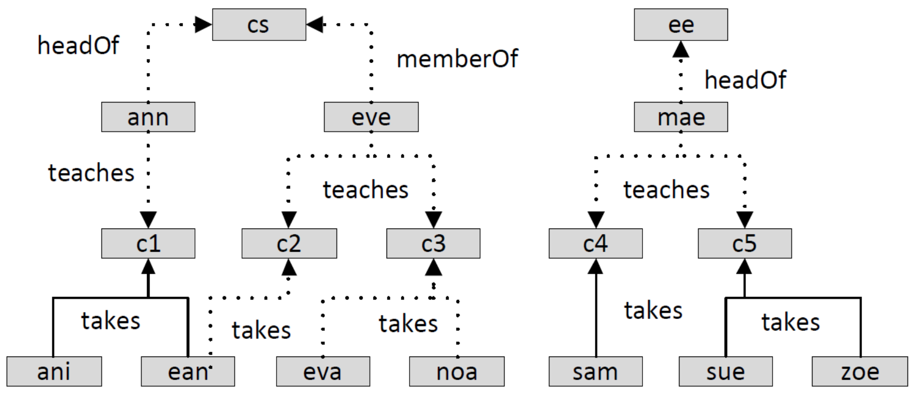
3 Example

In the following, we look at an example to discuss the optimization of instance checking and instance retrieval by the techniques introduced above.

Example 1 (Example Ontology for Island Reasoning). The example ontology $\mathcal{O}_{Ex1} = \langle \mathcal{T}_{Ex1}, \mathcal{R}_{Ex1}, \mathcal{A}_{Ex1} \rangle$ is defined as follows

$$\begin{aligned}
\mathcal{T}_{Ex1} = \{ & \\
& Chair \equiv \exists headOf.Department, Student \equiv \exists takes.Course, \\
& GraduateStudent \sqsubseteq \forall takes.GraduateCourse, \\
& UndergraduateCourse \sqcap Chair \sqsubseteq \bot, GraduateCourse \sqcap Chair \sqsubseteq \bot, \\
& UndergraduateCourse \sqsubseteq Course, GraduateCourse \sqsubseteq Course, \\
& Student \sqcap Chair \sqsubseteq \bot, \top \sqsubseteq \forall takes.Course \\
& \} \\
\mathcal{R}_{Ex1} = \{ & headOf \sqsubseteq memberOf, teaches \equiv isTaughtBy^- \}
\end{aligned}$$

Fig. 2 Individual relationships and splittability for Example 1



$$
\begin{aligned}
\mathcal{A}_{Ex1} = \{ & \\
& Department(cs), Department(ee), \\
& Professor(ann), Professor(eve), Professor(mae), \\
& UndergraduateCourse(c1), UndergraduateCourse(c4), \\
& UndergraduateCourse(c5), \\
& GraduateCourse(c2), GraduateCourse(c3), \\
& Student(ani), Student(ean), Student(eva), Student(noa), \\
& Student(sam), Student(sue), Student(zoe), \\
& headOf(ann, cs), memberOf(eve, cs), headOf(mae, ee), \\
& teaches(ann, c1), teaches(eve, c2), teaches(eve, c3), \\
& teaches(mae, c4), teaches(mae, c5), \\
& takes(ani, c1), takes(ean, c1), takes(ean, c2), takes(eva, c3), \\
& takes(noa, c3), takes(sam, c4), takes(sue, c5), takes(zoe, c5) \\
& \}.
\end{aligned}
$$

The relationships among individuals of $\mathcal{A}_{Ex1}$ are depicted in Figure 2. Please note that only role assertions are used to build the graph, since we only want to emphasize the relationship between the ABox individuals. $\mathcal{SHI}$-splittable role assertions are indicated with a dashed line. For instance, the role assertion $takes(ani, c1)$ is not $\mathcal{SHI}$-splittable because the concept description $GraduateCourse$ can be propagated via role description $takes$. Please note that all these role assertions would be $\mathcal{SHI}$-splittable if we had a disjointness axiom for $GraduateCourse$ and $UndergraduateCourse$. However, to show the behavior of reasoning in case of $\mathcal{SHI}$-unsplittability, we omitted the disjointness axiom here.

3.1 Instance Checking

For instance checking, we are given an ontology $\mathcal{O} = \langle \mathcal{T}, \mathcal{R}, \mathcal{A} \rangle$, an atomic concept description C, and an individual $a \in NInd(\mathcal{A})$, and we would like to find out, whether $\mathcal{O} \models C(a)$. The process of instance checking is done in two steps. First, we take the one-step node $osn^{a,\mathcal{A}}$ of individual a and check, whether $osn^{a,\mathcal{A}} \models C(a)$. If yes, then we are done, since we know that one-step nodes are sound for instance checking with respect to the input ontology $\mathcal{O}$. If $osn^{a,\mathcal{A}} \not\models C(a)$, then we distinguish two cases. First, if $osn^{a,\mathcal{A}}$ is splittable, then we know that we have $\mathcal{O} \not\models C(a)$. Otherwise, if $osn^{a,\mathcal{A}}$ is not splittable, then we load the individual island ISL_a for individual a and perform instance checking over ISL_a.

As an example for instance checking, we would like to check, whether the individual ann is an instance of concept description $Chair$ with respect to the ontology $\mathcal{O}_{Ex1}$. The one-step node $osn^{ann,\mathcal{A}_{Ex1}}$ is defined as follows:

$$\begin{aligned} osn^{ann,\mathcal{A}_{Ex1}} = & \langle \{Professor\}, \emptyset, \{\langle \{headOf\}, \{Department\} \rangle, \\ & \langle \{teaches\}, \{UndergraduateCourse\} \rangle \} \rangle. \end{aligned}$$

One possible one-step node realization of $osn^{ann,\mathcal{A}_{Ex1}}$ is

$$ABox(osn^{ann,\mathcal{A}_{Ex1}}) = \{Professor(ann), headOf(ann, a_1), teaches(ann, a_2)\}.$$

It is easy to see that we have $\langle \mathcal{T}, \mathcal{R}, ABox(osn^{ann,\mathcal{A}_{Ex1}}) \rangle \models Chair(ann)$, and thus we have $ABox(osn^{ann,\mathcal{A}_{Ex1}}) \models_{\mathcal{T}_{Ex1},\mathcal{R}_{Ex1}} Chair(ann)$ and by soundness of one-step node reasoning $\mathcal{O}_{Ex1} \models Chair(ann)$.

As a second example for instance checking, we would like to check, whether the individual $c1$ is an instance of concept description $Chair$ with respect to the ontology $\mathcal{O}_{Ex1}$. The one-step node $osn^{c1,\mathcal{A}_{Ex1}}$ is as follows:

$$\begin{aligned} osn^{c1,\mathcal{A}_{Ex1}} = & \langle \{UndergraduateCourse\}, \emptyset, \{\langle \{teaches^{-}\}, \{Professor\} \rangle, \\ & \langle \{takes^{-}\}, \{Student\} \rangle \} \rangle. \end{aligned}$$

One possible one-step node realization of $osn^{c1,\mathcal{A}_{Ex1}}$ is

$$ABox(osn^{c1,\mathcal{A}_{Ex1}}) = \{UndergraduateCourse(c1), teaches(a_1, c1,), takes(a_2, c1)\}.$$

It is easy to see that we have $\langle \mathcal{T}, \mathcal{R}, ABox(osn^{c1,\mathcal{A}_{Ex1}}) \rangle \not\models Chair(c1)$. In this case, the one-step node does not indicate entailment, and since $osn^{c1,\mathcal{A}_{Ex1}}$ is not a splittable one-step node, we should refer to the individual island of individual $c1$. However, another simple instance check can help us to avoid using the individual island here. It is easy to see that we have $\langle \mathcal{T}, \mathcal{R}, ABox(osn^{c1,\mathcal{A}_{Ex1}}) \rangle \models \neg Chair(c1)$, by disjointness of $UndergraduateCourse$ and $Chair$. And this means that we have $\mathcal{O}_{Ex1} \models \neg Chair(c1)$. Thus, in some cases, the "‘negated instance check"’ for one-step nodes can also help us to avoid performing reasoning on (more complex) individual islands. However, if the negated instance check fails, and the one-step node is unsplittable, then we really have to use sound and complete individual islands.

3.2 Instance Retrieval

In the following, we discuss instance retrieval optimization over ontologies. This is a direct extension of instance checking, by using one-step node similarity in addition. The first naive approach would be to apply instance checking techniques to each named individual in the ABox. For ontology $\mathcal{O}_{Ex1}$, we would have to perform 17 instance checks in that case. However, we have introduced the notion of one-step node similarity. The idea is that similar one-step nodes entail the same set of concept descriptions for the named root individual. Given the set of all one-step nodes for an input ontology, we can reduce the number of instance checks.

For example, assume that we would like to perform instance retrieval for the concept description $Chair$ with respect to ontology $\mathcal{O}_{Ex1}$. First, we retrieve the one-step node for each named individual in $\mathcal{A}_{Ex1}$. The resulting one-step nodes are shown below:

$$\begin{aligned}
osn^{ani,\mathcal{A}_{Ex1}} &= osn^{sam,\mathcal{A}_{Ex1}} = osn^{sue,\mathcal{A}_{Ex1}} = osn^{zoe,\mathcal{A}_{Ex1}} = \langle\{Student\},\emptyset,\\
&\quad \{\langle\{takes\},\{UndergraduateCourse\}\rangle\}\rangle\\
osn^{ean,\mathcal{A}_{Ex1}} &= \langle\{Student\},\emptyset,\\
&\quad \{\langle\{takes\},\{UndergraduateCourse\}\rangle,\langle\{takes\},\{GraduateCourse\}\rangle\}\rangle\\
osn^{eva,\mathcal{A}_{Ex1}} &= osn^{noa,\mathcal{A}_{Ex1}} = \langle\{Student\},\emptyset,\\
&\quad \{\langle\{takes\},\{GraduateCourse\}\rangle\}\rangle\\
osn^{c1,\mathcal{A}_{Ex1}} &= osn^{c4,\mathcal{A}_{Ex1}} = osn^{c5,\mathcal{A}_{Ex1}} = \langle\{UndergraduateCourse\},\emptyset,\\
&\quad \{\langle\{teaches^-\},\{Professor\}\rangle,\langle\{takes^-\},\{Student\}\rangle\}\rangle\\
osn^{c2,\mathcal{A}_{Ex1}} &= osn^{c3,\mathcal{A}_{Ex1}} = \langle\{GraduateCourse\},\emptyset,\\
&\quad \{\langle\{teaches^-\},\{Professor\}\rangle,\langle\{takes^-\},\{Student\}\rangle\}\rangle\\
osn^{ann,\mathcal{A}_{Ex1}} &= osn^{mae,\mathcal{A}_{Ex1}} = \langle\{Professor\},\emptyset,\\
&\quad \{\langle\{headOf\},\{Department\}\rangle,\langle\{teaches^-\},\{UndergraduateCourse\}\rangle\}\rangle
\end{aligned}$$

$$\begin{aligned}
osn^{eve,\mathcal{A}_{Ex1}} &= \langle \{Professor\}, \emptyset,\\
&\quad \{\langle \{memberOf\}, \{Department\}\rangle, \langle \{teaches^{-}\}, \{GraduateCourse\}\rangle\}\rangle\\
osn^{cs,\mathcal{A}_{Ex1}} &= \langle \{Department\}, \emptyset,\\
&\quad \{\langle \{headOf^{-}\}, \{Professor\}\rangle, \langle \{memberOf^{-}\}, \{Professor\}\rangle\}\rangle\\
osn^{ee,\mathcal{A}_{Ex1}} &= \langle \{Department\}, \emptyset,\\
&\quad \{\langle \{headOf^{-}\}, \{Professor\}\rangle\}\rangle.
\end{aligned}$$

Instead of 17 instance checks for 17 named individuals, we are left with 9 instance checks over 9 similar one-step nodes. For ontologies with a larger assertional part, similarity of one-step nodes reduces the number of instance checks usually by orders of magnitudes.

By performing instance checks for concept description $Chair$ over the 9 one-step nodes, we can conclude that individual ann and individual mae are instances of $Chair$. Additional instance checks for concept description $\neg Chair$ yields that $c1$, $c2$, $c3$, $c4$, $c5$, ani, ean, eva, noa, sam, sue and zoe are instances of concept description $\neg Chair$, and therefore are not instances of concept description $Chair$ if the input ontology is consistent. After one-step node retrieval, we are left to check three individuals for being an instance of concept description $Chair$, or not: cs, ee and eve. Usually, one would have to perform instance checks over the three individual islands. However, since the corresponding one-step nodes for these three individuals are splittable, we do not need to do any further checks, since the one-step nodes are already sound and complete for reasoning in $\mathcal{O}_{Ex1}$.

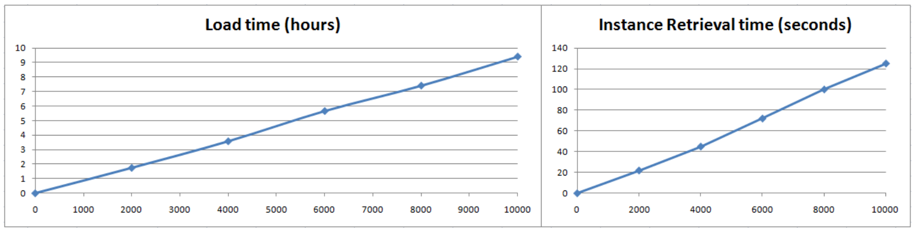
4 Implementation and Evaluation

In the following, we provide a general overview over the prototype. We have implemented the algorithms for reasoning over $\mathcal{SHI}$-ontologies using the programming language Java. We used the description logic reasoner Racer [HMW04] for our evaluation.

While TBox and RBox are kept in main memory, the ABox is serialized to a database. In our prototypical implementation, we used the relational database management system MySQL, see [WA02]. Apart from the assertional data, we also serialize the identifiers of one-step nodes for each individual and information about splittability of role assertions. For each serialized data structure, we have implemented caching algorithms, in order to avoid working on external memory directly for each update. During our experiments, a segmented least recently used cache, see for instance [KLW94], turned out to be most efficient.

We have used two benchmark ontologies for evaluation of our modularization techniques: one synthetic benchmark introduced in [GPH05] and a real world

Fig. 3 Load time and IR time for LUBM (up to 10000 universities)



multimedia annotation ontology used in the CASAM project and introduced in [GMN+09]. The results for both ontologies are outlined below.

4.1 LUBM

The Lehigh University Benchmark, short LUBM, is a synthetic ontology developed to benchmark knowledge base systems with respect to large OWL applications. The ontology is situated in the university domain. The background knowledge, i.e. the terminology, is described in a schema called Univ-Bench, see [GPH05] for an overview over the history, different versions and the predecessor Univ 1.0.

While the terminological part of LUBM is static, the assertional part is dynamic in size and can be generated as big as necessary/desired. The dataset we have used for our experiments was generated by the Univ-Bench artificial data generator.

We determined the number of ABox modules for different LUBM datasets. It turned out that most of the role assertions in LUBM can be broken up and the average module size (number of root individuals in the module) is 1.01.

Our next evaluation measure is load time, shown in Figure 3 on the left. The load time covers loading data from external memory (here: OWL files), applying the update algorithms and serializing the data to a database representation. We process the terminological part first and afterwards the assertional part is loaded. Please note that for 10000 universities the system has to deal with 1.380.000.000 ABox assertions.

In Figure 3, on the right-hand side, we show the instance retrieval time for the concept description *Chair* and different numbers of universities. It can be seen that the instance retrieval time is almost linear - even for more than 170 million individuals in LUBM(10000). Furthermore, we would like to emphasize that most of the instance retrieval time is spent by the database system to lookup the solution names on different pages in the data file. The actual description logic reasoning in our system is roughly constant for the number of universities. We conjecture that, if one finds a more sophisticated way to store the mapping between one-step nodes and individuals, instance retrieval times can be further reduced.

The space needed to store the data for 10000 universities is around 120 GByte (including all index structures).

4.2 CASAM Multimedia Content Ontology

We performed additional test with a real world ontology from the CASAM project. The project is focused on computer-aided semantic annotation of multimedia content. The novelty is the aggregation of human and machine knowledge. For a detailed discussion of the research objectives, see [GMN$^+$10], [PTP10], and [CLHB10]. Within the CASAM project, there is a need to define an expressive annotation language which allows for typical-case reasoning systems. The proposed annotation language is defined by the so-called Multimedia Content Ontology, short MCO, introduced in [GMN$^+$09]. Inspired by the MPEG-7 standard, see [IF02], strictly necessary elements describing the structure of multimedia documents are extracted. The intention is to exploit quantitative and qualitative time information in order to relate co-occurring observations about events in videos. Co-occurrences are detected either within the same or between different modalities, i.e. text, audio and speech, regarding the video shots.

Our tests show that all role assertions in the CASAM test ontology can be split up and therefore reasoning can be reduced to one-step nodes only. We do not provide diagrams for this ontology, since it is too small (only few thousand ABox assertions) and the time for reasoning can hardly be measured correctly.

5 Conclusions

The main goal of our work was to address the problem of instance retrieval over large ABoxes. We focused on the semi-expressive description logic $\mathcal{SHI}$, which can be seen as a first step towards more expressive description logics. We solve the given problem by applying ABox modularization techniques and using a compact representation of individual islands (modules). These compact representations can be used to group similar individuals together and handle them in one step during instance retrieval.

Our evaluation showed that we can handle more than one billion ABox assertions and perform sound and complete instance retrieval for $\mathcal{SHI}$-ontologies.

In the following, we would like to discuss interesting directions for future work.

An extension from the semi-expressive description logic $\mathcal{SHI}$ to $\mathcal{SHIQ}$ should be possible. We think that a syntactical analysis of the TBox and RBox can be used to identify a set of $\mathcal{SHIQ}$-unsplittable role assertions. Our homomorphism-based similarity criteria for individuals cannot be directly applied in the presence of cardinality restrictions. Further extensions, for instance to $\mathcal{SHOIQ}$, might be possible, but will surely require a lot of work and sophisticated analysis techniques.

Another direction for future work is the focus on more expressive query languages. While we focus on instance checking and instance retrieval, the next natural step is conjunctive query answering [GHLS07]. We think that query answering with respect to the class of grounded conjunctive queries, is straightforward. One would have to combine the results from sound (and complete reasoning) in order to identify possible variable bindings. The extension to standard conjunctive queries is without doubt much harder.

Since rules over ontologies have become more important recently, it would be interesting to implement a rule-based query answering engine on top of our ABox modularizations. We already performed first tests. By syntactical analysis of rule bodies we decided which individual islands have to be extended/merged. The first results are quite encouraging.

Finally, more comprehensive experimental studies are required. Recently published work [SCH10] on new data generation algorithms for synthetic test ontologies might be a good place to start from. In general, we believe that our results carry over to other ontologies. However there exist scenarios, especially extensive use of transitive roles, which make it much harder to find fine-grained ABox modularizations.

References

[Baa99] Franz Baader. Logic-Based Knowledge Representation. In *Artificial Intelligence Today*, pages 13–41. Springer-Verlag, 1999.

[BBL08] Franz Baader, Sebastian Brandt, and Carsten Lutz. Pushing the $\mathcal{EL}$ envelope further. In Kendall Clark and Peter F. Patel-Schneider, editors, *In Proceedings of the OWLED 2008 DC Workshop on OWL: Experiences and Directions*, 2008.

[Bec04] Dave Beckett. RDF/XML Syntax Specification (Revised). www.w3.org/TR/REC-rdf-syntax/, 2004.

[BKvH03] Jeen Broekstra, Arjohn Kampman, and Frank van Harmelen. Sesame: An Architecture for Storing and Querying RDF Data and Schema Information. In Dieter Fensel, James A. Hendler, Henry Lieberman, and Wolfgang Wahlster, editors, *Spinning the Semantic Web*, pages 197–222. MIT Press, 2003.

[BLHL01] Tim Berners-Lee, James Hendler, and Ora Lassila. The Semantic Web. *Scientific American*, 284(5):34–43, 2001.

[CDGL+05] Diego Calvanese, Giuseppe De Giacomo, Domenico Lembo, Maurizio Lenzerini, and Riccardo Rosati. DL-Lite: Tractable description logics for ontologies. In *Proc. of the 20th Nat. Conf. on Artificial Intelligence (AAAI 2005)*, pages 602–607, 2005.

[CLHB10] Chris Creed, Peter Lonsdale, Robert Hendley, and Russell Beale. Synergistic annotation of multimedia content. In *Proceedings of the 2010 Third International Conference on Advances in Computer-Human Interactions*, ACHI '10, pages 205–208, Washington, DC, USA, 2010. IEEE Computer Society.

[DFK+07] Julian Dolby, Achille Fokoue, Aditya Kalyanpur, Aaron Kershenbaum, Edith Schonberg, Kavitha Srinivas, and Li Ma. Scalable semantic retrieval through summarization and refinement. In *AAAI'07: Proceedings of the 22nd national conference on Artificial intelligence*, pages 299–304. AAAI Press, 2007.

[Fra11] Franz Inc. Allegrograph. http://www.franz.com/agraph/, 2011.

[GHLS07] Birte Glimm, Ian Horrocks, Carsten Lutz, and Uli Sattler. Conjunctive Query Answering in the Description Logic SHIQ. In *Proceedings of the 20th International Joint Conference on Artificial Intelligence (IJCAI 2007)*, 2007.

[GMN+09] O. Gries, R. Möller, A. Nafissi, K. Sokolski, and M. Rosenfeld. CASAM Domain Ontology. Technical report, Hamburg University of Technology, 2009.

[GMN+10] Oliver Gries, Ralf Möller, Anahita Nafissi, Maurice Rosenfeld, Kamil Sokolski, and Michael Wessel. A Probabilistic Abduction Engine for Media Interpretation Based on Ontologies. In Pascal Hitzler and Thomas Lukasiewicz, editors, *RR*, volume 6333 of *Lecture Notes in Computer Science*, pages 182–194. Springer, 2010.

[GPH05] Yuanbo Guo, Zhengxiang Pan, and Jeff Heflin. LUBM: A benchmark for OWL knowledge base systems. *J. Web Sem.*, 3(2-3):158–182, 2005.

[HKP+09] Pascal Hitzler, Markus Krötzsch, Bijan Parsia, Peter F. Patel-Schneider, and Sebastian Rudolph. OWL 2 Web Ontology Language Primer. W3C Recommendation, World Wide Web Consortium, October 2009.

[HMW04] V. Haarslev, R. Möller, and M. Wessel. Querying the Semantic Web with Racer + nRQL. In *Proceedings of the KI-2004 International Workshop on Applications of Description Logics (ADL'04), Ulm, Germany, September 24*, 2004.

[IF02] ISO/IEC15938-5FCD. Multimedia Content Description Interface (MPEG-7). http://mpeg.chiariglione.org/standards/mpeg-7/mpeg-7.htm, 2002.

[Kir06] Atanas Kiryakov. OWLIM: Balancing between scalable repository and light-weight reasoner. In *Proc. of WWW2006*, Edinburgh, Scotland, 2006.

[KLW94] Ramakrishna Karedla, J. Spencer Love, and Bradley G. Wherry. Caching strategies to improve disk system performance. *Computer*, 27(3):38–46, 1994.

[Mot08] Boris Motik. KAON2 - Scalable Reasoning over Ontologies with Large Data Sets. *ERCIM News*, 2008(72), 2008.

[MW88] David Maier and David Scott Warren. *Computing with Logic: Logic Programming with Prolog.* Benjamin/Cummings, 1988.

[PTP10] Katerina Papantoniou, George Tsatsaronis, and Georgios Paliouras. KDTA: Automated Knowledge-Driven Text Annotation. In José L. Balcázar, Francesco Bonchi, Aristides Gionis, and Michèle Sebag, editors, *ECML/PKDD (3)*, volume 6323 of *Lecture Notes in Computer Science*, pages 611–614. Springer, 2010.

[PTZ09] Jeff Z. Pan, Edward Thomas, and Yuting Zhao. Completeness Guaranteed Approximations for OWL-DL Query Answering. In Bernardo Cuenca Grau, Ian Horrocks, Boris Motik, and Ulrike Sattler, editors, *Description Logics*, volume 477 of *CEUR Workshop Proceedings*. CEUR-WS.org, 2009.

[RPZ10] Yuan Ren, Jeff Z. Pan, and Yuting Zhao. Soundness Preserving Approximation for TBox Reasoning. In Maria Fox and David Poole, editors, *AAAI*. AAAI Press, 2010.

[SCH10] Giorgos Stoilos, Bernardo Cuenca Grau, and Ian Horrocks. How Incomplete is your Semantic Web Reasoner? In *Proc. of the 20th Nat. Conf. on Artificial Intelligence (AAAI 10)*, pages 1431–1436. AAAI Publications, 2010.

[SPG+07] Evren Sirin, Bijan Parsia, Bernardo Cuenca Grau, Aditya Kalyanpur, and Yarden Katz. Pellet: A practical OWL-DL reasoner. *J. Web Sem.*, 5(2):51–53, 2007.

[TRKH08] Tuvshintur Tserendorj, Sebastian Rudolph, Markus Krötzsch, and Pascal Hitzler. Approximate OWL-reasoning with Screech. In Diego Calvanese and Georg Lausen, editors, *RR*, volume 5341 of *Lecture Notes in Computer Science*, pages 165–180. Springer, 2008.

[WA02] Michael Widenius and Davis Axmark. *MySQL Reference Manual*. O'Reilly & Associates, Inc., Sebastopol, CA, USA, 2002.

[WM08] Sebastian Wandelt and Ralf Möller. Island reasoning for ALCHI ontologies. In *Proceedings of the 2008 conference on Formal Ontology in Information Systems*, pages 164–177, Amsterdam, The Netherlands, 2008. IOS Press.

Handling Cyclic Axioms in Dynamic, Web-Scale Knowledge Bases

Yingjie Li and Jeff Heflin

Department of Computer Science and Engineering, Lehigh University
19 Memorial Dr. West, Bethlehem, PA 18015, U.S.A.
{yil308, heflin}@cse.lehigh.edu

Abstract. In recent years, there has been an explosion of publicly available Semantic Web data. In order to effectively integrate millions of small, distributed data sources and quickly answer queries, we previously proposed a tree structure query optimization algorithm that uses source selectivity of each query subgoal as the heuristic to plan the query execution and uses the most selective subgoals to provide constraints that make other subgoals selective. However, this constraint propagation is incomplete when the relevant ontologies contain cyclic axioms. Here, we propose an improvement to this algorithm that is complete for cyclic axioms, yet still able to scale to millions of data sources.

Keywords: Information Integration, Cyclic Axioms, Magic Sets, Equality Reasoning

1 Introduction

In recent years, there has been an explosion of publicly available distributed Semantic Web data. These data are often small (around 50 RDF triples per source), which is supported by the fact that many large data sets such as DBpedia, GeoNames and DBLP provide dereferenceable URIs for each of their instances; we treat each such URI as a lightweight source. In order to effectively integrate these Semantic Web data sources and answer queries, we proposed an inverted index mechanism and a complete non-structure query answering algorithm using this index [6]. Because this index only indicates whether URIs or literals are present in a document, the non-structure algorithm can not scale to the real Semantic Web. Therefore, we subsequently proposed a tree-structure query optimization algorithm that uses source selectivity of each query subgoal as the heuristic to plan the query execution by selecting a small subset of relevant sources from millions of possible sources [5]. This algorithm was designed for OWLII ontologies (the intersection of OWL with GAV and LAV rules) - a subset of OWL DL (Description Logic) [10]. However, it is incomplete when cyclic axioms are considered because it does not load all relevant sources that correspond to a query subgoal, but instead only loads those that contain the subgoal predicate and its available variable constraints. On the other hand, each iteration in the cyclic axiom could generate recursive variable constraints that can be propagated into the following iterations. Consequently, without the fix point computation of cyclic axioms, the tree-structure algorithm will miss those sources collected by applying the recursive variable constraints in each iteration. Therefore, in order to guarantee completeness, we need special treatment for cyclic axioms. Furthermore, this process should be dynamic because data on the web is constantly in flux.

In DL, a cyclic axiom is one that references the same (or equivalent) classes (or properties) on both sides of the subsumption relation. Such an axiom may be explicit or inferred. For instance, $\exists P.C \sqsubseteq C$ is a cyclic axiom, where C is a class and P is a property. $P \circ P \sqsubseteq P$ is also a cyclic axiom, where P actually stands for a transitive property. Note, $rdfs{:}subClassOf$, $rdfs{:}subPropertyOf$, *owl*:*equivalentClass* and *owl*:*equivalentProperty* are not cyclic unless they are used to define a class/property to be a subclass/subproperty of itself. To the best of our knowledge, no one has calculated how many cyclic axioms are used in real world ontologies. However, Wang et al. [13] surveyed 1275 ontologies and found 39 (3%) that contained a transitive property. Instance coreference is a special case of cyclic axiom because *owl*:*sameAs* is a ubiquitous transitive property as defined: $owl{:}sameAs \circ owl{:}sameAs \sqsubseteq owl{:}sameAs$. The Billion Triple Challenge 2010 data set has 6,932,678 URI resources connected by 8,711,398 unique *owl*:*sameAs* statements [2]. The graph made of these *owl*:*sameAs* statements consists of 2,890,027 weakly connected components. Most components are pairs of nodes joined by *owl*:*sameAs* links. This observation implies that the typical *owl*:*sameAs* network is small but not ignorable.

Although handling cyclic axioms is routine for typical inference algorithms, they present challenges when querying large distributed Knowledge Bases (KBs). Some related but different work has been proposed. Pan et al. described a fix point computation algorithm for cyclic axioms in DLDB3 [9]. Mei et al. discussed the fix point computation of cyclic axioms on ontology query answering over databases [8]. Urbani et al. proposed a MapReduce reasoner to deal with the fix point computation of *owl*:*sameAs* triples [12]. Qasem et al. presented an extended GNS algorithm to handle instance coreference [11]. Their main drawback is that they all precompute (lacking flexibility) rather than dynamically compute the fix point of cyclic axioms for centralized KBs as opposed to large distributed KBs. In addition, Lam et al. [4] proposed an approach to blocking the expansion of cyclic axioms in order to guarantee termination of the computation during their cycle handling. However, this process aims to resolve the unsatisfied ontology but not for query answering over large distributed KBs. Another important approach of handling cyclic axioms is *Magic Sets* [1], which is a general algorithm for rewriting logical rules to compute the fix point of cyclic axioms. It applies the sideways information passing strategy (SIPS) and improves query answering efficiency by restricting the computation to facts that are related to the query. Since SIPS is basically similar to our constant constraint propagation, we incorporate the *Magic Sets* theory into our algorithm. More details will be given in Sections 2 and 3. Therefore, inspired by the traditional *Magic Sets* theory and based on the tree-structure algorithm [5], we propose a dynamic cyclic axiom handling algorithm for query answering over large distributed KBs. Our main contributions are: 1) we develop a dynamic stack-based cyclic axiom handling algorithm, which dynamically computes the fix point of cyclic axioms and does instance equality reasoning (*owl:sameAs* inference) in a separate process, 2) we demonstrate that our algorithm can perform well on both a synthetic data set with 20 ontologies having significant heterogeneity and a real world data set with 73,889,151 triples distributed among 21,008,285 documents.

The remainder of the paper is organized as follows: Section 2 describes some preliminary work. In Section 3, we describe the cyclic axiom handling algorithms for large distributed KBs. Section 4 presents the experiments that we have conducted to evaluate the proposed algorithms. Finally, in Section 5, we conclude and discuss future work.

2 Preliminaries

In this section, we first introduce some background and the main drawback of our tree-structure algorithm [5]. Then, we will describe the traditional *Magic Sets* theory [1], especially the part related to our algorithm.

2.1 Tree-structure algorithm

In the Semantic Web, there exist many ontologies, which can contain classes, properties and individuals. We assume that the assertions about the ontologies are spread across many data sources, and that mapping ontologies are defined to align the classes and properties of the domain ontologies. For convenience of analysis, we separate ontologies (i.e. the class/property definitions and axioms that relate them) and data sources (assertions of class membership or property values). Formally, we treat an ontology as a set of axioms and a data source as a set of RDF triples. A collection of ontologies and data sources constitute a *semantic web space*:

Definition 1. *(Semantic Web Space) A Semantic Web Space SWS is a tuple* $\langle D, o, s\rangle$*, where* D *refers to the set of document identifiers,* o *refers to an ontology function that maps* D *to a set of ontologies and* s *refers to a source function that maps* D *to a set of data sources.*

We have chosen to focus on *conjunctive queries*, which provide the logical foundation of many query languages (SQL, SPARQL, Datalog, etc.). A conjunctive query has the form $\mathrm{Q}\langle\overline{X}\rangle \leftarrow B_1\left(\overline{X}_1\right) \wedge \ldots \wedge B_n\left(\overline{X}_n\right)$ where each variable appearing in $\langle\overline{X}\rangle$ is called a distinguished variable and each $B_i(\overline{X}_i)$ is a query triple pattern (QTP) $\langle s_i, p_i, o_i\rangle$, where s_i is a URI or variable, p_i is a predicate URI, and o_i is a literal, URI, or variable.

Our problem is, given a SWS, how do we *efficiently* answer a conjunctive query? The key point to this problem is how to prune sources that are clearly irrelevant and focus on those that might contain useful information for answering the query. We have shown that a term index could be an efficient mechanism for locating the documents relevant to queries over distributed and heterogeneous semantic web resources [6]. Based on the term index, we proposed an effective tree-structure algorithm [5]. Given a rule-goal tree that aims to encapsulate all possible ways the required information could be represented in the sources [3] by using the axioms in the domain ontologies and the mapping ontologies, our tree-structure algorithm uses a bottom-up process to select sources and the selectivity of each goal node as a heuristic to greedily optimize and plan the query execution. The source selectivity of a selection procedure *sproc* for a query α is defined to be the number of sources not selected divided by the total number of sources available:

$$Sel_{sproc}(\alpha) = \frac{|D| - |sproc(\alpha)|}{|D|} \tag{1}$$

The tree-structure algorithm always starts with the most selective QTP, incrementally loads the relevant sources, and uses the data from the sources to further constrain related $QTPs$ in order to answer queries. It is only complete for acyclic ontologies expressed in OWLII defined below:

Definition 2. *The syntax of OWLII consists of DL axioms of the forms* $C \sqsubseteq D$, $A \equiv B$, $P \sqsubseteq D$, $P \equiv Q$, $P \equiv Q^-$, *where* C *is an* L_a *class,* D *is an* L_c *class,* A, B *are* L_{ac} *classes and* P, Q *are properties.* L_{ac}, L_a *and* L_c *are defined as:*

- L_{ac} *is a DL language where* A *is an atomic class and* i *is an individual. If* C *and* D *are classes and* R *is a property, then* $C \sqcap D$, $\exists R.C$ *and* $\exists R.\{i\}$ *are also classes.*
- L_a *includes all classes in* L_{ac}*. Also, if* C *and* D *are classes then* $C \sqcup D$ *is also a class.*
- L_c *includes all classes in* L_{ac}*. Also, if* C *and* D *are classes then* $\forall R.C$ *is also a class.*

In the presence of cyclic axioms, the tree-structure algorithm becomes incomplete because the cyclic axioms require that the goal node (e.g. the coreferenced instance for *owl:sameAs*) be iterated over to collect sources until a fix point is reached in order to obtain the complete answers. For instance, take the cyclic Datalog axiom $ancestor(?x, ?y)$:- $ancestor(?x, ?z) \wedge ancestor(?z, ?y)$ [1] and its query $ancestor(John, ?y)$. Assuming we have collected sources containing the substitutions $\{?z/Bob, ?y/Andy\}$ by using the subgoals $ancestor(John, ?z)$ and $ancestor(?z, ?y)$ respectively on the term index, the tree-structure algorithm then finishes processing this axiom because all of its subgoals have been handled and their corresponding sources have also been collected. However, those sources containing the recursive descendants of *Bob* and *Andy* are still relevant but will be missed because the given axiom is not recursively applied. Consequently, the tree-structure algorithm is clearly incomplete.

[1] The property composition axiom is actually beyond OWLII's expressivity and its use in the paper is for the purpose of giving an example. *owl:sameAs* is specially handled in Section 3.2.

2.2 Magic Sets

The *Magic Sets* method executes a top-down evaluation of a query by adding rules which narrow the computation to what is relevant for answering the query. As mentioned in section 1, it applies the SIPS strategy that describes how bindings passed to a rule's head by unification are used to evaluate the predicates in the rule's body. For instance, let V be an atom that has not yet been processed, and Q be the set of already considered atoms, then a SIPS specifies a propagation $V \rightarrow_X Q$, where X is the set of the variables bound by V, passing their values to Q.

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

(1) Rule adornment: this phase is to materialize, by suitable adornments, binding information for predicates. These are strings of the letters b and f, denoting bound or free for each argument of a predicate. First, adornments are created for query predicates. The adorned query is $ancestor^{bf}(John, Y)$. In the given rule, $ancestor^{bf}(John, Y)$ passes its binding information to $ancestor(X, Z)$ by $ancestor^{bf}(X, Y) \rightarrow_X ancestor(X, Z)$. Then, $ancestor(X, Z)$ is adorned $ancestor^{bf}(X, Z)$. Now, we consider $ancestor(Z, Y)$, for which there is no binding information and we can still use the given axiom to expand it. Finally, we have two resulting adorned rules: $ancestor^{bf}(X, Y)$:- $ancestor^{bf}(X, Z)$, $ancestor^{ff}(Z, Y)$ and $ancestor^{ff}(Z, Y)$:- $ancestor^{ff}(Z, W)$, $ancestor^{ff}(W, Y)$, where W is a new introduced variable.

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

(3) Rule modification: the adorned rules are modified by including magic atoms generated in Step (2) in the rule bodies. The resultant rules are called modified rules. For each adorned rule whose head is h, we extend the rule body by inserting $magic(h)$. In our example, $ancestor^{bf}(X, Y)$:- $magic\_ancestor^{bf}(X, Y)$, $ancestor^{bf}(X, Z)$, $ancestor^{ff}(Z, Y)$ and $ancestor^{ff}(Z, Y)$:- $magic\_ancestor^{ff}$ (Z, Y), $ancestor^{ff}(Z, W)$, $ancestor^{ff}(W, Y)$ are generated.

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

The complete rewritten program consists of the magic, modified, and query rules. Given a non-disjunctive datalog program P, a query Q, and the rewritten program $P^{'}$, it is well known that P and $P^{'}$ are equivalent w.r.t. Q [1]. In

Magic Sets, the adornments of Step (1) aim to cover all possible bound/free information based on the given query and rules. Then, the generated magic rules in the following steps can easily avoid irrelevant facts while guaranteeing completeness during the fix point computation of the cyclic axioms. For our tree-algorithm, as shown by [5], the constant propagation mechanism is basically the same as the SIPS strategy, using the available binding of the rule's head to constrain the rule's body. In addition, because our purpose is to collect relevant sources by constructing boolean queries using the available constant constraint (bound value) and the predicate instead of the real computation of the fix point, which is actually accomplished by the **Reasoner**, it is sufficient for us to only incorporate the rule adornment step into our algorithm. Then, we can easily detect if two terms have the same predicate and adornment. If so, a cycle is formed and we can collect only those sources that are necessary for this cycle's fix point computation.

3 Cyclic Axiom Handling Algorithms

In this section, we first introduce the *Magic Sets*-inspired dynamic cyclic axiom handling algorithm without instance coreference. Then, we discuss the equality reasoning (instance coreference).

3.1 Cyclic Axiom Handling

To handle cyclic axioms, there are four key points we particularly need to take care of:

- How to represent and annotate cyclic axioms in the original rule-goal tree of the query reformulation?
- Within each iteration of one cyclic axiom, how to compute the new generated substitutions of the given cyclic axiom that will be passed into the next iteration? In this process, we call the set of new substitutions *Relevant Substitutions (RS)*.
- How to apply the *RS* into the selection of relevant sources by using the term index?
- In case of multiple cyclic axioms mutually nested in one query, how to identify their correct computation order?

For the first point, as the traditional *Magic Sets* theory does, we adorn the cyclic axioms by using their binding information. Then, we mark them in the rule-goal tree. In theory, if one goal node G is detected to be one that can be unified with its one ancestor goal node A on condition that G and A are the same predicate and have the same adornments, then we detect a cycle C starting with A and ending with G. However, in practice, we apply the heuristic that is if A and G also have the same bound value, then they are not a cycle because A and G collect the same sources by using the term index and there is no recursive source collection. For instance, in Fig. 1, even though G_1 and G_2 compose a cycle, we can skip it because both of them only collect sources containing $John$ and $ancestor$. Formally, a cycle C is denoted as $C(A, G)$, where A is C's starting node and G is C's ending node. After the cycle is marked, the rule-goal tree is transformed into a rule-goal graph and each cyclic axiom will be converted into one or more rule-goal graph cycles correspondingly. Essentially, a rule-goal graph cycle means its corresponding axiom will be iteratively executed until a fix point is reached. For the second point, in the rule-goal graph, the RS of each iteration for one cyclic axiom essentially consists of the new generated substitutions of the cycle distinguished variables (CDVs) of this cyclic axiom's rule-goal graph cycles. We define each graph cycle's CDVs to be a set of the distinguished variables of the starting node of this cycle. In the previous example, C's CDVs contains all A's distinguished variables. At the end of each cycle iteration, we will compute the RS by asking the reasoner and apply it into the next iteration if the new RS is not empty. Otherwise, it means we have reached the fix point of the current cycle. Furthermore, if the RSs of all cycles in the rule-goal graph for one given cyclic axiom are empty, it means the fix point of this cyclic axiom has been reached. For the third point, we use the conjunction of each value in the RS and the goal predicate to query the term index. This helps us to significantly reduce the number of potentially relevant sources because of the constant constraints. For the fourth point, we will employ a cycle stack to plan the cyclic axiom handling sequence. Each cycle can be pushed onto the stack only if it is not already in the stack. Otherwise, we will postpone its processing.

We begin with the cyclic axiom $ancestor(?x, ?y)$:- $ancestor(?x, ?z) \wedge ancestor$ $(?z, ?y)$ and its query $ancestor(John, ?y)$ to introduce our algorithm. Fig. 1 shows its rule-goal graph. The back arrow means a cycle is marked. Each goal node has associated adornments (bf or ff) and selectivity (the number of relevant sources).

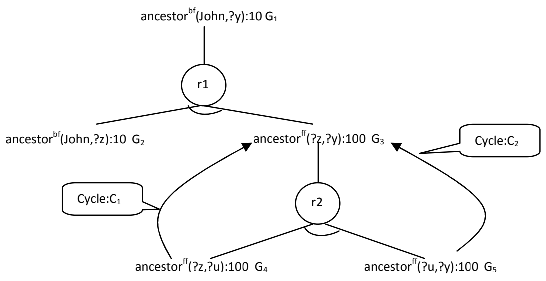

ancestor^bf(John,?y):10 G1
r1
ancestor^bf(John,?z):10 G2
ancestor^ff(?z,?y):100 G3
Cycle:C2
Cycle:C1
r2
ancestor^ff(?z,?u):100 G4
ancestor^ff(?u,?y):100 G5

Fig. 1. An example cyclic axiom

At the beginning, using the term index, each goal node of the rule-goal graph is initialized with their respective selectivities and bindings. In this example, we have two cycles: $C_1(G_3, G_4)$ and $C_2(G_3, G_5)$. In our cycle detection, if a goal node is an ending node of some cycle, we will say a cycle is detected. We use

S to stand for our cycle stack. Initially, we start with the most selective node G_2 and use its substitutions to constrain its sibling G_3. In this process, we start with G_3's most selective node G_4 (Here, G_4 and G_5 have the same selectivity and we randomly select G_4), where C_1 is detected and pushed onto S. Then, we still start with G_4 in processing C_1. Now, C_1 is detected again and postponed because it is also already in S. We evaluate G_4 and apply its available constant substitutions into its sibling G_5, where C_2 is detected and pushed onto S. Now, S contains C_1 and C_2. We then start to process C_2 still beginning with G_4, where C_1 is detected and postponed again. Then, we evaluate G_4 and apply its substitutions to G_5, where C_2 is detected again and postponed. Now, we are at the end of one iteration of C_2, compute C_2's RS and apply it into the next iteration to select relevant sources. If the new RS is empty, it means C_2's fix point has been reached and C_2 is popped. Now, S only contains C_1. Then, we go back to the context of C_1. Obviously, in processing the next iteration of C_1, C_2 will be met again. The previous process of C_2 is repeated. Meanwhile, at the end of each C_1's iteration, we also compute C_1's RS and apply it into the next iteration to collect sources until the new RS is empty meaning C_1's fix point has been reached. Finally, we finish processing C_1 and C_2 and correspondingly collect all relevant sources of the given cyclic axiom.

Note, in the above process, $C_1(G_3, G_4)$ and $C_2(G_3, G_5)$ are actually redundant cycles because they collect the same data sources. Therefore, we need to avoid such repeated source collections. In our algorithm, we detect if two cycles are redundant, which means that each node in one cycle exactly has the same predicate and same adornments as a node in the other one and vice versa. These redundant cycles are categorized into different redundant cycle classes and stored into a structure called the Redundant Cycle Base (RCB). Each redundant cycle class is a set of cycles that are redundant to each other. Redundant cycles cause redundant source collection because they could generate the same recursive constants and then collect the same sources multiple times. Therefore, during the process of each redundant cycle, we need to check if the new recursive constant has been used by other cycles that are redundant with the current cycle. If not, we go ahead and start the next generation. Otherwise, we will skip this constant. Here, we cannot handle only one cycle instead of the whole set of redundant cycles because redundant cycles could appear in different positions of the rule-goal graph and they thus could have different recursive constants generated to collect different data sources. Then, even though $C_1(G_3, G_4)$ and $C_2(G_3, G_5)$ are both pushed onto our cycle stack in the given example, we can avoid the repeated source collection. In addition, for those instances that match query constants or that are used as join conditions, we will compute their equivalence (*owl*:*sameAs*) closure on the fly by calling our equality reasoning algorithm (Fig. 3). More details can be found in section 4.2.

The pseudo code for our cyclic axiom handling algorithms is shown in Fig. 2. The bold lines in Alg. 1 and Alg. 2 and the whole Alg. 3 are new results from this paper. In Alg. 1, RCB stands for Redundant Cycle Base defined in the previous paragraph. EKB is a structure that collects and organizes equivalence

```
Algorithm 1 Source selection
function getSourceList(rgraph, $rs$, q) returns a list of sources
  inputs: rgraph, a given rule-goal graph (cyclic or non-cyclic)
          $rs$, a list of substitutions
          q, a list of query triple patterns
1:  Let frontier = leaf nodes or cycle ending nodes, static EKB = ϕ, static RCB = ϕ
      srcs[] = array of sets of sources, indexed by goal nodes
2:  for each goal node n in rgraph do
3:      if n has constant C and $C.equivalenceClass \notin EKB$ then
4:          computeSameAs({C}, EKB)
5:      for each $\theta \in rs$ do
6:          srcs[n] = qsources($n\theta$, EKB)
7:  do
8:      Let n = $min_{node \in frontier}(|srcs[node]|)$, p = n.parent
9:      if n is a cycle ending node AND $n.cycle \notin CycleStack$ then
10:         update(n.cycle, RCB)
11:         push(CycleStack, n.cycle)
12:         $srcs[n] = srcs[n] \cup$ getCyclicSourceList($n.cycle, rs, q$)
13:         pop(CycleStack)
14:     if n is a child of an AND rule node r then
15:         $srcs[p] = srcs[p] \cup$ OptimizeANDNode($rgraph,$
                    $n, siblings\ of\ n, srcs, q, EKB, RCB$)
16:     else
17:         $srcs[p] = srcs[p] \cup srcs[n]$
18:         if n is a child of rgraph.root and rgraph is a cycle then
19:             load(srcs[n], KB)
20:             Let $rsc$ = askReasoner (KB, rgraph)
21:             Let insts = extractJoinInsts(rsc)
22:             computeSameAs(insts, EKB)
23:             rgraph.RS = computeRS(rgraph.CDVs, RCB)
24:     remove n and its siblings from frontier
25:     if p has no descendants on frontier then
26:         add p to frontier
27: while ($frontier \neq \{rgraph.root\}$)
28: return srcs[rgraph.root]
```

```
Algorithm 2 Node optimization
function OptimizeANDNode(rgraph,on,sibs,srcs,q,EKB,RCB) return a list of sources
  inputs: rgraph, a rule-goal graph; on, a goal node
          sibs, on's sibling nodes;  srcs, an array of sets of sources
          q, a list of query triple patterns; EKB, the EquivalenceKB;
          RCB, the redundant cycle base
1:  Let $allsrcs = srcs[on]$, load(srcs[on], KB)
2:  do
3:      q = $q \wedge on, rs$ = askReasoner (KB, q)
4:      Let insts = extractJoinInsts(rs)
5:      computeSameAs(insts, EKB)
6:      for each $qtp \in sibs$ do
7:          srcs[qtp] = getSourceList(subgraph rooted at qtp, rs, q)
8:      Let on = $min_{t \in sibs\ that\ join\ with\ query}(srcs[t])$
9:      Remove on from sibs
10:     $allsrcs = allsrcs \cup srcs[on]$, load($srcs[on], KB$)
11:     if on is a child of rgraph.root AND rgraph is a cycle AND
        $sibs = \emptyset$ then
12:         load(srcs[on], KB)
13:         Let $rsc$ = askReasoner (KB, $rgraph$)
14:         Let insts = extractJoinInsts(rsc)
15:         computeSameAs(insts, EKB)
16:         rgraph.RS = computeRS(rgraph.CDVs, RCB)
17: while ($sibs \neq \emptyset$)
18: return allsrcs
```

```
Algorithm 3 Source selection for cyclic axioms
function getCyclicSourceList(rgraph, $rs$, q) returns a list of sources
  inputs: rgraph, a given rule-goal graph;  $rs$, a list of substitutions
          q, a list of query triple patterns
1: Let $rsInc = rs$, firstIt = true, allsrcs = ∅
2: while ($rsInc \neq \emptyset$ OR firstIt)
3:     $allsrcs = allsrcs \cup$ getSourceList($rgraph, rsInc, q$)
4:     $rsInc$ = rgraph.RS
5:     clear(rgraph.RS), firstIt = false
6: return allsrcs
```

Fig. 2. Pseudo code of handling cyclic axioms

information about instances, which will be elaborated in section 4.2. In the given rule-goal graph *rgraph*, each goal node has been adorned with its own binding information. In line 6 of Alg. 1, *qsources* is a source evaluation function. Given a QTP q and a term index I, $qsources(q, EKB) = \bigcap_{c \in terms(q,EKB)} I(c)$, which is essentially a set of data sources that are relevant to q [5]. The EKB is used here to collect q's relevant sources by using both q's constants and their equivalent constants in EKB. In line 9, when the current most selective QTP (on) is a cycle ending node, it means that a cycle is detected and we need to use it to update our RCB and then push it into the cycle stack (lines 10 and 11). Note, each goal node in the rule-goal graph can only be involved in one cycle as an ending node because two cycles sharing one ending node is equivalent to one cycle starting and ending at these two cycle's root nodes that has been annotated before. Then, we enter Alg. 3 to compute the cycle's fix point (line 12). In Alg. 3, we repeatedly collect sources by executing Alg. 1 if the current cycle's RS is not empty (lines 2-5). Here, the RS are computed at the end of each cycle iteration in lines 18-23 of Alg. 1 and lines 11-16 of Alg. 2 by extracting the new substitutions of the current cycle's CDVs and then passed to Alg. 3 for the recursion use. In this process, the function $extractJoinInsts(rsc)$ extracts join instances from the given subsitution list rsc (line 21 in Alg. 1, and lines

4 and 14 in Alg. 2). Its results are passed to our instance coreference handling algorithm (Alg. 4) to compute the *owl*:*sameAs* closure (lines 4 and 22 in Alg. 1, and lines 5 and 15 in Alg. 2). The function $computeRS(rgraph.CDVs, RCB)$ is to compute the RS of the given rule-goal graph *rgraph* (line 23 in Alg. 1 and line 16 in Alg. 2). Here, for each recursive constant, we check if its redundant cycles has used it before by using the RCB and rule out it from RS if it's been used. When the fix point is reached, we will return all collected sources (line 6) and go back to Alg. 1. Then, we continue to execute line 13 in Alg. 1 to pop the processed cyclic axiom.

3.2 Equality Reasoning

Our equality reasoning is based on the heuristic that within the term index, the QTPs with constant constraints are often highly selective. For instance, given two QTPs: *owl*:*sameAs*(*rpi*:*james*, ?*y*) and *owl*:*sameAs*(?*x*, ?*y*), the first is much more selective than the second because of the specific constant *rpi*:*james*. Therefore, compared to the way of loading all sources containing the *owl*:*sameAs* predicate to compute the instance coreference closure, this way helps us significantly reduce the number of sources that are involved in the closure computation. Given a query, we call the set of all instances that are used for the instance coreference closure computation as *Relevant Instances (RI).* Since we only compute the equivalence closure of the query constant instances and the join instances during the query solving, the cardinality of RI is often small. In our algorithm, we design an EquivalenceKB structure (EKB) that collects and organizes equivalence information about instances in RI. EKB essentially supports the disjoint set data structure operations on sets of equivalence classes of all known instances. An equivalence class in EKB is a set of instances that are equivalent to each other (explicitly or implicitly connected by *owl*:*sameAs*). Given an instance Ins in RI, we dynamically issue a boolean query "Ins" AND "*owl*:*sameAs*" to our index to find all relevant sources that contain Ins and its equivalent instances. Then, for each new discovered instance $newIns$, we further find $newIns$'s equivalent instances and merge the equivalence classes containing Ins and $newIns$. This process is repeated until no new instances are discovered. Since the cardinality of RI is often small as stated before, the computation will quickly and eventually terminate. Note, the equivalence class of each instance in RI is only computed once. The algorithm pseudo code is shown in Fig. 3.

First, we start with a set of seed instance URIs (Line 2), and use the term index to find all sources that contain each of these URIs concatenated with the "*owl*:*sameAs*" predicate (Lines 4 and 5). Note, the seed instances are not all coreferenced instances, but the instances in the RI of the given query and determined by Alg. 1. Then, we extract the new equivalent URIs (Line 7), merge the equivalence classes of the seed URIs and the new extracted URIs (Line 8), and collect the new URIs (Line 9). This process is iteratively repeated by using any new URIs discovered as seeds (Lines 10-11). Since there are a finite small number of seed instances as input, and the process will only continue as long as new URIs are discovered, the algorithm can quickly and eventually terminate.

```
Algorithm 4 Fix point computation for instance coreference
function computeSameAs(insts, EKB) returns a list of instances
inputs: insts, a list of seed URIs
Let $inslist = insts$, $oldinsts = insts$
for each uri ∈ insts do
    Let bquery = uri + "AND" + "owl:sameAs"
    Let srcslist = askIndexer(bquery)
    for each $s \in srcslist$ do
        Let sameAsPairs = $\{t \mid t = < x, owl{:}sameAs, y > \in s, x = uri \vee y = uri\}$
        updateEquivalenceKB(uri, sameAsPairs, EKB)
        $inslist = inslist \cup$ all instances URIs from sameAsPairs
Let newinsts = inslist – oldinsts
$inslist = inslist \cup$ ComputeSameAs($newinsts, EKB$)
return inslist
```

Fig. 3. Pseudo code of handling instance coreference

4 Evaluation

To evaluate our algorithms, we have conducted two groups of experiments based on a synthetic data set and a real world data set respectively. The first group measures the cycle handling performance of our algorithm. The second group tests the scalability and practicality of our algorithm using a subset of the real world Billion Triple Challenge (BTC) data set. For both groups, we use a graph-based synthetic query generator to produce a set of queries that are guaranteed to have at least one answer each. These queries range from one to eight triples, have at most seven variables each, and each QTP of each query satisfies the join condition with at least one sibling QTP. All of our experiments are done on a workstation with a Xeon 2.93G CPU and 6 GB memory running UNIX. Our indexer component uses a term index [5] implemented with Lucene. Our reasoner is KAON2.

4.1 Cyclic Axiom Evaluation Using a Synthetic Data Set

In this group of experiments, we conducted two separate experiments. The first aims to compare the tree-structure algorithm with the cycle handling algorithm without equality reasoning (tree-structure(cycle w/o sameAs)) to the tree-structure algorithm without the cycle handling (tree-structure(non-cycle)) and the non-structure algorithm using a synthetic data generator that is designed to approximate realistic conditions. The second aims to compare the tree-structure algorithm with the cycle handling including the equality reasoning (tree-structure(cycle)) to the tree-structure algorithm without the cycle handling (tree-structure(non-cycle)) and the non-structure algorithm. For both experiments, first, we ensure that each generated file is a connected graph, which is typical of most real-world RDF files. Based on a random sample of 200 semantic web documents, we set the average number of triples in a generated document to be 50. In order to achieve a very heterogeneous environment, we conducted experiments with 20 ontologies, 8000 data sources, and a diameter of 2, meaning that the longest sequence of mapping ontologies between any two domain ontologies was 2. In this configuration, the average number of sources committing to each ontology is 400. This configuration resulted in an index size of 75.3MB, which was built in 21.6 seconds.

Cyclic Axioms Without Equality Reasoning In this experiment, we issued 120 random queries to our synthetic data set to measure our cycle handling algorithm with the increasing cycle complexity, which is related to two factors: the average number of cycles per query and the average length per cycle. In addition, since the cycle complexity increases with the number of unconstrained QTPs, where an unconstrained QTP is one with variables for both its subject and object or with an *rdf:type* predicate paired with a variable subject, we group our 120 test queries by the number of unconstrained QTPs (from 0 to 5). The reason for selecting 5 as our maximum number of unconstrained QTPs is that the non-structure algorithm can only effectively scale to queries with at most 5 unconstrained QTPs [5]. In the metrics, we computed the average query response time and the cycle complexity. The experimental results are shown in Fig. 4.

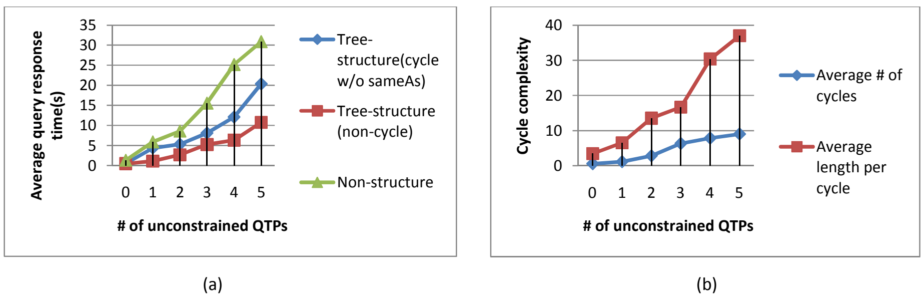


Fig. 4. Cyclic axiom handling algorithm w/o *owl:sameAs* experimental results. Average query response time (a) and cyclic axiom complexity (b) as the number of unconstrained QTPs varies.

Fig. 4 (a) shows how each algorithm's average query response time is affected by increasing the number of unconstrained QTPs with cycle complexity increasing. From this result, we can see that the tree-structure (cycle w/o sameAs) algorithm is faster than the non-structure algorithm. The reason is that unconstrained QTPs are typically the least selective; thus, the more unconstrained QTPs there are, the more opportunities there are for the tree-structure(cycle w/o sameAs) optimization algorithm to use constraints to enhance the selectivity of goals. Due to the additional cycle handling, the tree-structure (cycle w/o sameAs) algorithm is slower than the tree-structure algorithm (non-cycle), while the former can return more answers.

Fig. 4 (b) shows how the cyclic axiom complexity changes with the increasing number of unconstrained QTPs. As shown in this figure, our most complex test queries have 5 unconstrained QTPs, 10 cyclic axioms per query and 35.5 nodes per cyclic axiom. To the best of our knowledge, this complexity is significantly greater than most queries issued to the semantic web. Therefore, we can conclude that our cyclic axiom handling algorithm can effectively scale to the real world.

Full Cyclic Axiom Evaluation In this experiment, we introduce the *owl*:*sameAs* triples in our synthetic data set based on the *owl*:*sameAs* statistics of the Billion Triple Challenge 2010 data set [7]. The ratio of sources containing *owl*:*sameAs* is 27.1%. The number of *owl*:*sameAs* triples is 2,765 of 45,673 total triples in 8000 sources. All *owl*:*sameAs* triples are categorized into 571 equivalence classes. The largest equivalence class contains 10 instances and the average equivalence class is 3.7. Like the last experiment, we still issued 120 random queries to our synthetic data set and group them by the number of unconstrained QTPs (from 0 to 5). In the metrics, we computed the average query response time and the query completeness. The experimental results are shown in Fig. 5.

Fig. 5 (a) shows how each algorithm's average query response time is affected by increasing the number of unconstrained QTPs with the increasing number of unconstrained QTPs. From this result, we can see that the tree-structure (cycle) algorithm is faster than the non-structure algorithm. The reason is that unconstrained QTPs are typically the least selective; thus, the more unconstrained QTPs there are, the more opportunities there are for the tree-structure(cycle) optimization algorithm to use constraints to enhance the selectivity of goals. Due to the additional cycle and *owl*:*sameAs* handling, the tree-structure (cycle) algorithm is slower than the tree-structure algorithm (non-cycle).

Fig. 5 (b) shows the comparison of the completeness of the tree-structure (cycle) algorithm, the tree-structure (non-cycle) algorithm and the non-structure algorithm. Because the non-structure algorithm is complete [6], we take its results as ground truth. The percentage numbers in the graph are the completeness of the tree-structure (non-cycle) algorithm at each point. From this result, we can see that our tree-structure (cycle) algorithm is more complete than the tree-structure (non-cycle) algorithm. Furthermore, it returns the same number of answers as the non-structure algorithm, but has better query response time than the non-structure algorithm (as shown in Fig. 5 (a)).

4.2 Scalability Evaluation Using the BTC Data Set

In this section, we evaluate our algorithm's scalability by using a subset of the BTC 2009 data set (much of which comes from the Linking Open Data Project Cloud). We have chosen four collections, as summarized in Table 1, with a total of 73,889,151 triples including *owl*:*sameAs*. Using the provenance information in the BTC, we re-created local N3 versions of the original files from the BTC resulting in 21,008,285 documents. The size of documents varies from roughly 5 to 50 triples each. In order to integrate these heterogeneous documents, we manually created some mapping ontologies, primarily using *rdfs:subClassOf* and *rdfs:subPropertyOf* axioms (these schemas do not have any meaningful alignments that are more complex). In this experiment, our cyclic axioms are mainly

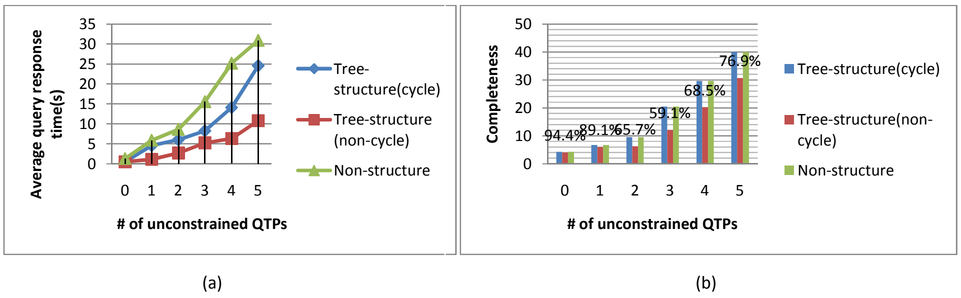


Fig. 5. Full cyclic axiom handling algorithm experimental results. Average query response time (a) and query completeness (b) as the number of unconstrained QTPs varies.

from mapping ontologies and *owl*:*sameAs* statements. The latter creates the most cycles. Our index construction time is around 58 hours and its size is around 18GB. Each document takes around 10ms on average to be indexed. The Lucene configurations are 1500MB for RAMBufferSize and 1000 for MergeFactor, which are the best tradeoff between index building and searching for our experiment.

Data Source	**Namespace**	**# of Sources**	**# of Triples**
http://data.semanticweb.org/	swrc	41,974	174,816
http://sws.geonames.org/	geonames	2,324,253	14,866,924
http://dbpedia.org/	dbpedia	10,615,260	48,694,372
http://dblp.rkb-explorer.com/	akt	8,026,878	10,153,039
Total		21,008,285	73,889,151

Table 1. Data sources selected from the BTC 2009 dataset.

Because the non-structure algorithm does not refine goals with constraint information from related goals, it cannot scale to the BTC data set. In fact, most of our synthetic queries cannot be solved by this algorithm. For example, consider the query Q:$\{\langle ?x_0, swrc{:}affiliation, \text{“}lehigh-univ\text{”}\rangle.\langle ?x_2, akt{:}has-title, \text{“}Hawkeye\text{”}\rangle.\langle ?x_2, foaf{:}maker, ?x_0\rangle.\langle ?x_0, akt{:}full-name, ?x_1\rangle\}$. For the non-structure algorithm, the number of sources that can potentially contribute to solving $\langle ?x_2, foaf{:}maker, ?x_0\rangle$ is 3,485,607, which is far too many to load into a memory-based reasoner. Even though some reasoners can load this amount of data as long as the system has 3GB of memory, load times are typically in the 7 hours range, which is clearly unsuitable for real-time queries. However, the tree-structure algorithms (cycle and non-cycle) can solve this problem because the number of sources for the same QTP becomes 114 after variable constraints are applied. For this reason, we only compare the tree-structure family algorithms.

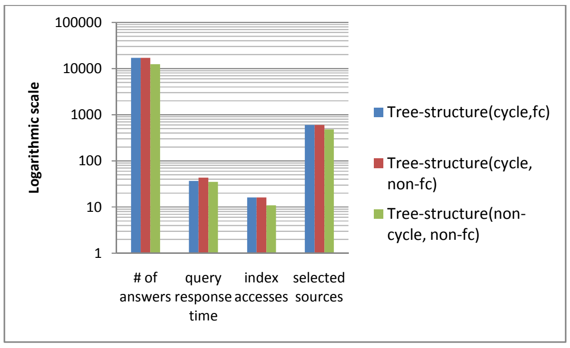


Fig. 6. BTC data set experimental results.

We executed 150 synthetic queries with at most 10 QTPs. In the metrics, we computed the average number of answers, average query response time, average number of selected sources and average index accesses of three algorithms: the tree-structure (cycle, fc), the tree-structure (cycle, non-fc) and the tree-structure(non-cycle, non-fc) algorithm. Here, "fc" stands for front-coding, which is an optimization technique we applied in order to improve the query response time of our algorithms. This is because of the fact that many URIs in the BTC data set have the same server name, and within each such set, there are many with the same namespace. The "fc" technique replaces each common server name with a number. As a result, our boolean query lengths are greatly compressed. The results are shown in Fig. 6 using a logarithmic scale. According to the results, we can see that the tree-structure (cycle, fc) and the tree-structure (cycle, non-fc) algorithms returned 36.8% more answers than the tree-structure (non-cycle, non-fc) algorithm even though they have small increases in the other three metrics because of the additional cycle processing. However, with the front-coding optimization, the query response time of the tree-structure (cycle, fc) algorithm has gained around 20% improvement over the tree-structure (cycle, non-fc) algorithm and is only around 5% more than the tree-structure (non-cycle, non-fc) algorithm.

5 Conclusions and Future Work

In this paper, we proposed a stack-based fix point computation algorithm to dynamically handle cyclic axioms including instance coreference for query answering over large distributed KBs. Using this algorithm, our system can deal with cyclic axioms on the fly and scale to queries with 8 QTPs (5 unconstrained QTPs), 10 cyclic axioms per query and 35.5 nodes per cyclic axiom on average. Meanwhile, it can return the same number of answers as the complete non-structure algorithm. In addition, we have also shown that our algorithm scales well on a real world data set, allowing randomly generated queries against 20 million heterogeneous data sources to complete in 30 seconds.

Despite showing initial promise, there is still significant room for improvement. First, the algorithm only focuses on conjunctive queries in SPARQL without FILTER and OPTIONALs. In addition, in order to avoid the computational challenges of higher-order logics, it does not allow variables in the predicate position. Second, the implementation only works with OWLII. In the future, we will explore how to extend our algorithms to support richer SPARQL queries and more expressive ontologies such as OWL 2, and also consider how to theoretically prove the correctness of our approach. We believe that this paper provides a major step towards a pragmatic solution for dynamic cyclic axiom handling in querying a large, distributed, and ever changing Semantic Web.

References

1. F. Bancilhon, D. Maier, Y. Sagiv, and J. D. Ullman. Magic sets and other strange ways to implement logic programs. In *PODS*, pages 1–15, 1986.
2. L. Ding, J. Shinavier, Z. Shangguan, and D. L. McGuinness. Sameas networks and beyond: Analyzing deployment status and implications of OWL: sameas in linked data. In *International Semantic Web Conference*, pages 145–160, 2010.
3. A. Y. Halevy, Z. G. Ives, J. Madhavan, P. Mork, D. Suciu, and I. Tatarinov. The Piazza peer data management system. *IEEE Trans. Knowl. Data Eng.*, 16(7):787–798, 2004.
4. J. S. C. Lam, D. H. Sleeman, J. Z. Pan, and W. W. Vasconcelos. A fine-grained approach to resolving unsatisfiable ontologies. *J. Data Semantics*, 10:62–95, 2008.
5. Y. Li and J. Heflin. Using reformulation trees to optimize queries over distributed heterogeneous sources. In *International Semantic Web Conference*, pages 502–517, 2010.
6. Y. Li, A. Qasem, and J. Heflin. A scalable indexing mechanism for ontology-based information integration. *Web Intelligence and Intelligent Agent Technology, IEEE/WIC/ACM International Conference on*, 2010.
7. Y. Li, Y. Yu, and J. Heflin. A multi-ontology synthetic benchmark for the semantic web. In *In Proc. of the 1st International Workshop on Evaluation of Semantic Technologies*, 2010.
8. J. Mei, L. Ma, and Y. Pan. Ontology query answering on databases. In *International Semantic Web Conference*, pages 445–458, 2006.
9. Z. Pan, Y. Li, and J. Heflin. A semantic web knowledge base system that supports large scale data integration. In *The Workshop on Scalable Semantic Web Knowledge Base Systems, ISWC*, 2009.
10. A. Qasem, D. A. Dimitrov, and J. Heflin. Efficient selection and integration of data sources for answering semantic web queries. *International Conference on Semantic Computing*, pages 245–252, 2008.
11. A. Qasem, D. A. Dimitrov, and J. Heflin. Towards scalable information integration with instance coreferences, 2009.
12. J. Urbani, S. Kotoulas, J. Maassen, N. Drost, F. Seinstra, F. V. Harmelen, and H. Bal. WebPIE: A web-scale parallel inference engine. In *In: Third IEEE International Scalable Computing Challenge (SCALE2010), held in conjunction with the 10th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGrid)*, 2010.
13. T. D. Wang, B. Parsia, and J. A. Hendler. A survey of the web ontology landscape. In *International Semantic Web Conference*, pages 682–694, 2006.

Probabilistic Abox Abduction in Description Logics

Murat Şensoy[1], Achille Fokoue[2], Mudhakar Srivatsa[2], and Jeff Z. Pan[1]

[1]Department of Computing Science, University of Aberdeen, UK
[2]IBM T. J. Watson Research Center, NY, USA

Abstract. *ABox* abduction is the process of finding statements that should be added to an ontology to entail a specific conclusion. In this paper, we propose an approach for probabilistic abductive reasoning for $\mathcal{SHIQ}$. Our evaluations show that the proposed approach significantly extends classical abduction by effectively and correctly estimating probabilities for abductive explanations. Lastly, based on the ideas proposed for $\mathcal{SHIQ}$, we describe a tractable algorithm for $\mathsf{DL\text{-}Lite}_{\mathcal{R}}$.

1 Introduction

The prevailing framework for querying and reasoning over data on the semantic web has been based on logical deduction: the ability to infer and retrieve implied facts logically entailed from a knowledge base. A number of highly optimized deductive reasoners (*e.g.,* Pellet [20], KAON2 [8], Hermit [14], TrOWL [22]) have been developed in compliance with the Web Ontology Language standard (OWL) and successfully used in various applications (*e.g.,* Matching Patients to Clinical Trials [16]).

However, there is a growing realization in the semantic web community [5] that deduction is insufficient for new classes of applications that could leverage the increasing number of formal ontologies available on the semantic web and in knowledge rich domains such as heathcare and life sciences. Applications that seek to explain observations (*e.g.,* a patient's symptoms or errors or failures in a complex systems) or expectations that are not logically entailed from our current knowledge would greatly benefit from an abductive reasoning paradigm. Abductive reasoning is the process of finding the explanations for a set of observations. In this context, an explanation is a set of axioms $\mathcal{S}$ that, if added to a knowledge base $\mathcal{K}$, will ensure that the combined knowledge base ($\mathcal{K} \cup \mathcal{S}$) now logically entails the set of observations.

In many situations, the number of abductive explanations for an axiom could be very high, making it very costly to process all of these explanations. Various criteria have been considered in the literature to define a preference order on abductive explanations and to select the *best* explanations. Preferred explanations are typically the smallest in terms of either their number of axioms or according to subset inclusion. However, the smallest explanation in terms of size or set inclusion is not necessarily the most likely. Formal attempts to define the best logical abductive solutions in terms of their likelihood have traditionally required an explicit specification of a probabilistic model as part of the background knowledge [9, 17].

In this paper, we formalize the notion of likelihood of solutions w.r.t. to a background knowledge without extending the Description Logic (DL) formalism with a probabilistic model. We propose a novel approach for probabilistic *Abox* abduction for $\mathcal{SHIQ}$, one of the most expressive *DLs*. In this approach, instead of extending DL formalism with a probabilistic model, we rely on the ability to discover patterns of explanations in a background knowledge base and compute simple statistics to find the most prevalent patterns in the knowledge base. Then, we estimate the likelihoods of abductive explanations based on these statistics. Through empirical studies, we compare the proposed approach with non-probabilistic abduction where the abductive explanations are assumed equally likely. We show that the proposed approach can effectively estimate the likelihoods of the abductive explanations.

Reasoning in $\mathcal{SHIQ}$ is known to be intractable [1]. Our empirical studies also highlight the fact that abductive reasoning in $\mathcal{SHIQ}$ is computationally expensive. Based on the ideas proposed for $\mathcal{SHIQ}$, we have described a tractable algorithm for probabilistic abduction in $\textsf{DL-Lite}_{\mathcal{R}}$ [2], the theoretical underpinning of OWL 2.0 QL profile. In particular, we show that the computational complexity class of this algorithm is the same as the computational complexity class of instance checking and conjunctive query answering in $\textsf{DL-Lite}_{\mathcal{R}}$.

The remainder of the paper is organized as follows. Section 2 introduces preliminaries necessary to follow the paper. Section 3 describes a non-probabilistic abduction approach for $\mathcal{SHIQ}$ and Section 4 builds our probabilistic abduction for $\mathcal{SHIQ}$ upon it. Section 5 evaluates the proposed approach through empirical studies and Section 6 proposes a tractable approach for probabilistic abduction in $\textsf{DL-Lite}_{\mathcal{R}}$. Lastly, Section 7 discusses the related work and Section 8 concludes the paper with an overview.

2 Preliminaries

2.1 $\mathcal{SHIQ}$ Description Logics

In this paper, unless stated otherwise, we consider ontologies of $\mathcal{SHIQ}$ expressiveness. In this section, we briefly introduce the semantics of $\mathcal{SHIQ}$, which is equivalent to OWL-DL 1.0 [1] minus nominals and datatype reasoning, as shown in Table 1 (We assume the reader is familiar with Description Logics [1]). Let $\mathcal{N}_C$ be the set of atomic concepts, $\mathcal{N}_R$ be the set of atomic roles, and $\mathcal{N}_I$ be the set of individuals. $\mathcal{N}_C$, $\mathcal{N}_R$, and $\mathcal{N}_I$ are mutually disjoint. Complex concepts and roles are built using constructs presented in Table 1(a).

A $\mathcal{SHIQ}$ knowledge base $\mathcal{K} = (\mathcal{T}, \mathcal{A})$ consists of a Tbox $\mathcal{T}$ and an Abox $\mathcal{A}$. A Tbox $\mathcal{T}$ is a finite set of axioms, including:

- transitivity axioms of the form Trans(R) where R is a role.

[1] http://www.w3.org/2001/sw/WebOnt

Definitions	Semantics
$C \sqcap D$	$C^{\mathcal{I}} \cap D^{\mathcal{I}}$
$C \sqcup D$	$C^{\mathcal{I}} \cup D^{\mathcal{I}}$
$\neg C$	$\Delta^{\mathcal{I}} \backslash C^{\mathcal{I}}$
$\exists R.C$	$\{x \vert \exists y. < x, y > \in R^{\mathcal{I}}, y \in C^{\mathcal{I}}\}$
$\forall R.C$	$\{x \vert \forall y. < x, y > \in R^{\mathcal{I}} \Rightarrow y \in C^{\mathcal{I}}\}$
$\leq nR$ C	$\{x \vert\ \vert\{< x, y > \in R^{\mathcal{I}} \wedge y \in C^{\mathcal{I}}\}\vert \leq n\}$
$\geq nR$ C	$\{x \vert\ \vert\{< x, y > \in R^{\mathcal{I}} \wedge y \in C^{\mathcal{I}}\}\vert \geq n\}$
R^-	$\{< x, y > \mid < y, x > \in R^{\mathcal{I}}\}$

(a) Constructors

Axioms	Satisfiability conditions
Trans(R)	$(R^{\mathcal{I}})^+ = R^{\mathcal{I}}$
$R \sqsubseteq P$	$< x, y > \in R^{\mathcal{I}} \Rightarrow < x, y > \in P^{\mathcal{I}}$
$C \sqsubseteq D$	$C^{\mathcal{I}} \subseteq D^{\mathcal{I}}$
$a : C$	$a^{\mathcal{I}} \in C^{\mathcal{I}}$
$R(a, b)$	$< a^{\mathcal{I}}, b^{\mathcal{I}} > \in R^{\mathcal{I}}$
$a \dot{\neq} b$	$a^{\mathcal{I}} \neq b^{\mathcal{I}}$

(b) Axioms

Table 1. SHIQ Description Logic

- role inclusion axioms of the form $R \sqsubseteq P$ where R and P are roles. $\sqsubseteq^*$ denotes the reflexive transitive closure of the $\sqsubseteq$ relation on roles.
- concept inclusion axioms of the form $C \sqsubseteq D$ where C and D are concept expressions.

An Abox $\mathcal{A}$ is a set of axioms of the form $a : C$, $R(a, b)$, and $a \dot{\neq} b$.

As for First Order Logic, a model theoretical semantic is adopted here. In the definition of the semantics of $\mathcal{SHIQ}$, $\mathcal{I}= (\Delta^{\mathcal{I}}, .^{\mathcal{I}})$ refers to an interpretation where $\Delta^{\mathcal{I}}$ is a non-empty set (the domain of the interpretation), and $.^{\mathcal{I}}$, the interpretation function, maps every atomic concept C to a set $C^{\mathcal{I}} \subseteq \Delta^{\mathcal{I}}$, every atomic role R to a binary relation $R^{\mathcal{I}} \subseteq \Delta^{\mathcal{I}} \times \Delta^{\mathcal{I}}$, and every individual a to $a^{\mathcal{I}} \in \Delta^{\mathcal{I}}$. The interpretation function is extended to complex concepts and roles as indicated in the second column of Table 1(a).

An interpretation $\mathcal{I}$ is a model of a knowledge base $\mathcal{K} = (\mathcal{T}, \mathcal{A})$, denoted $\mathcal{I} \models \mathcal{K}$, iff. it satisfies all the axioms in $\mathcal{A}$, and $\mathcal{T}$ (see Table 1(b)). A knowledge base $\mathcal{K} = (\mathcal{T}, \mathcal{A})$ is consistent iff. there is a model of $\mathcal{K}$. Let α be an axiom, a knowledge base $\mathcal{K}$ entails α, denoted $\mathcal{K} \models \alpha$, iff. every model of $\mathcal{K}$ satisfies α.

2.2 Conjunctive Query

Given a knowledge base $\mathcal{K}$ and a set of variables $\mathcal{N}_V$ disjoint from $\mathcal{N}_I$, $\mathcal{N}_R$, and $\mathcal{N}_C$, a conjunctive query q is of the form $x_1, ..., x_n \leftarrow t_1 \wedge ... \wedge t_m$ where, for $1 \leq i \leq n$, $x_i \in \mathcal{N}_V$ and, for $1 \leq j \leq m$, t_j is a query term. A query term t is of the form $C(x)$ or $R(x, y)$ where x and y are either variables in $\mathcal{N}_V$ or individuals in $\mathcal{N}_I$, C is an atomic concept and R is an atomic role. $body(q)$ denotes the set of query terms of q. $Var(q)$ refers to the set of variables occurring in query q, and $DVar(q) = \{x_1, ..., x_n\}$ is the subset of $Var(q)$ consisting of distinguished (or answer) variables. Non-distinguished variables (i.e., variables in $Var(q) - DVar(q)$) are existentially quantified variables.

Let π be a total function from the set $DVar(q)$ of distinguished variables to the set $\mathcal{N}_I$ of individuals. We say that π is an answer to q in the interpretation

$\mathcal{I}$, denoted $\mathcal{I} \models q[\pi]$, if there exists a total function ϕ from $Var(q) \cup \mathcal{N}_I$ to $\Delta^{\mathcal{I}}$ such that the following hold:

- if $x \in DVar(q)$, $\phi(x) = \pi(x)^{\mathcal{I}}$
- if $a \in \mathcal{N}_I$, $\phi(a) = a^{\mathcal{I}}$
- $\phi(x) \in C^{\mathcal{I}}$ for all query terms $C(x) \in body(q)$.
- $(\phi(x), \phi(y)) \in R^{\mathcal{I}}$ for all query terms $R(x, y) \in body(q)$.

$ans(q, \mathcal{I})$ denotes the set of all answers to q in $\mathcal{I}$. π is said to be a certain answer to q over a knowledge base $\mathcal{K}$ iff. $\pi \in ans(q, \mathcal{I})$ for every model $\mathcal{I}$ of $\mathcal{K}$. The set of all certain answers of q over $\mathcal{K}$ is denoted $cert(q, \mathcal{K})$.

2.3 Abductive Reasoning

Abduction is an important reasoning service which provides possible explanations (or hypotheses) for observations that are not entailed by our current knowledge. In this section, we briefly formalize the notion of Abox abduction.

Definition 1 *An abox abduction problem is a tuple $(\mathcal{K}, \mathcal{H}, A(a))$, where $\mathcal{K} = (\mathcal{T}, \mathcal{A})$ is a knowledge base, called the background knowledge base, $\mathcal{H}$ is a set of atomic concepts or roles, A is an atomic concept and a is an individual appearing in the Abox $\mathcal{A}$ of $\mathcal{K}$ such that $\mathcal{K}$ does not entail $A(a)$.*

In the previous definition, K corresponds to the background knowledge whose Tbox provides a conceptualization of the domain of discourse. $\mathcal{H}$ represents the concepts and roles that may appear in an explanation for the observation $A(a)$.

Definition 2 *A solution to an abox abduction problem $\mathcal{P} = (\mathcal{K} = (\mathcal{T}, \mathcal{A}), \mathcal{H}, A(a))$ is a set $\mathcal{S} = \{C(u) | C \in \mathcal{H}, u \in \mathcal{N}_I\} \cup \{R(u, v) | R \in \mathcal{H}, (u, v) \in \mathcal{N}_I^2\}$ of abox assertions such that:*

1. *The knowledge base $(\mathcal{T}, \mathcal{A} \cup \mathcal{S})$ is consistent*
2. *$(\mathcal{T}, \mathcal{A} \cup \mathcal{S}) \models A(a)$*

3 Non-Probabilistic Abduction for $\mathcal{SHIQ}$

Given an extensionally reduced $\mathcal{SHIQ}$ knowledge base $\mathcal{K} = (\mathcal{T}, \mathcal{A})$, the KAON2 transformation computes a disjunctive datalog program with equality, denoted by $DD(\mathcal{K})$. This datalog program is the union of a set of function-free rules compiled from $\mathcal{T}$ by exploiting certain resolution operations [8] and a set of ground rules directly translated from $\mathcal{A}$. Hustadt *et al.* showed that $\mathcal{K}$ is consistent if and only if $DD(\mathcal{K})$ is satisfiable [8]. The most state-of-the-art abduction systems [10, 13] are built on Prolog engines that work on plain datalog programs. Du *et al.* described a procedure to translate $DD(\mathcal{K})$ into a Prolog program $\bar{\mathcal{K}}$. That is, using a chain of transformation, we can convert $\mathcal{K}$ into a Prolog program $\bar{\mathcal{K}}$. Then, we can solve the abductive reasoning problem using $\bar{\mathcal{K}}$ and existing abductive reasoning methods for plain datalog programs [4]. In this section, we exploit this approach to find all abductive explanations for $A(a)$, then in Section 4 we propose an approach to estimate likelihoods for these explanations.

Figure 1 shows a simplified Prolog program for abductive reasoning over $\bar{\mathcal{K}}$. This simple program is composed of six rules. Using only six rules, this program defines the predicate $abduce(A, \mathcal{S})$, where A is an axiom such as $'Alcoholic'('Victoria')$ and $\mathcal{S}$ is a solution to the abduction problem (i.e., abductive explanation) computed by the program. For this purpose, it simply starts with an empty set of axioms as shown in the rule 1 and populates it iteratively based on other rules. The rule 2 guarantees that already entailed Abox axioms do not appear in $\mathcal{S}$. The rule 4 expands a complex abox axiom into its components by finding a clause in $\bar{\mathcal{K}}$ so that the head of the clause unifies the axiom. The rule 5 prevents redundancies in the solution. The rule 6 expands an existing partial solution by adding a new axiom if this axiom is an *abducible* and this addition does not create an inconsistency. Abducibles correspond to $\mathcal{H}$, i.e., the concepts or roles that we desire to appear in the solution. If a concept or role does not appear in the head of any clause in the prolog knowledge base, then it should also be an abducible. We may note that concept or role expressions in $\mathcal{K}$ may results in cycles in $\bar{\mathcal{K}}$. For instance, an expression such as $\exists hasParent.human \sqsubseteq human$ leads to a Prolog clause $human(X) :- hasParent(X, Y), human(Y)$. For the sake of simplicity, we have not shown it in Figure 1, but we implemented rule 4 so that it does not expand an axiom if this expansion results in a loop, instead this axiom is added directly to the solution. In this way, we prevent infinite loops during abductive reasoning.

```
1. abduce(A,S):-
       abduce(A,[],S).
2. abduce(A,S,S):-
       holds(A),!.
3. abduce((A,B),S0,S):-!,
       abduce(A,S0,S1),
       abduce(B,S1,S).
4. abduce(A,S0,S):-!,
       clause(A,B),
       abduce(B,S0,S).
5. abduce(A,S,S) :-
       member(A,S),!.
6. abduce(A,S,[A|S]):-
       abducible(A),
       checkConsistency([A|S]).
```

Fig. 1. Simplified abductive reasoner for plain datalog programs.

4 Probabilistic Abduction for $\mathcal{SHIQ}$

Various minimality criteria have been considered in the literature to define a preference order on solutions to an abduction problem. Preferred solutions are typically the smallest in terms of either their number of axioms or according to subset inclusion. However, the smallest explanation in terms of size or set inclusion is not necessarily the most likely. Formal attempts to define the best logical abductive solutions in terms of their likelihood have traditionally required an explicit specification of a probabilistic model as part of the background knowledge [9, 17]. In this section, we formalize the notion of likelihood of solutions w.r.t. to the background knowledge without extending the DL formalism with a probabilistic model. First, we introduce, as a running example, the following knowledge base $K = (\mathcal{T}, \mathcal{A})$:

Example 1. $\mathcal{T} = \{\exists addictedTo.AlcoholicBeverage \sqsubseteq Alcoholic,$
$Wine \sqsubseteq AlcoholicBevarage, Whisky \sqsubseteq AlcoholicBevarage,$
$Vodka \sqsubseteq AlcoholicBevarage, RedWine \sqsubseteq Wine, WhiteWine \sqsubseteq Wine$
$Man \sqsubseteq Person, Woman \sqsubseteq Person, Man \sqsubseteq \neg Woman\}$
$\mathcal{A} = \{addictedTo(Mary, red_1), addictedTo(Helen, red_2), addictedTo(Jane, white_1)$
$addictedTo(Elisabeth, vodka_1), addictedTo(Elisabeth, whisky_1),$
$addictedTo(John, vodka_2), addictedTo(Paul, vodka_1), addictedTo(Henry, vodka_2),$
$addictedTo(Bob, vodka_3), addictedTo(James, whisky_1),$
$WhiteWine(white_1), Whisky(whisky_1), Woman(Victoria)\}$
$\bigcup\{RedWine(red_n) | 1 \leq n \leq 2\} \bigcup\{Vodka(vodka_n) | 1 \leq n \leq 3\}$

The following are valid solutions to the abduction problem $\mathcal{P} = (K, \mathcal{H} = \{addictedTo, Vodka, Wine, RedWine, WhiteWine, Whisky\}, Alcoholic(Victoria))$:

$$\mathcal{S}_1 = \{addictedTo(Victoria, vodka_1)\}$$
$$\mathcal{S}_2 = \{addictedTo(Victoria, newWine), Wine(newWine)\}$$

Which of the two explanations is more likely given the background knowledge base? In $(\mathcal{T}, \mathcal{A} \cup \mathcal{S}_1)$, the only abox justification[2] for $Alcoholic(Victoria)$, i.e. a minimum set of Abox assertions $\mathcal{J}$ such that $(\mathcal{T}, \mathcal{J}) \models Alcoholic(Victoria)$, is $\mathcal{J}_1 = \{addictedTo(Victoria, vodka_1), Vodka(vodka_1)\}$, whereas $\mathcal{J}_2 = \{addictedTo(Victoria, newWine), Wine(newWine)\}$ is the only justification for $Alcoholic(Victoria)$ in $(\mathcal{T}, \mathcal{A} \cup \mathcal{S}_2)$. Now, the justification $\mathcal{J}_1$ can be abstracted into a pattern of justifications $\widehat{\mathcal{J}_1} = x \leftarrow addicted(x, y) \wedge Vodka(y)$ representing all justifications of $Alcoholic(x)$ involving an addiction to a $Vodka$. In the background knowledge base K, 5 out of 10 justifications for $Alcoholic(x)$, with x an individual in $\mathcal{A}$, are instances of the pattern $\widehat{\mathcal{J}_1}$. Intuitively, a justification $\mathcal{J}$ for $Alcoholic(a)$, where a is an individual in $\mathcal{A}$, is an instance of the pattern $\widehat{\mathcal{J}_1} = x \leftarrow addictedTo(x, y) \wedge Vodka(y)$ iff. the conjunctive query $x \leftarrow addictedTo(x, y) \wedge Vodka(y)$ issued over $(\mathcal{T}, \mathcal{J})$ has a as an answer. On the other hand, only 3 out of 10 justifications for $Alcoholic(x)$ in the background KB K are instances of the justification pattern $\widehat{\mathcal{J}_2} = x \leftarrow addictedTo(x, y) \wedge Wine(y)$ associated with $\mathcal{J}_2$. The likelihood of the solution $\mathcal{S}_1$ (resp. $\mathcal{S}_2$)), denoted $Pr(\mathcal{S}_1)$ (resp. $Pr(\mathcal{S}_2)$), is 0.5 (resp. 0.3). Thus, $\mathcal{S}_1$ appears as the most likely solution.

Next, we formally define the notions of a justification pattern and an instance of a justification pattern.

Definition 3 *Let $\mathcal{B}$ be a subset of the Abox $\mathcal{A}$ of a knowledge base $\mathcal{K}$ and a be an individual in $\mathcal{B}$. The abox pattern with focus a, denoted $\widehat{\mathcal{B}(a)}$, associated with $\mathcal{B}$*

[2] An abox justification for an axiom is a minimal set of a-box assertions entailing it.

is the conjunctive query $x \leftarrow t_1 \wedge ... \wedge t_n$ *such that* $\{t_1, ..., t_n\} = \{A(\pi(b)) | A(b) \in \mathcal{B}\} \cup \{R(\pi(b), \pi(c)) | R(b, c) \in \mathcal{B}\}$, *where* π *is an injective mapping from individuals in* $\mathcal{B}$ *to new variables such that* $\pi(a) = x$.

Definition 4 *Given a knowledge base* $\mathcal{K} = (\mathcal{T}, \mathcal{A})$, *a subset* $\mathcal{S}$ *of* $\mathcal{A}$ *containing the individual* b *is an instance with focus* b *of an abox pattern* $q = x \leftarrow t_1 \wedge ... \wedge t_n$, *denoted* $\mathcal{S} \models^b q$, *iff.* $(x \rightarrow b) \in cert(q, (\mathcal{T}, \mathcal{S}))$.

Notation 1 $\Omega(K, A(a))$ *denotes the set of all abox justifications for* $A(a)$ *in* $K = (\mathcal{T}, \mathcal{A})$, *where* A *is an atomic concept and* a *is an individual in* $\mathcal{A}$. *Then,* $|\Omega(\mathcal{K}, A|B)|$ *is the number of all justifications in the background knowledge base* $\mathcal{K}$ *for answers of the conjunctive query* $x \leftarrow A(x)$ *that are also instances of the concept* B. *Given* $Ind(\mathcal{A})$ *denotes the set of individuals in* $\mathcal{A}$, $|\Omega(\mathcal{K}, A|B)|$ *is computed as follows.*

$$|\Omega(\mathcal{K}, A|B)| = \sum_{a \in Ind(\mathcal{A}) \ s.t. \ \mathcal{K} \models B(a)} |\Omega(\mathcal{K}, A(a))|$$

Let $\mathcal{S}$ be a solution to an abduction problem $(\mathcal{K} = (\mathcal{T}, \mathcal{A}), \mathcal{H}, A(a))$, computed on $\bar{\mathcal{K}}$ as described in Section 3. Assuming that $\mathcal{J}$ is the only justification for $A(a)$ in $\mathcal{A} \cup \mathcal{S}$, the likelihood of the solution $\mathcal{S}$ is intuitively the fraction of justifications for the answers to the query $x \leftarrow A(x)$ in the background KB $\mathcal{K}$ that conform to (ie. are instances of) the abox pattern associated with $\mathcal{J} \cup \mathcal{S}$. Now, in Definition 5, we define how unconditional likelihoods for the solutions could be computed, where the number of justifications in the background KB, $|\Omega(\mathcal{K}, A)|\top|$, is a measure of our confidence in the computed likelihood. Here, the unconditional likelihood of an explanation $\mathcal{S}$ is formalised as the likelihood of the most probable justification of $A(a)$ in $\mathcal{K} \cup S$. Let $\mathcal{J}$ be a justification of $A(a)$, then the probability of $\mathcal{J}$, denoted as $Pr_{just}(\mathcal{J})$, is computed as frequency of the justification pattern derived from $\mathcal{J}$ in all abox justifications for the current instances of A, i.e., $\Omega(\mathcal{K}, A|\top)$. Consider Example 1, the likelihood of $\mathcal{S}_1$ is higher than that of $\mathcal{S}_2$, since addiction to vodka is more frequent than addiction to wine among all known alcoholics.

Definition 5 *The unconditional likelihood of a solution* $\mathcal{S}$ *of an abduction problem* $(\mathcal{K} = (\mathcal{T}, \mathcal{A}), \mathcal{H}, A(a))$, *denoted* $Pr(\mathcal{S})$, *is the real number between 0 and 1 defined as follows:*

- *if* $cert(x \leftarrow A(x), \mathcal{K}) = \emptyset$, $Pr(\mathcal{S}) = 0$
- *if* $cert(x \leftarrow A(x), \mathcal{K}) \neq \emptyset$,
  $Pr(\mathcal{S}) = \max_{\mathcal{J} \in \Omega((\mathcal{T}, \mathcal{A} \cup \mathcal{S}), A(a))} Pr_{just}(\mathcal{J})$
  $Pr_{just}(\mathcal{J}) = \frac{\sum_{b \in ind(\mathcal{A})} |\{\mathcal{J}' | \mathcal{J}' \in \Omega(\mathcal{K}, A(b)) \ and \ \mathcal{J}' \models^b \widehat{(\mathcal{J} \cup \mathcal{S})}(a)\}|}{|\Omega(\mathcal{K}, A|\top)|}$

Following Definition 5, let us note that if S has an axiom not entailed by any justification for $A(b)$, with b an individual in $\mathcal{A}$, $Pr(S) = 0$. This follows from the fact that no justification $\mathcal{J}'$ in $\mathcal{K}$ will be an instance of the pattern,

$\widehat{(\mathcal{J} \cup \mathcal{S})}(a)$, associated with $\mathcal{J} \cup \mathcal{S}$, where $\mathcal{J}$ is a justification in $(\mathcal{T}, \mathcal{A} \cup \mathcal{S})$ for $A(a)$. For example, $Pr(S_1 \cup \{Person(Victoria)\}) = 0$, where S_1 is the first solution introduced in the running example.

While computing likelihoods for solutions it is key to compute $\Omega(\mathcal{K}, A(a))$. Now we briefly describe how to compute it. Given an arbitrary primitive concept C and individual i in $\mathcal{K}$, $\mathcal{K} \models C(i)$ iff. $\bar{\mathcal{K}} \models C(i)$, so in order to efficiently compute abox justifications for $C(i)$, we use a Prolog meta-interpreter [21] that tracks the steps while proving $C(i)$ in $\bar{\mathcal{K}}$. Hence, using the meta interpreter, we enumerate all proofs for $A(i)$ in $\bar{\mathcal{K}}$, each of which is one justification. Based on these justifications, we compute $\Omega(\mathcal{K}, A(i))$.

Now, suppose the set of axioms $\{Woman(Mary), Woman(Helen), Woman(Jane)$, $Woman(Elisabeth)$, $Man(John)$, $Man(Paul)$, $Man(Henry)$, $Man(Bob)$, $Man(James)\}$ were added to the Abox $\mathcal{A}$ of Example 1. It could be argued that the most likely solution for $\mathcal{P}$, is $\mathcal{S}_2 = \{$ $addictedTo(Victoria, newWine)$, $Wine(newWine)\}$ instead of $\mathcal{S}_1 = \{addictedTo(Victoria, vodka_1\}$ because, when we consider only female alcoholics, 3 out of 4 are addicted to Wine and Victoria is known to be a woman. Assuming that we have enough instances of the concept $Woman$ in the background KB, a more appropriate measure is the likelihood of a solution to $\mathcal{P}$ knowing that $Victoria$ is a $Woman$. The following definition formalizes the notion of conditional likelihood.

Definition 6 *The conditional likelihood of a solution $\mathcal{S}$ of an abduction problem $(\mathcal{K} = (\mathcal{T}, \mathcal{A}), \mathcal{H}, A(a))$ knowing a is an instance of a concept C, denoted $Pr(\mathcal{S}|C)$, is the real number between 0 and 1 defined as follows:*

- *if $cert(x \leftarrow A(x) \wedge C(x), \mathcal{K}) = \emptyset$, $Pr(\mathcal{S}) = 0$*
- *if $cert(x \leftarrow A(x) \wedge C(x), \mathcal{K}) \neq \emptyset$,*
  $Pr(\mathcal{S}|C) = \max_{\mathcal{J} \in \Omega((\mathcal{T}, \mathcal{A} \cup \mathcal{S}), A(a))} Pr_{just}(\mathcal{J}|C)$
  $Pr_{just}(\mathcal{J}|C) = \frac{\sum_{b \in ind(\mathcal{A}) \text{ and } \mathcal{K} \models C(b)} |\{\mathcal{J}' | \mathcal{J}' \in \Omega(\mathcal{K}, A(b)) \text{ and } \mathcal{J}' \models^b \widehat{(\mathcal{J} \cup \mathcal{S})}(a)\}|}{|\Omega(\mathcal{K}, A|C)|}$

Definition 6 allows us to estimate the likelihood of abductive explanations (i.e., solutions for the abduction problem) for $A(a)$ within the context that we know a is an instance of the concept C. If C is too specific, then it significantly reduce our confidence on $Pr(\mathcal{S}|C)$ by reducing $|\Omega(\mathcal{K}, A|C)|$. For instance, if C is a concept with only three instances $\{a, b, c\}$ in $\mathcal{K}$, then $|\Omega(\mathcal{K}, A|C)|$ can be much smaller than desired. To select the best context, here we propose starting from the most specific context and iteratively generalize it until our criteria for an acceptable context holds. This idea is formalized in the algorithm below. The algorithm accepts a knowledge base $\mathcal{K} = (\mathcal{T}, \mathcal{A})$ and the threshold N as input to find the most specific concept description C acceptable for serving as a context. Here, N determines the minimum number of individuals C should have to stop further generalization. In the algorithm, we keep a set S representing the set of C's direct super concepts. Initially, C is set to the most specific context C', which is the intersection of a's direct types (line 1), and S contains only C' (line 2). While C has a number of individuals less than N and S contains some elements, we get the most specific element $s \in S$ (line 4). The most specific element is a concept or a concept description having the longest path to the top concept $\top$ when put into the concept hierarchy derived from $\mathcal{K}$. Then, we update S by removing s and adding super concepts of s (line 5). Some super concepts of s may be equivalent to s, so we remove these while updating S. Lastly, at the end of each iteration, we set C to a DL concept description, the intersection of elements in S (line 6). If S contains only one element, C is set to this element. By selecting the most specific element at each iteration during generalization, we aim fine grained generalization. However, if the total number of individual in the ontology is less than N, S becomes empty after the removal of the top concept $\top$ and the generalization stops; the algorithm returns $\top$ in this situation.

```
findContext($\mathcal{K} = (\mathcal{T}, \mathcal{A}), a, N)$)
Input: $\mathcal{K} = (\mathcal{T}, \mathcal{A})$ a knowledge base, $a$ an individual, $N$ a threshold
Output: $C$ a concept description representing context
(1)     $C = C' = intersectionOf(getDirectTypes(a))$
(2)     $S = \{C'\}$
(3)     while $|getIndividuals(C, \mathcal{K})| < N$ and $|S| > 0$
(4)         $s = getMostSpecific(S, \mathcal{T})$
(5)         $S = S \setminus \{s\} \cup supers(s, \mathcal{T}) \setminus equivalents(s, \mathcal{T})$
(6)         $C = intersectionOf(S)$
(7)     return $C$
```

The algorithm allows us to generalize context gradually until reaching a concept description C that has desirable number of instances. However, at each iteration, the distance between the new C and the most specific context C' may increase further.This brings the risk of having a context with enough number of instances, but failing represent a as expected. Once we define a distance metric between two concept descriptions C' and C, we can introduce another threshold δ for distance and avoid over generalization by testing $distance(C, C') < \delta$ during iterations, at line 3 of the algorithm. We can use various distance metrics, two of which can be summarized as: **i)** the number of iterations done to derive C from C' and **ii)** the distance between the concept descriptions C' and C in the concept hierarchy derived from $\mathcal{K}$, after inserting them as new concepts into $\mathcal{K}$ if they do not exist there already.

In Definition 6, the likelihood of a solution is defined in terms of justifications in the background knowledge base. Unfortunately, computing all justifications is well known to be intractable [11]. In Section 6, we show that abductive explanations and their likelihoods can be computed efficiently (PTime in the size of the TBox and LogSpace in the size of the Abox) for DL-Lite$_\mathcal{R}$ KBs.

5 Evaluation of Abduction in $\mathcal{SHIQ}$

In order to evaluate our approach, we have randomly created five $\mathcal{SHIQ}$ ontologies.Properties of these ontologies are listed in Table 2. Each of these ontologies contains 10 *target concepts*. We measure the performance of the proposed approach through these target concepts. Each target concept C has at least n possible patterns of justification.

For each individual I in ontology O, we select a target concept C. Then, among all justification patterns of C, we randomly select one pattern $\widehat{\mathcal{J}}$. Based on the selected justification pattern, we add new ABox axioms to O so that I would be an instance of C. For instance, if $\widehat{\mathcal{J}}$ is a pattern of three atoms such as $C(X) \leftarrow A(X), B(X, Y), D(Y)$, three ABox axioms $A(I)$, $B(I, i)$, and $D(i)$ are added to O, where i is another individual from O. We also extend the ABox by randomly adding other axioms about I, as long as these axioms do not lead to a second justification for $C(I)$. Let us note that justifications for instances of C in O are not uniformly distributed, because while selecting justification patterns of C for individuals, we use power low distribution instead of uniform distribution. This means that most of the justifications for C are instances of small number of justification patterns of C, while most justification patterns of C have a few or no instances in O.

We evaluate an abductive reasoning approach based on a target concept C as follows. First, we pick I, an instance of C. Let $\mathcal{J}$ be the justification for $C(I)$. Second, we randomly select a subset of the axioms in $\mathcal{J}$, denoted as Γ. Third, we remove the axioms in Γ from O. Hence, $C(I)$ does not hold any more. Fourth, using the abductive reasoning approach, we compute all abductive explanations of $C(I)$ for O with their probabilities. Let E_Γ be the abductive explanation unifying with Γ. Performance of the abduction approach is $Pr(E_\Gamma)$, which is the estimated probability of E_Γ by the abduction approach. Here, we compared three abductive reasoning approaches: non-probabilistic abduction (NPA), unconditional probabilistic abduction (UPA), and conditional probabilistic abduction (CPA). In NPA, after computing all abductive explanations, each explanation is considered equally likely, so if there are n explanations in total, each of them will have probability $1/n$. In CPA, we have used threshold $N = 20$ while generalizing context during abduction.

Table 2. Synthetic ontologies with different numbers of atomic concepts ($\#C$), roles ($\#R$), individual ($\#I$), $TBox$ axioms ($\#TA$), and $ABox$ axioms ($\#AA$).$\langle n \rangle$ denotes average number of justification patterns for main concepts.

Ontology	#C	#R	#I	#TA	#AA	$\langle n \rangle$
O_1	187	20	1000	339	4000	18
O_2	285	40	3732	602	12000	33
O_3	393	50	5199	719	13646	54
O_4	381	60	4795	722	17985	63
O_5	353	70	5283	691	23418	234

We have conducted experiments for 100 individuals in each ontology. Average values for our experiments are listed in Table 3, where $\langle \#E \rangle$ is the average number of explanations for $C(I)$; $\langle P_{NPA} \rangle$, $\langle P_{UPA} \rangle$, and $\langle P_{CPA} \rangle$ are the average performances of NPA, UPA, and CPA respectively; $\langle T_{NPA} \rangle$, $\langle T_{UPA} \rangle$, and $\langle T_{CPA} \rangle$ are average time spent in milliseconds by NPA, UPA, and CPA respectively. We can summarize our findings as follows. As the number of explanations increase, the performance of classical abduction decreases as expected. while the performance of UPA always significantly outperform NPA, it could not exceed 0.23 in the experiments. However, the performance of CPA is always around 0.8. These are the results for synthetic ontologies, where we have created abox axioms around a number of justification patterns. To test our approach using an ontology which is not created with justification patterns in mind, we have also conducted experiments using $Wine^-$ ontology, which is the W3C's Wine ontology [19] without nominals. Our results for $Wine^-$ endorse our findings based on the synthetic ontologies. That is, UPA and CPA significantly outperform NPA, i.e. at the magnitudes of 7 and 11 respectively. In general, during the computation of probabilities, CPA requires significantly more time than UPA does. Our analysis of time consumption highlights that the probabilistic abduction in $\mathcal{SHIQ}$ is expensive computationally as expected. In the following section, we propose a tractable algorithm for probabilistic abductive reasoning in $\textsf{DL-Lite}_{\mathcal{R}}$.

Table 3. Results for synthetic ontologies and $Wine^-$ ontology ($N = 20$ for CPA).

Ontology	$\langle \#E \rangle$	$\langle P_{NPA} \rangle$	$\langle P_{UPA} \rangle$	$\langle P_{CPA} \rangle$	$\langle T_{NPA} \rangle$	$\langle T_{UPA} \rangle$	$\langle T_{CPA} \rangle$
O_1	15	0.073	0.23	0.84	73 ms.	850 ms.	1293 ms.
O_2	31	0.033	0.099	0.82	204 ms.	5612 ms.	7266 ms.
O_3	45	0.023	0.177	0.80	449 ms.	9643 ms.	12274 ms.
O_4	52	0.020	0.055	0.83	678 ms.	34183 ms.	35264 ms.
O_5	135	0.011	0.056	0.82	2658 ms.	70356 ms.	73938 ms.
$Wine^-$	70	0.014	0.099	0.16	37251 ms.	37288 ms.	46916 ms.

6 $\textsf{DL-Lite}_{\mathcal{R}}$ Probabilistic Abduction Algorithm

In this section, we present a tractable algorithm to compute solutions to an abduction problem $\mathcal{P} = (\mathcal{K}, \mathcal{H}, A(a))$ along with their likelihood. The expressivity of background knowledge base $\mathcal{K}$ is restricted to $\textsf{DL-Lite}_{\mathcal{R}}$ [2], the theoretical underpinning of OWL 2.0 QL profile. In particular, we show that the computational complexity class of this algorithm is the same as the computational complexity class of instance checking and conjunctive query answering in $\textsf{DL-Lite}_{\mathcal{R}}$.

First, we remind the restrictions imposed by $\textsf{DL-Lite}_{\mathcal{R}}$. Concepts and roles are formed according to the following syntax (A denotes an atomic concept and P an atomic role):

$$B \rightarrow A \mid \exists R \qquad\qquad R \rightarrow P \mid P^- \tag{1}$$

$$C \rightarrow B \mid \neg B \qquad\qquad E \rightarrow R \mid \neg R \tag{2}$$

Furthermore, Tbox axioms are restricted to the following forms: $B \sqsubseteq C$ and $R \sqsubseteq E$.

A key property of $\textsf{DL-Lite}_{\mathcal{R}}$ is that conjunctive query answering in a KB $\mathcal{K} = (\mathcal{T}, \mathcal{A})$ can be reduced to union of conjunctive query answering against the KB $\mathcal{K}' = (\emptyset, \mathcal{A})$ with an empty Tbox. In other words, through query rewrite, the relevant part of $\mathcal{T}$ can be compiled into a new query. In [2], for a conjunctive query q and a Tbox $\mathcal{T}$, the result of the rewritting, denoted $PerfectRef(q, \mathcal{T})$, is a set of conjunctive queries such that, for any Abox $\mathcal{A}$, $cert(q, (\mathcal{T}, \mathcal{A})) = \bigcup_{q' \in PerfectRef(q,\mathcal{T})} cert(q', (\emptyset, \mathcal{A}))$.

For example, after removing the axiom $\exists addictedTo.AlcoholicBeverage \sqsubseteq Alcoholic$ from the Tbox of our running example, it becomes a DL-Lite$_{\mathcal{R}}$ KB. For the query $q = x \leftarrow AlcoholicBeverage(x)$,

$$PerfectRef(q, \mathcal{T}) = \{x \leftarrow AlcoholicBeverage(x), x \leftarrow Wine(x), x \leftarrow RedWine(x),$$

$$x \leftarrow WhiteWine(x), x \leftarrow Vodka(x), x \leftarrow Whisky(x)\}$$

Our approach to compute solutions to an abduction problem and their likelihood relies on the observation that an abox justification for $A(a)$ in a DL-Lite$_{\mathcal{R}}$ KB $\mathcal{K} = (\mathcal{T}, \mathcal{A})$ must be an instance of the pattern formed by the body of one query in $PerfectRef(x \leftarrow A(x), \mathcal{T})$.

Proposition 1 *Let $K = (\mathcal{T}, \mathcal{A})$ be a DL-Lite$_{\mathcal{R}}$ knowledge base and A be an atomic concept such that $(\mathcal{T}, \emptyset)$ does not entail $\top \sqsubseteq A$. $\mathcal{J}$ is an abox justification for $A(a)$ in K iff. there exist $q \in PerfectRef(x \leftarrow A(x), \mathcal{T})$ and a mapping π from $Var(q) \cup Ind(\mathcal{J})$ to the set $Ind(\mathcal{J})$ of individuals in $\mathcal{J}$ such that (1) $\pi(x) = a$, (2) for $b \in Ind(\mathcal{J}), \pi(b) = b$, and*

$$(3)\ \mathcal{J} = \{C(\pi(u)) | C(u) \in body(q)\} \cup \{R(\pi(u), \pi(v)) | R(u, v) \in body(q)\}$$

Notation 2 *In the remainder of the paper, for a query q and a mapping π from $Var(q) \cup \mathcal{N}_I$ to $\mathcal{N}_I$, $construct(q, \pi)$ is the abox defined as follows:*

$$construct(q, \pi) = \{C(\pi(u)) | C(u) \in body(q)\} \cup \{R(\pi(u), \pi(v)) | R(u, v) \in body(q)\}$$

Proof.
- Suppose $\mathcal{J}$ is a abox justification for $A(a)$ in K. Since a is an answer to $x \leftarrow A(x)$, there exist $q \in PerfectRef(x \leftarrow A(x), \mathcal{T})$ and a mapping π from variables in q to individuals in $\mathcal{J}$ such that $\pi(x) = a$ and $construct(q, \pi) \subseteq \mathcal{J}$. π is extended to individuals b in $\mathcal{J}$ ($\pi(b) = b$). Since $construct(q, \pi)$ entails $A(a)$ and $\mathcal{J}$ is an abox justification for $A(a)$, it follows that $construct(q, \pi) = \mathcal{J}$.
- Let us assume that there exist $q \in PerfectRef(x \leftarrow A(x), \mathcal{T})$ and a mapping π from $Var(q) \cup Ind(\mathcal{J})$ to the set $Ind(\mathcal{J})$ of individuals in $\mathcal{J}$ satisfying the three conditions of Proposition 1. The rewritting performed by $PerfectRef(p, \mathcal{T})$ is such that every generated query $p' \in PerfectRef(q, \mathcal{T})$ has at most the same number of atoms as p and has at least one atom. Therefore $|construct(q, \pi)| = 1$, which makes $construct(q, \pi)$ mininal since $(\mathcal{T}, \emptyset)$ does not entail $\top \sqsubseteq A$. Thus, $construct(q, \pi)$ is an abox justification for $A(a)$.

Algorithm COMPUTESOLUTIONS computes a set of canonical solutions. Those solutions are canonical in the sense that, as shown in Theorem 1, an explanation for a solution not returned by COMPUTESOLUTIONS is always an instance of the pattern formed by a solution returned by COMPUTESOLUTIONS. Algorithm computeSolutions invokes Algorithm computeOmega to compute in $|\Omega(\mathcal{K}, A|C)|$. Before formally presenting properties of Algorithm computeSolutions, we briefly introduce below an important notation used in computeSolutions.

Notation 3 *For a conjunctive query q, the query $\overline{q}$ is the conjunctive query with the same body as q, but whose set of distinguished variables consists of all variables in q (i.e., $DVar(\overline{q}) = Var(q)$): $\overline{q} = x_1, ..., x_k \leftarrow t_1 \wedge ... \wedge t_m$ where $t_j \in body(q)$ for $1 \leq j \leq m$, and $x_1, ..., x_k$ are all variables in $body(q)$. Example, if $q = x \leftarrow A(x) \wedge R(x, y) \wedge S(y, z)$, then $\overline{q} = x, y, z \leftarrow A(x) \wedge R(x, y) \wedge S(y, z)$.*

computeSolutions($\mathcal{P} = (\mathcal{K} = (\mathcal{T}, \mathcal{A}), \mathcal{H}, A(a)), C$)
Input: $\mathcal{P} = (\mathcal{K}, \mathcal{H}, A(a))$ a abduction problem, C is a concept description or $\top$ s.t. $\mathcal{K} \models C(a)$
Output: set of pairs $(\mathcal{S}, p)$, where $\mathcal{S}$ is a solution to $\mathcal{P}$ with an conditional likelihood p knowing a is an instance of C

```
(1)   $\Omega \leftarrow$ computeOmega($\mathcal{K}, A, C$)
(2)   foreach $q_i$ in $PerfectRef(x \leftarrow A(x), \mathcal{T})$
(3)       $\pi$ a mapping from $Var(q_i) \cup \mathcal{N}_I$ to $\mathcal{N}_I$ s.t. (1) $\pi(x) = a$, (2) for $b \in$
          $\mathcal{N}_I, \pi(b) = b$, and, (3) for $y \in Var(q_i)$ such that $y \neq x$, $\pi(y)$ is a new
          individual not present in $\mathcal{K}$
(4)       $\mathcal{S} \leftarrow \{C(\pi(u)) | C(u) \in q_i\} \cup \{R(\pi(u), \pi(v)) | R(u, v) \in q_i\}$
(5)       if $(\mathcal{T}, \mathcal{A} \cup \mathcal{S})$ is consistent and concepts and roles in $S$ are all in $\mathcal{H}$
(6)           $\omega \leftarrow |\{\pi' | \pi' \in cert(\overline{q_i}, (\emptyset, \mathcal{A}))\} \wedge \mathcal{K} \models C(\pi'(x))|$
(7)           emitSolution( $(\mathcal{S}, \omega/\Omega)$ )
```

computeOmega($\mathcal{K} = (\mathcal{T}, \mathcal{A}), A, C$)
Input: $\mathcal{K} = (\mathcal{T}, \mathcal{A})$ a DL-Lite$_\mathcal{R}$ knowledge base, A is an atomic concept, C is a concept description or $\top$
Output: $|\Omega(\mathcal{K}, A|C)|$

```
(1)   $pref \leftarrow PerfectRef(x \leftarrow A(x), \mathcal{T})$
(2)   $r \leftarrow 0$
(3)   foreach $q_i$ in $pref$
(4)       foreach $\pi$ in $cert(\overline{q_i}, (\emptyset, \mathcal{A}))$
(5)           $new \leftarrow true$
(6)           $\mathcal{J} \leftarrow construct(q_i, \pi)$
(7)           // (8)-(10) ensure that $\mathcal{J}$ is not counted if it was previously discovered
              by $q_j$ for $j < i$
(8)           foreach $q_j$ in $pref$ s.t. $j < i$
(9)               if $\{\pi' | \pi' \in cert(\overline{q_j}, (\emptyset, \mathcal{J}))$ and $\pi'(x) = \pi(x)\} \neq \emptyset$
(10)                  $new \leftarrow false$
(11)          if $new$ and $\mathcal{K} \models C(\pi(x))$
(12)              $r \leftarrow r + 1$
(13)  return r
```

Theorem 1 *Let $\mathcal{P} = (\mathcal{K}, \mathcal{H}, A(a))$ be a abductive problem such that $K = (\mathcal{T}, \mathcal{A})$ a DL-Lite KB. Let C be a concept description or the top concept ($\top$). Algorithm* *computeSolutions*$(\mathcal{P}, C)$ *terminates. Furthermore, if* $\mathcal{S}_0$ *is a solution to* $\mathcal{P}$*, then, for each* $\mathcal{J} \in \Omega((\mathcal{T}, \mathcal{A} \cup \mathcal{S}_0), A(a))$*, there is* $(\mathcal{S}, p) \in$ *computeSolutions*$(\mathcal{P}, C)$ *such that the following hold:*

1. $Pr_{just}(\mathcal{J}|C) \leq p$ *(i.e. the conditional likelihood of* $\mathcal{J}$ *in* K *is less than or equal to* p*)*
2. $\mathcal{J} \models^a \widehat{\mathcal{S}(a)}$ *(i.e.* $\mathcal{J}$ *is an instance of the pattern formed by* $\mathcal{S}$*)*

The proof of Theorem 1 is a consequence of the Proposition 1.

Theorem 2 *Let* $\mathcal{P} = (\mathcal{K}, \mathcal{H}, A(a))$ *be a abductive problem such that* $K = (\mathcal{T}, \mathcal{A})$ *a DL-Lite KB. Let* C *be a concept description or the top concept* ($\top$)*. Algorithm* *computeSolutions*$(\mathcal{P}, C)$ *is PTime in the size of the TBox, and LogSpace in the size of the ABox (data complexity).*

Proof. The proof follows from the following properties of DL-Lite$_\mathcal{R}$ established in [2]:

– Consistency check and instance checking (*i.e.,* checking $\mathcal{K} \models C(b)$ for an individual b) in DL-Lite$_\mathcal{R}$ is PTime in the size of the TBox, and LogSpace in the size of the ABox.
– Conjunctive query answering against a KB with an empty Tbox is LogSpace in the size of the ABox (i.e., same complexity as conjunctive query answering against a database)
– For a conjunctive query q and a Tbox $\mathcal{T}$, the maximum size of $PerfectRef(q, \mathcal{T})$ is $(m(n+1)^2)^n$, where m is the size of the Tbox and n the size of the query q (i.e., the number of atoms in *body*(q)). Therefore $|PerfectRef(x \leftarrow A(x), \mathcal{T})| \leq 4 \times m$
– For a conjunctive query q and a Tbox $\mathcal{T}$, if $q' \in PerfectRef(q, \mathcal{T})$ then the number of atoms in q' is at most the same as the number of atoms in q. Therefore, if $q' \in PerfectRef(x \leftarrow A(x), \mathcal{T})$ then q' has at most one atom.

7 Related Work

Abduction in logic programming without probabilities has attracted a lot of attention, and several algorithms, including meta-interpreters written in Prolog, have been made [10, 13]. However, probabilistic abductive logic programming has not been studied nearly to the same extent. Poole proposed a probabilistic abduction approach for horn logic [17]. This approach considers a logic programming approach that uses a mix between depth-first and branch and bound search strategies for abduction where the probabilities are considered and only the most likely explanations are generated. Henning has proposed an approach for probabilistic abductive Logic programming with constraint handling rules [3]. This approach differs from other approaches to probabilistic logic programming by having both interaction with external constraint solvers and integrity constraints. Henning used probabilities to optimize the search for explanations using Dijkstra's shortest path algorithm. Hence, the approach explores always the most probable direction, so that investigation of less probable alternatives is suppressed or postponed. For plan recognition tasks represented in datalog, Raghavan and Mooney proposed to use Bayesian networks while estimating probabilities of abductive explanations [18]. They suggest to learn a Bayesian network from abductive explanations using structure learning techniques. Once the network structure is determined, the parameters of the network are learned using an external training set.

There are only a few works in the literature for abductive reasoning in *DLs*. However, unlike our approach, none of these works estimates probabilities for the computed abductive explanations. For *TBox* abduction, Hubauer *et al.* proposed automata-based approach [7] while Noia *et al.* proposed an approach exploiting tableaux algorithms for DLs [15]. For *ABox* abduction, Perald *et al.* proposed an approach based on a backward inference method [6]. It restricts axioms in the DL-based ontology to some special forms and does not use a notion of minimality for abductive solutions. Klarman *et al.* have proposed an approach for *ABox* abduction in $\mathcal{ALC}$ fragment of OWL DL [12], but this approach cannot guarantee termination. Du *et al.* have propose another approach which is based on translation of $\mathcal{SHIQ}$ into Prolog and making abductive reasoning using existing a approaches for plain datalog programs [4]. They have showed that their guarantees termination and certain minimality of results. We have followed the same approach to enumerate all abductive explanations for an *ABox* axiom.

8 Conclusions

In this paper, first we formalize probabilistic ABox abduction problem in *DL*. Then, we have proposed an approach for estimating likelihoods of abductive explanations. The proposed approach exploits the frequencies of justification patterns in *ABox* within the context of specific individuals in a knowledge base. Our evaluations show that the proposed approach significantly outperform classical abduction approach where each explanation is assumed equally likely. Our findings also highlight that the probabilistic abduction in $\mathcal{SHIQ}$ is costly, as expected. That is why, we have presented a tractable algorithm for $\textsf{DL-Lite}_{\mathcal{R}}$ at the end. As a future work, we plan to study the strength and weaknesses of the proposed approach extensively using various benchmarks.

References

1. F. Baader, D. Calvanese, D. McGuinness, D. Nardi, and P. Patel-Schneider. *The Description Logic Handbook*. Cambridge University Press, 2003.
2. D. Calvanese, D. Lembo, M. Lenzerini, and R. Rosati. Tractable reasoning and efficient query answering in description logics: The dl-lite family. *J. of Automated Reasoning*, pages 385–429, 2007.
3. H. Christiansen. Constraint handling rules. chapter Implementing Probabilistic Abductive Logic Programming with Constraint Handling Rules, pages 85–118. 2008.

4. J. Du, G. Qi, Y.-D. Shen, and J. Z. Pan. Towards practical abox abduction in large owl dl ontologies. In *The Twenty-Fifth AAAI Conference on Artificial Intelligence (AAAI-11)*, San Francisco, USA, August 2011.
5. C. Elsenbroich, O. Kutz, and U. Sattler. A case for abductive reasoning over ontologies. In *OWLED'06*, pages –1–1, 2006.
6. S. Espinosa Perald, A. Kaya, S. Melzer, R. Moller, and M. Wessel. Towards a media interpretation framework for the semantic web. In *Proceedings of the IEEE/WIC/ACM International Conference on Web Intelligence*, WI '07, pages 374–380, 2007.
7. T. Hubauer, S. Lamparter, and M. Pirker. Automata-based abduction for tractable diagnosis. In *Description Logics*, 2010.
8. U. Hustadt, B. Motik, and U. Sattler. Reasoning in description logics by a reduction to disjunctive datalog. *J. Autom. Reason.*, 39:351–384, October 2007.
9. P. D. R. K. James Blythe, Jerry R. Hobbs and R. J. Mooney. Implementing weighted abduction in markov logic. In *International Conference on Computational Semantics*, 2011.
10. A. C. Kakas, B. Van Nuffelen, and M. Denecker. A-system: problem solving through abduction. In *Proceedings of the 17th international joint conference on Artificial intelligence (IJCAI'01)*, pages 591–596, 2001.
11. A. Kalyanpur. *Debugging and Repair of OWL-DL Ontologies*. PhD thesis, University of Maryland, 2006.
12. S. Klarman, U. Endriss, and S. Schlobach. Abox abduction in the description logic $\mathcal{ALC}$. *J. Autom. Reason.*, 46:43–80, 2011.
13. P. Mancarella, G. Terreni, F. Sadri, F. Toni, and U. Endriss. The CIFF proof procedure for abductive logic programming with constraints: Theory, implementation and experiments. *Theory Pract. Log. Program.*, 9:691–750.
14. B. Motik, R. Shearer, and I. Horrocks. Hypertableau Reasoning for Description Logics. *Journal of Artificial Intelligence Research*, 36:165–228, 2009.
15. T. D. Noia, E. D. Sciascio, and F. M. Donini. A tableaux-based calculus for abduction in expressive description logics: Preliminary results. In *Description Logics*, 2009.
16. C. Patel, J. J. Cimino, J. Dolby, A. Fokoue, A. Kalyanpur, A. Kershenbaum, L. Ma, E. Schonberg, and K. Srinivas. Matching patient records to clinical trials using ontologies. In *ISWC/ASWC*, pages 816–829, 2007.
17. D. Poole. Probabilistic horn abduction and bayesian networks. In *Artificial Intelligence*, 1993.
18. S. Raghavan and R. Mooney. Bayesian abductive logic programs. In *Proceedings of the AAAI Workshop on Statistical Relational AI*, pages 82–87, 2010.
19. E. Sirin, B. Cuenca Grau, and B. Parsia. From wine to water: Optimizing description logic reasoning for nominals. In *Proceedings of KR-2006*, pages 90–99, 2006.
20. E. Sirin and B. Parsia. Pellet: An owl dl reasoner. In *Description Logics*, 2004.
21. L. Sterling and L. U. Yalcinalp. Explaining prolog based expert systems using a layered meta-interpreter. In *Proceedings of the 11th international joint conference on Artificial intelligence*, pages 66–71, 1989.
22. E. Thomas, J. Z. Pan, and Y. Ren. TrOWL: Tractable OWL 2 Reasoning Infrastructure. In *the Proc. of the Extended Semantic Web Conference (ESWC2010)*, 2010.

Ontology Based Data Integration Over Document and Column Family Oriented NOSQL stores

Olivier Curé[1], Myriam Lamolle[2], Chan Le Duc[2]

[1] Université Paris-Est, LIGM, Marne-la-Vallée, France
ocure@univ-mlv.fr
[2] LIASD Université Paris 8 - IUT de Montreuil
{myriam.lamolle, chan.leduc}@iut.univ-paris8.fr

Abstract. The World Wide Web infrastructure together with its more than 2 billion users enables to store information at a rate that has never been achieved before. This is mainly due to the will of storing almost all end-user interactions performed on some web applications. In order to reply to scalability and availability constraints, many web companies involved in this process recently started to design their own data management systems. Many of them are referred to as NOSQL databases, standing for 'Not only SQL'. With their wide adoption emerges new needs and data integration is one of them. In this paper, we consider that an ontology-based representation of the information stored in a set of NOSQL sources is highly needed. The main motivation of this approach is the ability to reason on elements of the ontology and to retrieve information in an efficient and distributed manner. Our contributions are the following: (1) we analyze a set of schemaless NOSQL databases to generate local ontologies, (2) we generate a global ontology based on the discovery of correspondences between the local ontologies and finally (3) we propose a query translation solution from SPARQL to query languages of the sources. We are currently implementing our data integration solution on two popular NOSQL databases: MongoDB as a document database and Cassandra as a column family store.

1 Introduction

The distributed architecture of the World Wide Web and its more than 2 billion users, in 2011, enables to store vast amount of information from end-user interactions. The volumes of data retrieved this way are so large that it motivated the design and implementation of new data models and management systems able to tackle issues such as scalability, high availability and partition tolerance. In fact, the Web helped us to understand that the until now prevalent relational model does not fit all the data management issues [24].

These new data stores are regrouped under the NOSQL label (but coSQL [17] is another recently proposed name). This acronym stands for 'Not Only SQL' and generally identifies data stores based on the Distributed Hash Table (DHT) model which provides a hash table access semantics. That is in order to access and modify a data object, a client is required to provide the key for this object and a management system will lookup the object using an equality match to the required attribute key. The First successful NOSQL databases were developed by Web companies like Google (with Big Table [6]) and Amazon (Dynamo [9]). An important number of open source projects followed more or less inspired by these two systems, e.g. MongoDB[3], Cassandra[4] which respectively correspond to the document and column family categories. Nowadays, NOSQL systems are used in all kinds of application domains (e.g. social networks, science, finance) and are present in cloud computing environments. Hence, we consider that the Web of Data can not miss the opportunity to address and integrate technologies and datasets emerging from this ecosystem.

In this paper, we propose a data integration framework where the target schema is represented as a semantic web ontology and the sources correspond to NOSQL databases. The main difficulty in integrating these data sources concerns their schemalessness and lack of a common declarative query language.

Concerning the schemalessness, although this provides for a form of flexibility in term of data modeling, this makes the generation of correspondences between a global and local schemata more involved. Thus a first contribution of our work consists in generating a local schema for each integrated source using an inductive approach. This approach uses non-standard description logic (DL [2]) reasoning services like Most Specific Concept (MSC) and Least Concept Subsumer (LCS) in order to generate a concept for a group of similar individuals and to define hierarchies for these concepts. Our second contribution enables the specification of a global ontology based on the local ontologies generated for each data source. This global ontology results from the correspondences discovered between concept definitions present in each local ontology.

Concerning the lack of a common declarative query language, we propose a Bridge Query Language (BQL) that supports a translation from SPARQL queries expressed over the global ontology to the possibly different query languages accepted at the sources. In general, document and column family databases do not provide for a declarative query language like SQL. They rather propose a procedural approach based on the use of specific APIs, for instance for the Java language. Hence our last contribution is to present the main steps involved in this transformation and to provide a sketch of the BQL language.

This paper is organized as follows. In Section 2, we present related works in ontology based data integration. Section 3 provides some background knowledge on NOSQL databases, non-standard DL reasoning services and some alignment methods. Section 4 details our contributions in the design of our ontology-based data integration system and thus provides for an overview of this system's architecture. In Section 5, we present the query processing solution adopted for our system. Section 6 concludes the paper and gives perspectives on future works.

[3] http://www.mongodb.org/
[4] http://cassandra.apache.org/

2 Related work

To the best of our knowledge, this paper is a first attempt to integrate data stored in NOSQL systems into an ontology based framework. Hence, in this section, we focus on the broader subject of ontology-based data integration and concentrate on solutions addressing the relational model. Most of the work dedicated to bridging the gap between ontologies and relational databases concentrated on defining mapping languages, query answering and its relationship with reasoning over the ontology.

MASTRO [5] is the reference implementation for the Ontology-Based Data Access (OBDA) approach. In OBDA, ontologies, expressed in Description Logics, represent the conceptual layer of the data stored in relational databases. It allows for both sound and complete conjunctive query answering over an ontology by retrieving data from a relational databases. Most of the nice properties of MASTRO come from the computational characteristics of DL-Lite which motivated the creation of OWL2QL, an OWL2 fragment. Nevertheless, MASTRO requires that both the global ontology and relational schemata are known in order to define semantic mappings.

In [10], the SHER system is presented as a system for scalable conjunctive query answering over $\mathcal{SHIQ}$ ontologies where the ABox is stored in a relational database management system. A main contribution of this work is to implement an ABox *summarization* technique which improves the computational performances of query answering.

In the Maponto tool [1], the authors propose a solution that enables to define complex mappings from simple correspondences. This approach expects end-users or an external software to provide mappings and then uses them to generate new ones. Maponto is being provided with a set of relational databases and an existing ontology.

Systems like MARSON [14] and RONTO [19] discover simple mappings by classifying the relations of a database schema and validate the mapping consistency that have been generated. Like Maponto, these systems require that the target ontology is provided.

In comparison with these systems, our approach deals with the absence of a schema at the sources and of global ontology. Moreover, while all systems based on a relational model benefit from the availability of SQL, the existence of a common query language for the sources can not be assumed in the context of NOSQL databases.

3 Background

In this section, we present background knowledge concerning the two NOSQL databases we are focusing on in this paper, namely document and column-oriented stores. This is motivated by their ability to provide an efficient solution to the scalability issue by enabling to scale out quickly. For both of these approaches, we model a similar use case dealing with the submission and reviewing process of scientific conferences. Concerning ontology related operations, we present non-standard reasoning services encountered in DL, i.e. MSC, LCS and GCS, and provide information on methods used to align expressive ontologies.

3.1 Document oriented databases

Document oriented databases correspond to an extension of the well-known key-value concept where in this case the value consists of a structured document. A document contains hierarchically organized data similar to XML and JSON. This permits to represent one-to-one as well as one-to-many relationships in a single document. Therefore a complex document can be retrieved or stored without using joins. Since document oriented databases are aware of stored data, it enables to define document field indexes as well as to propose advanced query features. The most popular document oriented databases are MongoDB (10gen) and CouchDB (Apache).

Example 1 This document database (denoted `docDB`) stores data in 2 collections, namely `Person` and `Document`. In the `Person` collection, documents are identified by the email address of the person and contains information regarding the last name, first name, url, university, person type (i.e. either a user, author, conference member or reviewer) and possibly a list of reviewed document identifiers. The documents in the `Document` collection are identified by a 'doc' prefix followed by a unique numerical value. For each document, the system stores the title, email of the different authors (corresponding to keys in the `Person` collection), the abstract and full content of the paper. Finally, a list of reviews is stored for each document. Fig. 1 presents a graphical representation of a document for each collection. In this database, the reviews of a paper are stored within the paper document. This is easily structured in a document store which generally supports the nesting of documents. Similarly, the documents a person needs to review are stored in `Person` documents, i.e. in `writeReview`.

3.2 Column-family databases

Column family stores correspond to persistent, sparse, distributed multilevel hash maps. In column family stores, arbitrary keys (rows) are applied to arbitrary key value pairs (columns). These columns can be extended with further arbitrary key value pairs. Afterwards, these key value pair lists can be organized into column families and keyspaces. Finally, column-family stores can appear in a very similar shape to relational databases on the surface. The most popular systems are HBase and Cassandra. All of them are influenced by Google's Bigtable.

Example 2 Considering the kind of queries one can ask on this column family (denoted `colDB`), the structure consists of 3 columns families: `Person`, `Paper` and `Review`. The set of information stored in these column families is the

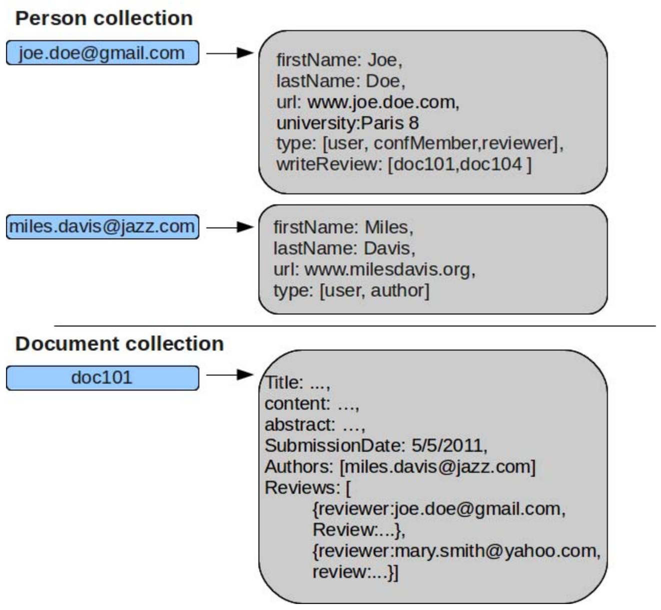


Fig. 1. Extract of the document oriented database

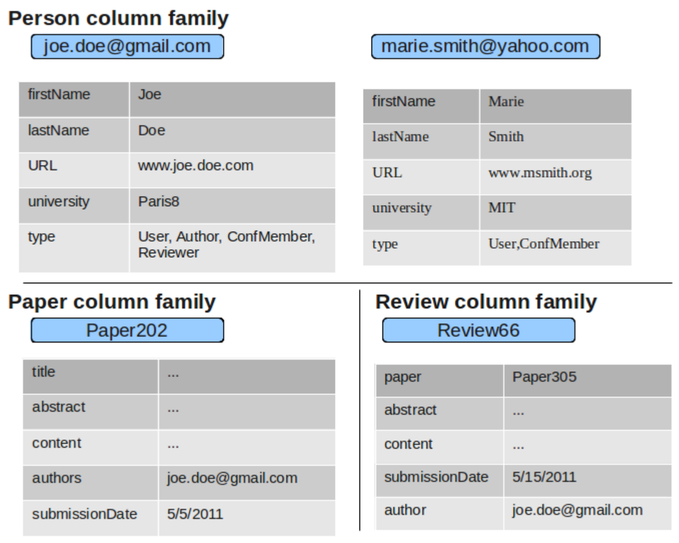
same as in Example 1. The row key for the `Person`, `Paper` and `Review` are respectively the email address of the person and system generated identifiers for papers and reviews. All other information entries are stored in columns with some of them being multi-valued. Fig. 2 provides a graphical representation of an extract of `colDB`. The `Paper` and `Review` column families have several columns in common (`abstract`, `content` and `submissionDate`). But while the `authors` column stores the list of authors of a paper, the `author` column of reviewer stores the identifier of its reviewer.

3.3 Non-standard Reasoning Services

We present in this section the basic versions of *Least Common Subsumer* (LCS) [18] , *Most Specific Concept* (MSC) [2] and *Good Common Subsumer* (GCS) [4]. The *MSC* of an individual consists in defining the least concept description that the individual is an instance of.

Definition 1 Given concept terms $C_1, ..., C_n$, the *MSC* of an individual a is a concept term C iff
- $C \sqsubseteq C_i$, for $1 \leqslant i \leqslant n$;
- C is the most specific concept term with this property, i.e., if D is a concept term such that $C_i \sqsubseteq D$ for $1 \leqslant i \leqslant n$, then $C \sqsubseteq D$.

And, the *LCS* of a set of concepts is the least concept that subsumes all of them, i.e., there is no sub-concept of this *LCS* that subsumes the set of concepts too.



Person column family

joe.doe@gmail.com

firstName	Joe
lastName	Doe
URL	www.joe.doe.com
university	Paris8
type	User, Author, ConfMember, Reviewer

marie.smith@yahoo.com

firstName	Marie
lastName	Smith
URL	www.msmith.org
university	MIT
type	User,ConfMember

Paper column family

Paper202

title	...
abstract	...
content	...
authors	joe.doe@gmail.com
submissionDate	5/5/2011

Review column family

Review66

paper	Paper305
abstract	...
content	...
submissionDate	5/15/2011
author	joe.doe@gmail.com

Fig. 2. Extract of the column family database

Definition 2 Given concept terms $C_1, ..., C_n$, the LCS of $C_1, ..., C_n$ is a concept term C such that

- $C_i \sqsubseteq C$ for $1 \leqslant i \leqslant n$;
- C is the least concept term with this property, i.e., if D is a concept term such that $C_i \sqsubseteq D$ for $1 \leqslant i \leqslant n$, then $C \sqsubseteq D$.

But, the *LCS* is very hard to process in practice. So, Baader [4] proposes an algorithm named *Good Common Subsumer (GCS)* to compute an approximation of *LCS* by determining the smallest conjunction of (negated) concept names subsuming the conjunction of the top level concept names of each considered concept. By computing the *MSC* and *LCS* of these individuals, more complex concept descriptions can be added to the ontology [3].

3.4 Alignment methods

The heterogeneity between ontologies must be reduced in order to facilitate interoperability of applications based on these ontologies. For this purpose, semantic correspondences between different entities belonging to two different ontologies are required to be established. This is the goal of ontology alignment as presented in [12]. An alignment consists of a set of correspondences between pairs of ontology entities. Two entities of each pair are connected by a semantic relation (e.g. equivalence, subsumption, incompatibility, etc.). Moreover, a similarity measure can be associated to each correspondence to specify its trust. Then, a set of correspondences (i.e. alignment) can be used to merge ontologies, migrate data or translate queries from one to another ontology.

In the literature, there are several alignment methods that can be categorized according to techniques employed to produce alignments. The most early methods are based on the comparison of linguistic expressions [11]. Another aligner presented in [8] has taken into account annotations of entities defined in ontologies. More recently, the methods introduced in [15], [22], [16] have exploited ontological structures related to concepts in question. These methods, namely simple alignment methods, are the most prevalent at present. They detect simple correspondences between atomic entities (or simple concepts) (e.g. $Human \sqsubseteq Person, Female \sqsubseteq Person$). As a result, some kinds of semantic heterogeneity in different ontologies can be solved by using these classical alignment methods.

However, simple correspondences are not sufficient to express relationships that represent correspondences between complex concepts since (i) it may be difficult to discover simple correspondences (or they do not exist) in certain cases, or (ii) simple correspondences do not allow for expressing accurately relationships between entities.

A second important issue is that generating a complex alignment has a certain impact during the consistency checking of the system. Indeed, a reasoner such as Pellet [23] or FaCT++ [25], running on a system consisting of two ontologies O_1 and O_2 and a simple alignment A_s, may reply that the system is not consistent. But, this same reasoner, with the same ontologies O_1 and O_2, and with a complex alignment A_c can deduce that the system is consistent.

Consequently, new works follow the way of complex alignment solutions such as [20]. But, currently, they address the alignment of simple concept with a complex concept, at best.

4 Architecture overview

In this section, we present the main components of our system and highlight on the approaches used at each steps of the data integration processing. These steps, depicted in Fig. 3, consist of the (1) creation of an ontology associated to each data sources, (2) aligning these ontologies and (3) creating a global ontology given these correspondences.

Finally, we present a query language enabling to retrieve information stored in the sources from a query expressed over the global ontology.

4.1 Source ontology generation

As explained earlier, NOSQL databases are generally schemaless. Although this provides flexibility for information storage, it makes the generation of associated ontologies more involved. In fact, one can only use containers, i.e. collections and column families in respectively document and column family databases, of key/value pairs as well as key labels to deduce a schema. Our approach considers that each container defines a DL concept and that each key label corresponds to a DL property that can either be a data type or object one and whose domain is the DL concept corresponding to its container.

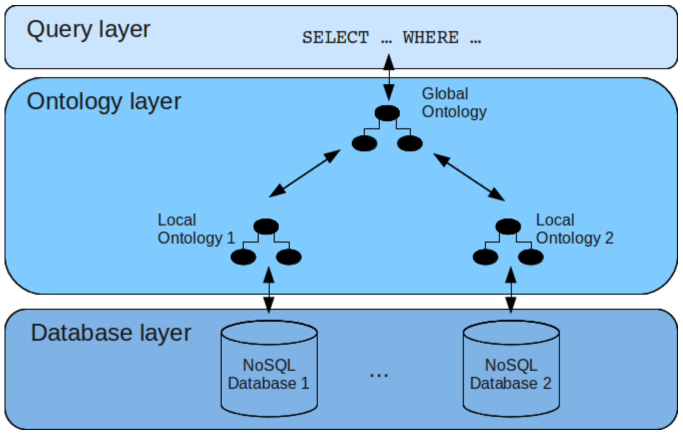


Fig. 3. Basic architecture of our data integration system

Example 3 Consider Example 1 (resp. Example 2), the following concepts are automatically generated: `Person` and `Document` (resp. `Person`, `Paper` and `Review`). Concerning DL properties, a `firstName` DL datatype property will be created in the cases of both Examples 1 and 2 with a domain corresponding to the `Person` DL concept. Additionally, a `writeReview` DL object property is created with a domain and range corresponding to respectively the `Person` DL and `Paper` concepts. This is due to the fact that the values of the `writeReview` (doc101 and doc 104 in the case of the document identified by joe.doe@gmail.com) served as identifier of other documents. The same approach applies for `authors`, `author` and `paper` in Example 2.

A modeling pattern frequently encountered in key/value stores supports the discovery of complementary DL concepts and some subsumption relationships. This pattern, henceforth denoted *type definition*, consists of a key whose range of values is finite and which do not correspond to container identifiers, i.e. they do not serve as foreign keys. We assume that each of these values specifies a DL concept. For instance, this is the case of the `type` key in respectively Examples 1 and 2. Its set of possible values is {User, Author, ConfMember and Reviewer}, each of them corresponding to a DL concept. These concepts can be organized into a hierarchy of concepts using methods of Formal Concept Analysis (FCA) [13]. In a recent paper [7], we have emphasized on an FCA methodology for ontology mediation. Some features of this method are to create concepts that are not in the source ontologies, to label the new concepts, and to optimize the resulting ontology by eliminating redundant or irrelevant concepts. This approach easily applies to the discover of DL concepts and their subsumption relationships in the context of a *type definition* pattern. That is, tuples of the key of the pattern (`type` in our example) correspond to objects in the FCA terminology and their values provide FCA attributes. Then a Galois connection lattice can easily be computed using the methods proposed in [7]. The nodes of this lattice correspond to DL concepts and arrows between them specify subsumption relationships.

Example 4 We consider the document database of Fig.1. The document identified by key 'joe.doe@gmail.com' has several `type` values (User, ConfMember, Author and Reviewer) while the document identified by 'miles.davis@jazz.com' is only characterized by the User value. Using the information coming from different documents, one can discover the following DL concept subsumptions: $Author \sqsubseteq User$
$Reviewer \sqsubseteq User$
$ConfMember \sqsubseteq User$

The method we have presented so far can be applied recursively to embedded structures where the nested container is reified into an object.

At this point in the local ontology generation process, we have created an ontology that is no more expressive than RDFS. We consider that using induction over the instances of the source database, we can enrich the ontology and leverage its expressiveness to a fragment of OWL2, namely OWL2EL. This is performed using the approach proposed in [21] to compute the GCS wrt to local ontology computed. This method exploits the TBox of the ontology and precomputes the conjunction of concept names using FCA. One issue in this precomputation is to handle a possibly very large set of FCA objects.

Ganter's *attribute exploration* interactive algorithm [13] is an efficient approach for computing an appropriate representation of a concept lattice that at certain stages asks contextualized questions to a domain expert. Instead of relying on this interactive process, we propose other solutions that may be used to select a subset of relevant objects. In some cases, the set of objects may be of a reasonable size, (e.g. fitting into main memory) and a complete analysis is possible. Nevertheless, in many situations, due to the size of individual data, a complete analysis is not realistic and some heuristics need to be proposed. The first naive approach one can think of is to randomly access a set of individuals. Apart from the hazardous results this approach could provide, it is not just doable in hash table context where the key of the container needs to be known a priori.

A simple heuristic consists in considering that the most frequently accessed individuals are the most representative of the ontology to generate. In order to discover this set, one can take advantage of the data store architecture, generally distributed over several servers and supervised by several tools such as load balancers. Using logs generated by these tools enables to identify a subset of the individuals that are the most frequently accessed in the application.

Finally an incremental schema generation approach can be implemented. That is each time a tuple is inserted or modified, the system checks if some labels are being introduced or deleted into the schema. This approach imposes that each update operation goes through this process.

At the end of this step, using an inductive approach, we have created a schema for each NOSQL source. The goal of this schema is twofold: it enables the creation of DL ontology which can be serialized into an OWL2 fragment (namely OWL2EL) and supports the definition between ontology entities (i.e. DL concepts and properties) with elements of the NOSQL source (i.e. documents, column families, columns and keys). Hence, the arrows linking the database and ontology layers of Fig. 3 have been generated. The task of the next section is to generate a global ontology via the discovery of alignments between local ontologies.

4.2 Discovering Alignments between ontologies and global ontology building

We now propose a new solution to detect both simple and complex correspondences. To do this, we follow several steps. The first step consists in enriching the two ontologies to be aligned using the IDDL reasoner [26]. This reasoner allows to add subsumption relations which are implicit in ontologies (see Fig.4).

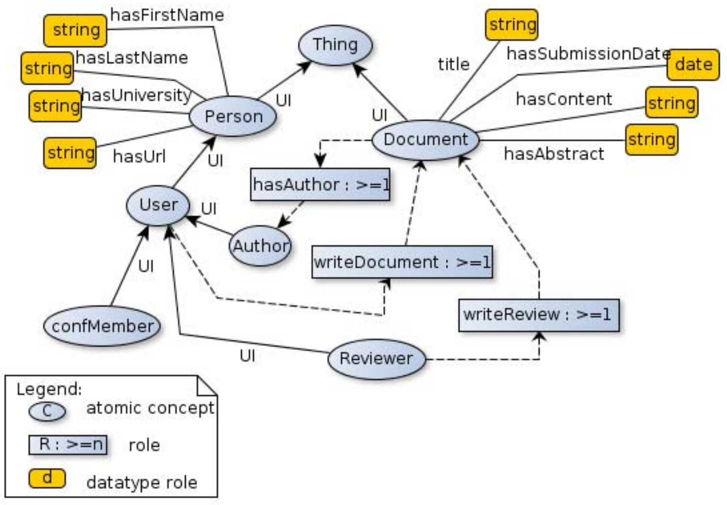


Fig. 4. Extract of review document family DB graph

The second step detects the simple correspondences using three classical alignment processes. We use the three conventional aligners OLA[5] [15], AROMA[6][8] et WN[7]. Each of them is based on a particular approach. The first aligner is a basic aligner, which uses the linguistic resource *WordNet*, the second is based on a structural approach and the third on the annotations associated to entities.

[5] OWL-Lite Alignment

[6] Association Rule Ontology Matching Approach

[7] basic aligner of API alignment named JWNLAlignment

Note that the last two chosen aligners are considered by OAEI[8] among the best alignment systems.

The last step detects the complex correspondences. Our idea is inspired from simple alignment methods which are based on graphs [15],[22]. Since, from a finite vocabulary, an infinite number of formulas can be processed, it is impossible to know what are the relevant formulas to align. A possible solution is to try to capture the semantic of OWL and to represent the constructors (for example, subsumption, disjunction, restriction of cardinality) by a graph formalism. Then, from the two graphs representing the ontologies to be aligned, it must search relevant subgraphs which can be aligned taking into account their structures and using a terminological similarity measure.

Proposition 1 Our first proposition allows correspondences between two complex concepts (i.e. formulas) to be detected.

For example, $Paper \sqcap \exists write.Author \sqcap \exists accept.Reviewer \sqsubseteq Paper \sqcap \exists write.Author$ can be deduced. The detection of this kind of matching is made by exploiting the structure of graphs, which expresses the semantics of the OWL-DL ontologies. Each graph consists of a set of subgraphs, which represent a formula. So, for all pairs of concepts (C_1, C_2) belonging to the ontologies (O_1, O_2), it is necessary to check whether their respective subgraphs can be aligned.

A subgraph of a given concept C consists of all concepts directly linked to C by simple edges (subsumptions or disjunctions) or properties.

To align two subgraphs, one of the following cases must be checked:

1. The first subgraph SG_1 subsumes the second subgraph SG_2 (i.e. $SG_1 \sqsupseteq SG_2$). In this case, a relation of subsumption is generated;
2. The second subgraph SG_2 subsumes the first subgraph SG_1 (i.e. $SG_1 \sqsubseteq SG_2$). In this case, a relation of subsumption is generated, in the opposite way of case 1;
3. The two subgraphs are equivalent. In other words, SG_1 subsumes SG_2 and SG_2 subsumes SG_1. In this case, a relation of equivalence is generated (i.e. $SG_1 \equiv SG_2$).

A subgraph SG_1 subsumes a subgraph SG_2 if the following conditions hold:

- All direct subclasses of SG_1 are similar to direct subclasses of SG_2,
- All disjoint subclasses of SG_1 are similar to disjoint subclasses of SG_2,
- All direct super classes of SG_1 or their generalization are similar to direct super classes of SG_2 or their generalization,
- All direct properties of SG_1 are similar to direct properties of SG_2,
- All properties cardinalities of SG_1 are equivalent or subsumed by properties cardinalities of SG_2,
- All domains or co-domains of these properties of SG_1 are similar to domains or co-domains or their generalizations in SG_2.

[8] Ontology Alignment Evaluation Initiative

Proposition 2 This proposition allow us to detect correspondences between a simple concept and a formula (e.g. $SubmittedPaper \sqsubseteq \exists submit.Author$). It is inspired by the research work presented in [20] with some simplification and generalization. We search correspondences between simple concepts and formulas based on syntactic similarities between concepts and properties. It is necessary to use a purely syntactic similarity measure to compare concepts labels to properties labels. Moreover, a concept to align with a formula having a property similar syntactically must be a concept specializing a concept already aligned to a concept source or target of this property (or one of its super concept).

To generate the complex correspondences detected by these two propositions, we used the language EDOAL[9], which extends the alignment format proposed by INRIA. This language can express complex structures between entities of different ontologies.

Example .

Given two ontologies O_1 and O_2 built from NOSQL databases to be aligned concerning a conference domain. OWL semantics of these ontologies are represented by graphs (cf. Fig. 4 of O_1 from example 1) .

The following simple correspondences are detected during the first step:

$O_1 : Document \equiv O_2 : Document$
$O_1 : Person \equiv O_2 : Person$
$O_1 : Reviewer \equiv O_2 : Referee$
$O_1 : Review \equiv O_2 : Review$
$O_1 : Conference \equiv O_2 : Conference$
$O_1 : Submit \equiv O_2 : Submit$
$O_1 : WriteReview \equiv O_2 : WriteReview$
$O_1 : ConfMember \equiv O_2 : ConfMember$

The second step consisting in traversing the relevant subgraphs detects complex correspondences such as these presented in Fig. 5 and 6. To do this, the neighborhood of the graphs nodes are considered. For example, the generated correspondence from the two subgraphs in Fig. 5 having respectively the nodes $O_1 : Paper$ and $O_2 : Published$ as starting point is:

$O_1 : Paper \sqcap \geqslant 1\ O_1 : hasAuthor.O_1 : contactPerson \sqcap \geqslant 1\ O_1 : Submit.O_1 : contactPerson \sqsupseteq O_2 : Published \sqcap \geqslant 1\ O_2 : Submit \geqslant .O_2 : Author \sqcap \geqslant O_2 : isAuthorOf.O_2 : Author \sqcap O_2 : AcceptedBy.O_2 : ComitteMember$

In the same way, the generated correspondence from the two subgraphs of Fig. 6 having respectively the nodes O_1 : *Reviewer* and O_2 : *Referee* as starting point is:

$O_1 : Reviewer \sqcap \geqslant 1\ O_1 : WriteReview.O_1 : Review \sqsubseteq O_2 : Referee \sqcap \exists\ WriteReview.O_2 : Review$

[9] http ://alignapi.gforge.inria.fr/edoal.html

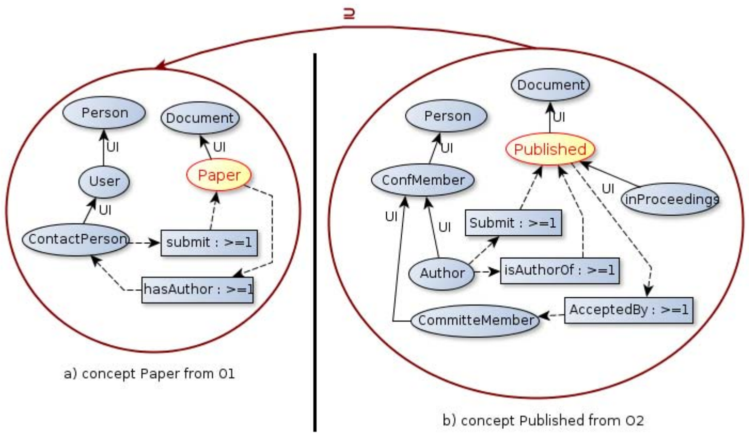


Fig. 5. Aligned subgraphs of concepts $O_1 : Paper$ and $O_2 : Published$

Taking into account the ontologies representing NOSQL data sources and their alignments, a global ontology GO is built. $GO = (O, A)$ represents networked ontologies $0 = \{0_1, ..., O_n\}$ through a set of alignments $A = \{A_1, ..., A_m\}$ where A_i is the set of correspondences between 0_k and $0_l (k \neq l)$.

5 Query processing

In this section, we present the query processing solution adopted in our ontology-based data integration system. The approach consists in two consecutive translation operations.

The first one transforms end-user written SPARQL queries expressed over the global ontology into a set of queries specified in the Bridge Query Language (BQL). This translation uses the correspondences discovered during the local and global ontology generation steps and occurrences of a set of RDF/S properties (e.g. `rdf:type`, `rdfs:subClassOf`) in SPARQL queries. Given these correspondences, a BQL query is generated over the local ontology of a NoSQL source. Due to space limitations, we do not provide a thorough presentation of BQL but rather sketch its main features. BQL is a high-level declarative query language and has low-level, procedural programming flavor that enables to retrieve information from data repositories. In fact, a BQL program is similar to specifying a query execution plan that can easily be translated into fully procedural programs satisfying a given API and programming language. Following a nested data model, a BQL program specifies a sequence of steps that each define a single high level data operation. Like a relational algebra, each step is specified via a relation definition which can serve as the input to another step. A main construct of BQL is a `foreach .. in` operation which permits to iterate other a defined relation and perform some associated operations. These operations

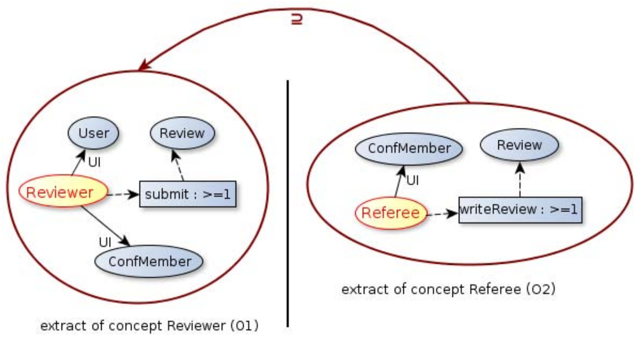


Fig. 6. Aligned subgraphs of concepts $O_1 : Reviewer$ and $O_2 : Referee$

generally consist in retrieving information from the database. This is specified using a `get` operation defined over a given database and container. It contains 2 parameters: a set of filters expressed over source keys with standard comparator (e.g. =, <, <=, ≠) and a set of attributes to retrieve from the resultset. In Example 5, we highlight a SPARQL to BQL transformation given our conference document database.

Example 5 Consider a query that retrieves titles of reviews written by a person with last name Doe. This corresponds to the following SPARQL query which is simplified for readability reasons:

```
SELECT ?t WHERE {?p rdf:type Person. ?p hasLastName 'Doe'.
?p writeReview ?r. ?r hasTitle ?title.}
```

The presence of a `rdf:type` property in the SPARQL query provides some information about which source database and container we can create a BQL query for. The query specifies that the ?p variable must of type `Person` which is mapped to the `person` container of the document NoSQL database. This query addresses a list of reviews hence an iteration needs to be performed over the `writeReview` attribute of the `Person` container. This first step of the BQL query is written as the following:

```
temp(paper) = docDB.Person.get({lastName='Doe'},{writeReview})
```

Intuitively, the `temp` relation stores the list of review identifiers written by the person whose last name is 'Doe'. The final result of the query is provided by the `ans` relation:

`ans(title) = foreach paper in temp : docDB.Paper.get({Key=paper}, {title})`. That is for each identifier in the `temp` relation, find documents in the `Paper` collection of the docDB database and retrieve its title.

The second translation corresponds to generating a program in a given programming language (e.g. Java) from the different relations of a BQL query. Given the procedural flavor of BQL, this translation is relatively straight forward but one set of rules needs to be defined for each language and each NoSQL database.

So far, we have implemented rules for the Java language for both the MongoDB and Cassandra stores. In the future, we aim to define such rules for more NoSQL stores and programming languages (e.g. Python, Ruby).

6 Conclusion

This paper tackles the problem of integrating data stores in two of the most popular NOSQL database categories, i.e. document and column family oriented stores, in a Semantic Web context. It is well recognized that scalability is a main issue for these systems. The most involved aspect of this integration concerns the fact that these databases are schemaless and generally lack a common declarative query language. Addressing this first issue, we emphasized that using existing techniques like FCA together with non-standard DL inferences like GCS, we could compute an ontology from the structure and instances of each databases source. Using a novel alignment ontology method, we highlighted that these ontologies can be linked to create a global ontology over which SPARQL queries are expressed. Finally, a bridge query language supports a translation approach to generate procedural queries, using specific APIs for each database source, from SPARQL queries. We have already implemented this translation for the Java language for both the MongoDB and Cassandra NOSQL databases and we are currently working on query optimisation. Recently, several propositions for a common NOSQL declarative query language are emerging (e.g. CQL for Cassandra, unQL for CouchDB). Studying these specifications is on our list of future works. Nevertheless, we consider that our data integration framework is not complete until we incorporate another category of NOSQL stores: graph databases.

References

1. Y. An, A. Borgida, and J. Mylopoulos. Inferring complex semantic mappings between relational tables and ontologies from simple correspondences. In *OTM Conferences (2)*, pages 1152–1169, 2005.
2. F. Baader, D. Calvanese, D. L. McGuiness, D. Nardi, and P. Patel-Schneider. *The Description Logic Handbook: Theory, Implementation, Applications.* Cambridge University Press, Cambridge, UK, 2003.
3. F. Baader, R. Ksters, and R. Molitor. Computing least common subsumers in description logics with existential restrictions. pages 96–101. Morgan Kaufmann, 1999.
4. F. Baader, B. Sertkaya, and A. yasmin Turhan. Computing the least common subsumer w.r.t. a background terminology. In *Journal of Applied Logic*, pages 400–412. Springer, 2004.
5. D. Calvanese, G. D. Giacomo, D. Lembo, M. Lenzerini, A. Poggi, M. Rodriguez-Muro, R. Rosati, M. Ruzzi, and D. F. Savo. The mastro system for ontology-based data access. *Semantic Web*, 2(1):43–53, 2011.
6. F. Chang, J. Dean, S. Ghemawat, W. C. Hsieh, D. A. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. Gruber. Bigtable: A distributed storage system for structured data (awarded best paper!). In *OSDI*, pages 205–218, 2006.

7. O. Curé and R. Jeansoulin. An fca-based solution for ontology mediation. *JCSE*, 3(2):90–108, 2009.
8. J. David, F. Guillet, and H. Briand. Association rule ontology matching approach. *International Journal Semantic Web Information Systems*, 2:27–49, 2007.
9. G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels. Dynamo: amazon's highly available key-value store. In *SOSP*, pages 205–220, 2007.
10. J. Dolby, A. Fokoue, A. Kalyanpur, E. Schonberg, and K. Srinivas. Scalable highly expressive reasoner (sher). *J. Web Sem.*, 7(4):357–361, 2009.
11. J. Euzenat. An api for ontology alignment. *In 3rd conference on international semantic web conference (ISWC)*, pages 698–712, 2004.
12. J. Euzenat and S. Pavel. *Ontology matching*. Springer Verlag, Heidelberg (DE), 2007.
13. B. Ganter and R. Wille. *Formal concept analysis - mathematical foundations*. Springer, 1999.
14. W. Hu and Y. Qu. Discovering simple mappings between relational database schemas and ontologies. In *ISWC*, pages 225–238, 2007.
15. J.-F. Kengue Djoufak, J. Euzenat, and P. Valtchev. Alignement d'ontologies dirigé par la structure. In Y. A. Ameur, editor, *Conférence Francophones sur les Architectures Logicielles, CAL 2008*, volume RNTI-L-2 of *RNTI*, pages 155–. Cépaduès-Éditions, 2008.
16. T. Lê Bach. *Construction d'un Web sémantique multi-points de vue*. Thèse de doctorat, École des Mines de Paris à Sophia-Antipolis, 2006.
17. E. Meijer and G. M. Bierman. A co-relational model of data for large shared data banks. *Commun. ACM*, 54(4):49–58, 2011.
18. R. Möller, V. Haarslev, and B. Neumann. Semantics-based information retrieval. In *Int. Conf. on Information Technology and Knowledge Systems*, pages 48–61, 1998.
19. P. Papapanagiotou, P. Katsiouli, V. Tsetsos, C. Anagnostopoulos, and S. Hadjiefthymiades. Ronto: Relational to ontology schema matching. *AIS SIGSEMIS Bulletin*, 3(4):32–36, 2006.
20. D. Ritze, C. Meilicke, O. Šváb Zamazal, and H. Stuckenschmidt. A pattern-based ontology matching approach for detecting complex correspondences. *Proceedings of the ISWC 2009 Workshop on Ontology Matching*, 2009.
21. B. Sertkaya. A survey on how description logic ontologies benefit from formal concept analysis. *CoRR*, abs/1107.2822, 2011.
22. P. Shvaiko, J. Euzenat, F. Giunchiglia, and B. He, editors. *SODA: an OWL-DL based ontology matching system*, volume 304 of *CEUR Workshop Proceedings*. CEUR-WS.org, 2008.
23. E. Sirin, B. Parsia, B. C. Grau, A. Kalyanpur, and Y. Katz. Pellet: a pratical owl-dl reasoner. *Journal of Web Semantics*, 5(2):51–53, 2007.
24. M. Stonebraker and U. Çetintemel. "one size fits all": An idea whose time has come and gone (abstract). In *ICDE*, pages 2–11, 2005.
25. D. Tsarkov and I. Horrocks. FaCT++ description logic reasoner: System description. In *Proc. of the Int. Joint Conf. on Automated Reasoning (IJCAR 2006)*, volume 4130 of *Lecture Notes in Artificial Intelligence*, pages 292–297. Springer, 2006.
26. A. Zimmermann and C. Le Duc. Reasoning with a network of aligned ontologies. *In Proceedings of the 2nd International Conference on Web Reasoning and Rule Systems (ICWRRS)*, pages 43–57, 2008.

Resource centered RDF data management

Ralf Heese[1] and Martin Znamirowski[2]

[1] Freie Univeristät Berlin, Corporate Semantic Web
[2] Humboldt-Univeristät zu Berlin, Databases and Information Systems

Abstract. Since the first publication of RDF researchers developed systems to store and to query RDF data efficiently on secondary storage. First, they mapped the RDF data model to the relational one. As relational databases performed poorly for generic RDF graphs researches focused on native repositories using B-trees to index triples in most cases. However, native repositories still calculate the result of a query by joining single triples.
Because queries can be decomposed into star-like graph patterns we argue that managing RDF data as a set of subgraphs has advantages over existing approaches. Thus, we propose a storage model that manages an RDF graph as a collection of subgraphs and evaluates queries by joining the results of star-like subqueries. In cause of the results of some preliminary tests we are confident that a system based on our approach can efficiently process queries.

1 Introduction

The Resource Description Framework (RDF) [15] is a recommendation of the W3C that basically defines a data model for describing information about entities (resources) across system boundaries. A piece of information is represented as a triple consisting of the considered resource, a property resource, and the value for that property (a resource or a literal). Every resource is uniquely identified by a URI. The components of the triple are often referred to as subject, predicate, and object, respectively. Having unique identifiers for resources a set of triples forms a graph considering subjects and objects as nodes and predicates as edges.

Since the first publication of the RDF specification researchers developed various systems to store and to query RDF data efficiently. At the beginning they focused on mapping the RDF data model to relational one because relational database management systems (RDBMS) have been researched for many many years and perform very well in almost all application scenarios. However, researchers recognized soon that storing RDF data into a relational database raises performance issues when querying the data (e.g., results in many (self) joins). Due to the openness of the RDF data model it is also difficult to define a fixed relational schema (e.g., many multivalued and optional properties).

As a consequence, researches developed hybrid and native RDF repositories. Hybrid RDF repositories use object-oriented features of object-relational databases to transfer parts of the RDF schema into the relational schema. In contrast, native repositories do not rely on RDBMS at all but store the RDF triples in a set of indexes, e.g., (specialized) B-trees. Native stores still calculate the result of a query by iterating over the variable bindings of its triple patterns. Thus, the processing is similar to the one of joins in RDBMS.

In this paper we present a native RDF repository that manages an RDF graph as a collection of subgraphs and evaluates queries by joining the results of star-like subqueries. We consider a subquery as star-like if it contains a basic graph pattern that triple patterns have all the same subject (either resource URI or variable). In contrast to existing approaches all triples matching a star-like subqueries can be accessed by loading a single database page.

In Section 2 we give an overview of existing approaches to store RDF graphs. Afterwards, we describe our approach: We outline first the design goals of our storage model in Section 3. We describe then the storage model in detail (Section 4) and explain basic operations on RDF graphs (Section 5). In Section 6 we present an estimation of the required disc space for storing RDF data and analyze the complexity of the basic operations. In Section 7 we present and discuss preliminary results of evaluating our approach. Finally, we conclude and give an outlook to future work in the last section.

2 Related work

In this section we give a short overview of existing approaches to manage RDF data. Hereby, we concentrate on repositories that store the data on secondary storage and are designed to run on single machines (cf. Table 1). We exclude explicitely clustered approaches from our considerations because they are designed for a different use case than our storage model and, thus, other challenges are in the focus of research.

We can distinguish two main categories of RDF storage models: Relational based and native approaches (cf. Figure 1). Regarding relational storage models we can distinguish between schema oblivious, schema aware, and hybrid. *Schema oblivious* approaches rely on a single database relation storing all triples. The key advantage of this model is that any RDF model can easily be imported without considering its schema. However, the query engine has to consider the complete database for evaluating a query and large queries require expensive self-join operations. *Schema aware* approaches convert the schema of an RDF graph to a relational schema, e.g., the database contains a relation for each property of the graph. Since the data is distributed over several relations the query engine can selectively access the underlying RDF data for processing of a query; although the resulting SQL queries get more complex. Thus, it can answer queries more efficiently than in the schema oblivious case. In return for efficient query processing changes of the RDF data require sophisticated update operations because they may also affect the database schema (e.g., adding or deleting relations). *Hybrid* approaches try to combine the advantages of the two other models by using dedicated relations for certain (frequent) properties and

System	Reference	Storage model	Status
3store	[11]	schema oblivious	open source, inactive
AllegroGraph	[10]	native	commercial
Hexa-Store	[24]	native	open source
HPRD	[17]	native	scientific
Jena2 SDB	[26]	hybrid	open source
Jena2 TDB	[1]	native	open source
Kowari	[27]	native	open source, inactive
Oracle	[8]	schema oblivious	commercial
RDF-3X	[20]	triple index	scientific
RDFSuite	[2]	hybrid	inactive
RStar	[18]	hybrid	scientific
Sesame	[7]	schema aware, native	open source
System *Π*	[28]	native	scientific
Virtuoso	[21]	native	commercial
YARS2	[13]	native	scientific
N.N.	[19]	schema aware	scientific

Table 1. RDF repositories and their storage model

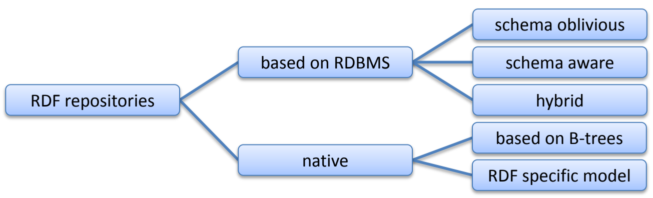


Fig. 1. Categories of RDF storage models

storing the remaining triples in a single relation. The characteristics of query processing are similar to the one in schema aware approaches. In each category there exist several variations and optimizations which we do not discuss here due to limited space.

Most *native* storage models rely on indexing triples (quadruples) in several B-trees. In [12] the author came to the result that eight indexes are sufficient to cover all possible access patterns to the data of named graphs. Only three are needed if named graphs are not considered. Besides B-trees some other systems such as RDF-3X use also other types of indexes (e.g., bitset indexes) to improve query performance and to reduce the amount of disc space occupied by indexes. So far only a few approaches considered indexing property paths separately [17] but this may change with the next version of SPARQL. Besides storage models based on B-trees there exist also approaches that propose different DBMS (e.g., deductive DBMS [25]).

In contrast to all mentioned approaches we propose a native RDF storage model that groups related triples on a database page. In addition, all triples matching a star-like subqueries can be accessed by loading a single database page. Another difference is that our indexes point to database pages containing the triples rather than storing them.

3 Design goals

Although the storage models presented in the previous section are based on very different access structures, we derived the following fundamental ideas and best practices for realizing an RDF repository.

Reduce join operations. As a consequence of the triple based data model joining triples is an essential operation. Causing significant costs during query processing it is important minimizing the number of join operations.

Normalize URIs. Managing strings unmodified, e.g., URIs and literals, causes performance problems because they are often long. On the one hand comparisons are expensive and on the other one they consume more disc space if they occur often. Thus, strings are normalized in all performant repositories.

Adequate disc consumption. Although disc space is inexpensive nowadays the system should deal with the disc space sparingly. Although replicating data for indexing purposes improves query performance the time for importing and updating data may increase.

Equal weighting of queries. A variable can occur at all positions in a triple pattern. Although some positions are more common an RDF repository should support all queries with similar performance.

Multivalued, optional properties. The semantics of RDF allow multiple values for a property of a resource [14]. In addition, it does not guarantees the presence of a property. As a result RDF graphs have often an unpredictable schema which the repository has nevertheless to handle efficiently.

Although (expensive) solid state discs allow random access to data nowadays we also aim at reducing the number of IO operations. This goal is not induced by the RDF data model but can be considered as a generally accepted rule for improving query performance in database context.

4 Storage concept

To achieve the above design goals we exploit characteristics of RDF data and its usage: (i) resources have unique identifiers, (ii) many properties are assigned to a single resource, (iii) resources of the same `rdf:type` have similar properties, and (iv) locality of queries.

The first characteristic is directly defined in the RDF specification and helps us to locate a resource easily in our repository. The second one we derived from the fact that RDF was designed to describe resources by defining their properties. Taking the well-known DBpedia dataset as an example, we derived from Table 2 in [4] that the ration between the number of resources and the number of triples is about $2.8 \cdot 10^6 : 70.2 \cdot 10^6 = 1 : 25$. Our assumption is also supported by widely accepted benchmarks SP^2Bench [23] and BSBM [5] which are based on real-world scenarios. The third point is of course only meaningful if the type is not `rdfs:Resource`. It is based on the assumption that similar entities share similar properties and, therefore, all entities are relevant to a query if one of them is. The thought behind the last one being that queries are likely to be a combination of star-like subqueries. As a consequence, a query engine has to process several triples with the same subject.

The starting point of our storage model is the lowest level of database design, the physical data organization on database pages (page for short). Developing the storage model we focused mainly on reducing the number of join operations during query processing and the number of pages loaded from secondary storage. Instead of having tuples with a predefined set of attributes on a database page as it is the case in relational databases we pay tribute to the openness of the RDF data model and store a resource (subject) together with all properties and their values on a page. Due to this organization of triples the query engine can easily determine the one page containing all triples of a given resource and does not need to perform join operations to retrieve them. In the unlikely event that the triples belonging to a resource do not fit onto a page – a 32k page can store about 8k triples – then the system creates an overflow page. As a consequence, the storage model is independent of the underlying RDF schema, e.g., the system can easily manage optional and multivalued properties. To optimize the number of IO operations further we also group resources according to their `rdf:type` because we think that queries refer often to same kind of entities. Both design decisions increase the probability that a query engine finds relevant triples on the currently processed page or on a recently loaded one.

In the following we formalize the mapping of RDF graphs to database pages. First, we introduce the terms database page and database.

Definition 1 (Database page). *A* database page p *is a persistent storage area of a predefined size* $\|\mathsf{p}\|$*. We refer to the set of all pages as the symbol* $\mathcal{P}$*.*

Each database page is assigned a unique identifier. To express that a page p contains a triple of an RDF graph, $t \in G$, we write $t \in \mathsf{p}$. We refer to the size of a page as $\|\mathsf{p}\|$ and to the number of triples per page as $|\mathsf{p}|$. We use the notations $\|t^s\|$, $\|t^p\|$ and $\|t^o\|$ to note the required space for the subject, predicate, or object of a triple t, respectively.

Definition 2 (Database). *A database* $\mathcal{D}$ *is defined as a set of database pages:* $\mathcal{D} = \{\mathsf{p} : \mathsf{p} \in \mathcal{P}\}$*.*

We write $|\mathcal{D}|$ to refer to the number of pages belonging to a database. Finally, the following definition describes how triples are distributed onto pages.

Definition 3 (Storage mapping). *Let* $G_1, \ldots, G_n \in \mathcal{G}$ *be RDF graphs and* $\mathcal{D}$ *a database. The triples of the graphs are stored w.r.t. the following criteria:*

(i) All triples t *of a page* p *belong to the same graph* G_i*:*

$$\forall \mathsf{p} \in \mathcal{D} \forall t_i, t_j \in \mathsf{p} \exists G_i \in \{G_1, \ldots, G_n\} : t_i \in G_i \wedge t_k \in G_i$$

(ii) All resources t^s of a page p *have the same* ***rdf:type*** τ*:*

$$\forall \mathsf{p} \in \mathcal{D} \forall t_i, t_j \in \mathsf{p} : \tau(t_i^s) = \tau(t_k^s)$$

(iii) All triples t having the same subject t^s are stored on the same page p*:*

$$\forall G_i \in \{G_1, \dots, G_n\} \forall t_i, t_k \in G_i : t_i^s = t_k^s \wedge t_i \in \mathsf{p}_1 \wedge t_k \in \mathsf{p}_2 \Rightarrow \mathsf{p}_1 = \mathsf{p}_2$$

(iv) URIs and literals are normalized using bijective mapping $\iota : \mathbb{I} \cup \mathbb{L} \rightarrow \mathbb{N}$.

Figure 2 illustrates the mapping of an RDF graph to database pages. After normalizing URIs and literals triples are stored on different pages according to the type of their subject, e.g., page 2 contains only triples of proceedings. Thus, pages are implicitly linked by these, e.g., `ex:pers1` links the two pages.

Normalization

ID	Value
1	`ex:proc1`
2	`ex:proc2`
3	`ex:pers1`
4	`ex:pers2`
5	`foaf:person`
6	`"John Due"`
7	...

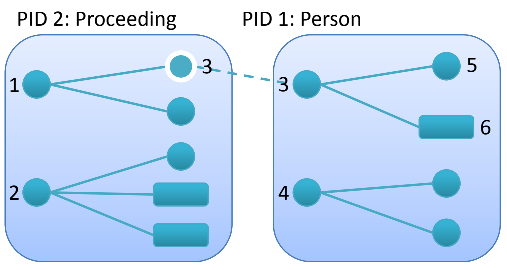


Fig. 2. Mapping of an RDF graph to database pages (properties are omitted)

The internal structure of a page is schematically shown on the left side of Figure 3. It is divided into head and body. The head contains metadata such as the page ID as well as the types and the number of stored subject resources[3]. The body contains the actual triples, however, split into their components. A triple index is used to manage the relationship between subjects and their corresponding predicates and objects (right side of Figure 3). For example, the first entry of the triple index contains the address of the first predicate/object of the first subject, the second entry for the first predicate/object of the second subject, and so on. The advantages of this column-store-like layout are that it eases the implementation (e.g., mapping a page to the domain model in a programming language), subjects are not repeated, and the system can easily access all components of the triples and iterate over them.

Besides the mapping of triples to pages we need also access structures allowing an efficient localization of triples in the database. For this reason we maintain indexes on subjects, predicates, and objects providing the addresses of triples. In the case of indexing subjects it is sufficient to know a mapping from a subject ID to the corresponding page because all triples of a given subject can only occur on a single page. In the case of predicates and objects we use bitset indexes that encode the address of matching triples (cf. Figure 4) – one bitset for each distinct property and object.

[3] A subject may have multiple types but all subjects on a page must have the same.

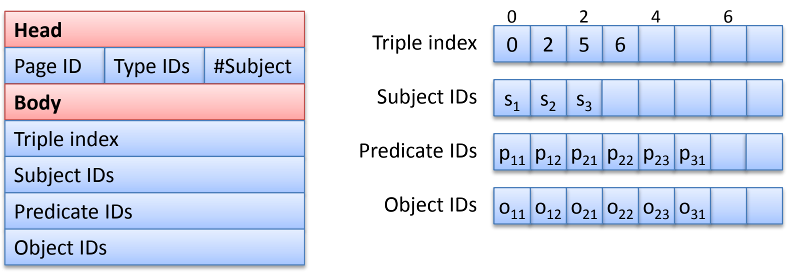

Head
Page ID
Type IDs
#Subject
Body
Triple index
Subject IDs
Predicate IDs
Object IDs
0 2 4 6
Triple index 0 2 5 6
Subject IDs s_1 s_2 s_3
Predicate IDs p_{11} p_{12} p_{21} p_{22} p_{23} p_{31}
Object IDs o_{11} o_{12} o_{21} o_{22} o_{23} o_{31}

Fig. 3. Internal structure of a page: Schematic representation (left) and detail view of the body (right)

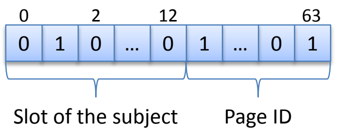


Fig. 4. Interpreting a position within a bitset

Bitset indexes have the advantage that the system can answer multidimensional point queries efficiently by combining them (logical AND or OR). Thus, it is possible locate efficiently triples having a certain combination of predicate and object or subjects having certain objects as property values. We use the algorithms of [29] to process bitsets in a compressed form.

For providing a fulltext search on literals the system has to maintain a dedicated index (e.g., a Lucene index) because bitsets are not very useful in this case. Since text retrieval is well researched we do not consider it in this paper.

5 Operations

In this section we describe the most important operations on the previously defined storage model: querying the repository as well as adding and removing triples. Executing update operations the system has to ensure the consistency of the database according to Definition 3.

Query processing. Regarding query processing we only consider basic graph patterns in this paper because they form the fundamental building block of SPARQL queries [22]. First of all we want to note that any query can be decomposed into joins of star-like patterns – a special form is a single triple pattern. We do not go into details on join algorithms at the moment because they are well-known from relational databases.

To compute the variable bindings for a star-like basic graph pattern (BGP) we distinguish two cases: (1) patterns with a given subject and (2) patterns with a variable as subject. For queries of the first type (lines 4–6, Algorithm 1) the query engine uses the subject index to determine the page ID storing the triples with the given subject. Then, it loads this page from secondary storage and generates the variable bindings by matching the BGP and the stored triples. In the second case (lines 8–22) the query engine loads all bitsets corresponding to the constant predicates and objects of the query and computes their logical AND. Based on the resulting bitset the query engine can determine the pages which may contain relevant triples (line 19). It loads these pages one after the other and scans them for matching subgraphs.

Algorithm 1 matchStarBGP(BGP bgp) : VariableBindings

```
 1: bindings = $\emptyset$
 2: tp = bgp.first {first triple pattern}
 3: if (!isVariable(tp.subject)) then
 4:    address = subjectIdx.get(tp.subject)
 5:    page = load(address.pageId)
 6:    bindings = match(page, address.slot, tp)
 7: else
 8:    bitset = $\sim 0$ {bitset with all bits set}
 9:    for (TriplePattern tp : bgp) do
10:       if (!isVariable(tp.predicate)) then
11:          bitset &= predicateIdx.get(tp.predicate)
12:       end if
13:       if (!isVariable(tp.object)) then
14:          bitset &= objectIdx.get(tp.object)
15:       end if
16:    end for
17:
18:    pageIds = subjectIdx.get(bitset) {determine the page IDs of subjects}
19:    for (pid : pageIds) do
20:       page = load(pid)
21:       bindings += match(page, tp)
22:    end for
23: end if
24: return bindings
```

Algorithm 1 realizes only a naive approach to determine relevant pages. A real query engine would also consider statistical data about resources. The query engine can furthermore use bitset indexes to support evaluating filter expressions at a very early stage, e.g., before loading any data from secondary storage.

Adding triples. Algorithm 2 gives the pseudocode for adding triples. Its input is a normalized triple constructed by applying the mapping function to the triple. Next, the system checks if the predicate of the new triple is `rdf:type` because this information is required to locate an appropriate page for storing it. If the triple does not contain type information then the type `rdfs:Resource` is assumed as defined in [6] (line 1–4). Depending on the existences of triples having the same subject as the new triple the system has either to load the corresponding page (line 7) or to locate a page with an appropriate type using the predicate index (line 9) – if such a page does not exist then a new one is allocated. The new triple is then added to the page. Lines 13 and 15 handle two special cases: Balancing the pages and restoring the storage constraints. In the first case the system has to ensure that there is enough free space on the page to store the triple because a page can only store a limited number of triples. In the event of an overflow it allocates a new page and distributes the triples as equally as possible over the two pages. Hereby, it always guarantees the constraints of Definition 3. In the second case the type constraint is violated, e.g., the new triple states a type of the subject different from the types of the page. The new type of the subject resource is determined as follows: $\tau(t^s) = t^o \cup \tau(\mathsf{p})$. This type is used to find a new page for storing all triples with the given subject. Finally, all indexes have to be updated. Adding a completely new triple this step requires only updating three index entries. In the worst case – pages need to be balanced – the system has to update all index entries belonging to moved triples.

Algorithm 2 insertTriple(NormalizedTriple t) : void

```
1: type = rdf:resource
2: if (t.predicate = rdf:type) then
3:    type = t.object
4: end if
5: pageId = subjectIdx.get(t.subject)
6: if (pageId) then
7:    page = load(pageId)
8: else
9:    page = findOrAllocatePage(type)
10: end if
11: addTriple(page, t)
12: if (type $\in \tau$(page)) then
13:    balancePage(page)
14: else
15:    restoreTypeContraint(page, t) {pages are also balanced}
16: end if
17: updateIndexes(t)
```

Deleting triples. As illustrated in Algorithm 3 the system uses first the subject index to locate the page containing the subject of the triple to be removed. The corresponding page is then loaded from secondary storage and the triple is deleted from the page (lines 3 and 4). Deleting triples can lead to underfull pages. To avoid loading many almost empty pages during query processing the system balances pages and ensures a minimal allocation. Furthermore, the predicate of the triple may be `rdf:type`. In this case the system has to restore the consistency of the repository (line 6). Finally, it has also to update the indexes – the best and worst cases are similar to adding a triple.

6 Analysis

In this section we first estimate the disc space needed to store RDF data. We look then at the complexity of the operations described in the previous section.

Algorithm 3 deleteTriple(NormalizedTriple t) : void

```
1: pageId = subjectIdx.get(t.subject)
2: if (pageId) then
3:     page = load(pageId)
4:     deleteTriple(page, t)
5:     if (t.predicate = rdf:type) then
6:         restoreTypeContraint(page, t) {pages are also balanced}
7:     else
8:         balancePage(page) {removes empty pages}
9:     end if
10:    updateIndexes(t)
11: end if
```

6.1 Data complexity

We measure data complexity of our storage model in terms of allocated database pages. To estimate the number of pages needed for storing an RDF graph we first have to know how many triples can be stored onto a page. In the following we consider only the size of the body of a page because the size of the head is almost constant and negligible small in relation to the size of the body. Thus, the number of triples that fit onto a page depends on the page size $\|\mathsf{p}\|$ and the required space for triples $m\|t^s\| + n(\|t^p\| + \|t^o\|)$ with m being the number of subjects and n being the number of triples. Since subject IDs are not repeated we have a minimal (formula 1) and a maximal (formula 2) number of triples per page, e.g., all triples have a distinct or the same subject, respectively. If the space of a page is fully allocated then we can calculate these values as follows:

$$\|\mathsf{p}\| = m\|t^s\| + n(\|t^p\| + \|t^o\|)$$

$$\overset{m=n}{\Longrightarrow} \quad n_{min} = \frac{\|\mathsf{p}\|}{\|t^s\| + \|t^p\| + \|t^o\|} \tag{1}$$

$$\overset{m=1}{\Longrightarrow} \quad n_{max} = \frac{\|\mathsf{p}\| - \|t^s\|}{\|t^p\| + \|t^o\|}$$

$$\overset{\|\mathsf{p}\| \gg \|t^s\|}{\Longrightarrow} \quad n_{max} \approx \frac{\|\mathsf{p}\|}{\|t^p\| + \|t^o\|} \tag{2}$$

In Table 2 we compiled the approximate minimal and maximal numbers of triples per page for typical page sizes. In this calculation the sizes of an ID and an entry in the triple index are 8 and 2 bytes, respectively.

Given the above formula we can derive formula for the minimal and maximal number of pages required to store an RDF graph G (formula 3 and 4, respectively). However, there is a lower bound (formula 5) for even small graphs that is induced by Definition 3; the system has to allocate at least one page for each type of resource (τ_G).

$\|\mathsf{p}\|$	8 kB	16 kB	32 kB	64 kB
$\sim n_{min}$	300	600	1,200	2,400
$\sim n_{max}$	500	1,000	2,000	4,000

Table 2. Number of triples per page

$$|\mathcal{D}|_{min} = \sum_{k \in \tau_G} \frac{|\{t \in G : \tau(t^s) = k\}|}{n_{max}} \tag{3}$$

$$|\mathcal{D}|_{max} = \sum_{k \in \tau_G} \frac{|\{t \in G : \tau(t^s) = k\}|}{n_{min}} \tag{4}$$

$$|\mathcal{D}|_{lb} = |\tau_G| \tag{5}$$

Besides the space required to store the triples the indexes for subjects, properties, and objects occupy disc space as well. As already mentioned the subject index contains only a mapping of subject IDs to page IDs; thus, a B-tree can be used. Based on [3] the data complexity is $O(N)$, N being the number of indexed values. According to [4] DBpedia contains for example $2,8 \cdot 10^6$ resources. Assuming a page size of 16 kB and a pointer size of 8 bytes the subject index would require about 2,740 pages and occupy 42 MB of disc space. Since the nodes of a B-tree have typically a large number of entries, the system can hold at least the first two levels of a B-tree in main memory.

The predicate and object indexes in contrast are based on bitset indexes. Our estimations for the required disc space are based on the WAH compressed bitset [29]. WAH bitsets are organized in words of 32 or 64 bits and are compressed word by word. The author estimates the size of a bitset as $2N$ in the worst case, e.g., all bits are set. To put the size of a bitset into relationship with B-trees, Wu et al. [29] mentions that a B-tree occupies $3N \sim 4N$ words. In the context of our storage model N equals the number of managed triples. Due to the characteristics of RDF data we can assume that the worst case will never occur and the sizes of bitsets are much smaller. We can use the following formula to calculate the number of pages occupied by an index:

$$|\mathsf{p}_{Bitset}| = \frac{\|w\| \cdot 2N}{\|\mathsf{p}\|},\ \|w\| = \text{word size}$$

Further compression strategies are applicable, e.g., K-of-N encoding, but they would require decompression during query evaluation and, thus, increase the overall processing time most likely.

In the proposed storage model we need a bitset for each distinct property and object. Therefore, the system has to manage $O(|P|N + |O|N)$ bitsets, where $|P|$ and $|O|$ are the number of properties and objects, respectively. We use a B-tree to locate the bitset corresponding to a resource. Please note, Lemire et al. assess in [16] that a system can efficiently manage even 100 million bitsets.

As an example, we determined the number of distinct properties and objects of DBpedia (Version 3.4). It contains about $44 \cdot 10^6$ triples, about 52.500 distinct properties, and $9.8 \cdot 10^6$ distinct objects ($5.9 \cdot 10^6$ literals and $3.8 \cdot 10^6$ Resources). Since the required space depends on the number of set bits, we also determined the number of triples having a given property or object.

Only a few properties occurred in more that a thousand triples (cf. Table 3) which is a very acceptable fraction w.r.t. bitsets. Assuming the worst case for

WAH bitsets – each set bit has to represented with 2 bytes – then a bitset with 1000 k set bits would require 250 pages and occupy 8 MB (page size 32 k). In general, however, the system can access almost all bitsets (about 98.2 %) with a single load operation.

cardinality	$> 1000k$	$500k$	$> 100k$	$> 50k$	$> 10k$	> 5000
absolute	3	3	59	65	449	379
percentage	0.006 %	0.006 %	0.112 %	0.129 %	0.854 %	0.721 %
cardinality	> 1000	> 500	> 100	> 50	> 10	≤ 10
absolute	1688	1212	6560	4361	9098	32546
percentage	3.209 %	2.304 %	8.292 %	5.191 %	17.298 %	61.879 %

Table 3. Number of triples per property in DBpedia

Analyzing DBpedia w.r.t. objects we only considered resources because literals are separately stored in a fulltext index (cf. Table 4). In contrast to the above numbers there are even fewer triples per object. The largest index has a size of 0.8 MB and requires only 25 pages (page size 32 k). Thus, the system can load actually all bitsets (about 99.9 %) accessing the disc once.

cardinality		$500k$	$> 100k$	$> 50k$	$> 10k$	> 5000
absolute		0	10	12	105	125
percentage		0.000 %	$3 \cdot 10^{-4}$ %	$3 \cdot 10^{-4}$ %	0.003 %	0.003 %
cardinality	> 1000	> 500	> 100	> 50	> 10	≤ 10
absolute	991	1228	10057	12010	93049	$3.8 \cdot 10^6$
percentage	0.026 %	0.032 %	0.261 %	0.311 %	2.412 %	96.953 %

Table 4. Number of triples per object in DBpedia

6.2 Complexity of operations

In the following we analyze the complexity of the operations defined in Section 5 which we measure in number of pages loaded from and written to secondary storage, including data pages as well as index pages. Before going into details of operations, we want to note that the complexity of accessing B-trees is $O(\log_b N)$ where b is the fanout and N is the number of indexed values [3] because the system has often to access B-trees to locate data pages.

Query processing. In the following we analyze only the complexity of Algorithm 1 which constructs variable bindings for star-like basic graph patterns. Since arbitrary basic graph patterns are decomposed into star-like pattern we have to add the complexity of joining the variable bindings for arbitrary basic graph pattern.

The complexity of Algorithm 1 depends on if the subject of the pattern is given. If it is known then the query engine has only to access the subject index (B-tree) to find the corresponding data page. Independently of the pattern size, this page contains all triples to answer the query. Thus, the complexity is $O(\log_b N)$.

In the case that the subject is not given the complexity is made up of (1) identifying the data pages to be loaded and (2) loading these pages to match the pattern. To identify the relevant pages the query engine combines the bitsets of the constant predicates and objects, e.g., a pages can only be relevant if it contains all resources contained in the BGP. This step requires to locate the relevant bitsets and to load them. In the worst case the complexity is as follows:

$$O(\underbrace{(|t_Q^p| + |t_Q^o|)\log_b N}_{\#\text{accesses B-tree}} + \underbrace{(|t_Q^p| + |t_Q^o|)2N}_{\#\text{accesses bitsets}})$$

where $|t_Q^p|$ and $|t_Q^o|$ are the numbers of constant predicates and objects, respectively. The $2N$ in the second part is the complexity of accessing the bitsets. As illustrated in the previous section these values are almost always very small, e.g., a page per bitset in the worst case. Thus, the complexity of step (1) depends only on the number of constants of the BGP but not on the size of the database. The complexity of step (2) is determined by the size of the solution $O(|S|)$, e.g., in the worst case the query engine has to access a page per triple. However, since we cluster the triples according to the type of the subject the worst case will occur rarely. As a result the complexity of both steps is

$$O((|t_Q^p| + |t_Q^o|)\log_b N + (|t_Q^p| + |t_Q^o|)2N + |S|)$$

Adding and deleting triples Considering both operations only the following operations are relevant for determining their complexity: (1) Locating and loading of the affected page and writing it back, (2) updating the subject index, and (3) updating the predicate and object indexes. Regarding the first two steps the complexity is dominated by the operations on B-tree of the subject index ($O(\log_b N)$ because the system needs to read and to write only a few pages, e.g., one page in the best case and two pages if balancing is required. In the worst case of a type conflict the system has to find an appropriate page which requires accessing the bitset of `rdf:type` and the one of the type resources. The complexity is comparable to the one of a query. Considering the second step the system has update two bitsets in the best case resulting in $\frac{2 \cdot 2N}{\|\mathsf{p}\|} + \frac{2 \cdot 2N}{\|\mathsf{p}\|}$ read and write operations – all bits being set. In the worst case when balancing is required then the system has to access more pages. Their number depends on the number of distinct predicates objects of the moved triples.

7 Evaluation

We implemented the proposed storage model in Java and run some preliminary test to evaluate our approach. As illustrated in Figure 5 the key components of our system are based on existing software. At the lowest level we use a Berkeley DB for storing database pages, the subject index, as well as the bitsets of the predicate and object indexes. The main reasons for a Berkeley DB are that it can efficiently manage key value pairs – values can be arbitrary objects – and we wanted to focus on implementing our storage model.

On top of the Berkeley DB we implemented a page manager and an index manager. The page manager is responsible for mapping page IDs (key) to pages (value) and to convert them to Java objects. The index manager maintains the subject, predicate, and object indexes. In case of the subject index the key is a resource ID and the value a page ID; in case of the others the key is a resource ID and the value is a bitset.

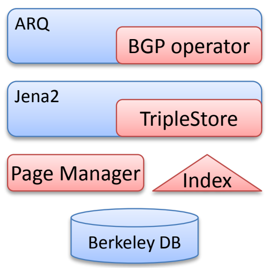


Fig. 5. Architecture of the RDF repository

To reduce the effort for implementing components of an RDF repository such as the RDF data model and SPARQL query engine we adapted Jena2 and ARQ [26], a SPARQL processor for Jena. Inside Jena we created the infrastructure for reading and writing RDF graphs using the presented storage model, e.g., a `TripleStore` based on normalized triples. In the query processor ARQ we had to replace the implementation of filtered BGPs because the original classes could not take advantage of the new storage model. Thus, we implemented an operator that recognizes star-like BGPs and evaluates them separately. If the query contains several star-like pattern the join is currenty performed by ARQ.

We run some preliminary tests to validate the feasibility of our storage model. Table 6 presents the processing time of evaluating star-like BGPs of different sizes and varying result sizes; the subject was always a variable. The underlying dataset was generated using BSBM [5] and containing 300 k triples. As a reference we also present the processing times of the same queries using Jena SDB [26] and RDF-3X [20]. The first one we chose because our system is based on Jena + ARQ and RDF-3X because it is one of the most performant RDF repositories. Looking at the result we feel confident that our storage model is competitive. While the processing time of the other two systems increases with larger BGPs the one of our system remains almost constant. The last query shows that retrieving larger result sets increases the processing time significantly in our system. Our explanation for this behavior is that the data distribution is not favorable, e.g., a clustering of resources based on `rdf:type` is not sufficient.

Query	$\lvert Q\rvert$	$\lvert S\rvert$	SDB	RDF-3X	ours
Q_1	2	1,000	415	30	110
Q_2	4	1,000	849	45	91
Q_3	8	1,000	1577	63	151
Q_4	11	402	4,440	138	104
Q_5	5	20,000	11,000	846	2,042

Fig. 6. Processing time (ms) of BGPs of different sizes and selectivities

Query	$\lvert Q\rvert$	$\lvert$Filter$\rvert$	σ	TDB	ours
Q_1	3	0		11,850	11,800
Q_2	7	0		19,575	25,590
Q_3	3	1	2.5 %	4,057	195
Q_4	7	1	2.5 %	4,440	452
Q_5	3	9	12 %	16,174	1,151
Q_5	7	9	12 %	17,253	2,751

Fig. 7. Processing time (ms) of filtered BGPs

Furthermore, we evaluated the processing of filtered BGPs to validate the feasibility of using bitset indexes for predicates and objects. Since the indexes store only URIs and bitsets are best suited for testing on equality the queries consisted of a star-like pattern with and a filter expression containing zero, one, or nine predicates. For example, such filters are used to restrict resources to certain types. We constructed the filter expressions manually so that they selected about 2.5 % or 12 % of the dataset (3.5 million triples). Table 7 gives the results for these queries – in this test set we compared our system with Jena TDB only. Comparing the processing times of the queries without a filter (rows 1 and 3) and the one with a filter we are confident that the query engine was able to reduce the number of candidate pages significantly.

8 Conclusion and future work

In this paper we presented a new approach for managing RDF data. Our storage model is based on the assumption that a query will often involve several properties of a resource. Thus, we store all triples having the same subject on the same database page. To ensure an efficient retrieval of triples we index them using bitsets. We implemented our storage model using Jena and ARQ as framework. Although we run only preliminary test – not being exhaustive – they indicate that the presented model has advantages in computing the results of star-like BGPs. The system can also process filter expression testing on equality efficiently.

However, our experiments also showed us that we are still open issues which we address in our future work. For example, joins between two BGPs are currently computed by ARQ; its join algorithm performs very poorly. Another important topic is reducing the load time significantly. Besides these two topics we also work on indexing graph pattern which could span over several star-like pattern. In combination with our storage model the key advantage is that only little information is needed even to index complex pattern, e.g., an index entry would only consist of addresses of subject resources.

References

1. Jena TDB, August 2011. http://openjena.org/TDB/.
2. S. Alexaki, V. Christophides, G. Karvounarakis, D. Plexousakis, and K. Tolle. The RDFSuite: Managing Voluminous RDF Description Bases. In S. Decker et al., editors, *2nd International Workshop on the Semantic Web*, pages 1–13, May 2001.
3. R. Bayer and E. McCreight. Organization and maintenance of large ordered indices. In *Proceedings of the 1970 ACM SIGFIDET, Workshop on Data Description, Access and Control*, pages 107–141, New York, NY, USA, 1970. ACM.
4. C. Bizer, J. Lehmann, G. Kobilarov, S. Auer, C. Becker, R. Cyganiak, and S. Hellmann. DBpedia - a crystallization point for the web of data. *Web Semantics*, 7(3):154–165, 2009.
5. C. Bizer and A. Schultz. The berlin sparql benchmark. *Int. J. Semantic Web Inf. Syst.*, 5(2):1–24, 2009.
6. D. Brickley and R. Guha. RDF Vocabulary Description Language: RDF Schema. http://www.w3.org/TR/rdf-schema/, February 2004. W3C Recommendation.
7. J. Broekstra, A. Kampman, and F. van Harmelen. Sesame: A generic architecture for storing and querying RDF and RDF schema. In I. Horrocks and J. A. Hendler, editors, *ISWC*, volume 2342 of *LNCS*. Springer, June 2002.

8. E. I. Chong, S. Das, G. Eadon, and J. Srinivasan. An efficient SQL-based RDF querying scheme. In K. Böhm et al., editors, *Proceedings of VLDB*, pages 1216–1227. ACM, September 2005.
9. I. F. Cruz, V. Kashyap, S. Decker, and R. Eckstein, editors. *Proceedings of the First International Workshop on Semantic Web and Databases*, September 2003.
10. Franz Inc. Allegrograph rdfstore^{TM}, 2010. Zuletzt besucht am 16.3.2010.
11. S. Harris. SPARQL query processing with conventional relational database systems. In M. Dean et al., editors, *Scalable Semantic Web Knowledge Base Systems*, volume 3807 of *LNCS*, pages 235–244, November 2005.
12. A. Harth and S. Decker. Optimized index structures for querying rdf from the web. In *Proceedings of the 3rd Latin American Web Congress*. IEEE Press, October 2005.
13. A. Harth, J. Umbrich, A. Hogan, and S. Decker. Yars2: A federated repository for querying graph structured data from the web. In K. Aberer et al., editors, *ISWC/ASWC*, volume 4825 of *LNCS*, pages 211–224. Springer, November 2007.
14. P. Hayes. RDF Semantics, February 2004. W3C Recommendation.
15. G. Klyne and J. J. Carroll. RDF: Concepts and Abstract Syntax. http://www.w3.org/TR/rdf-concepts/, 2004. W3C Recommendation.
16. D. Lemire, O. Kaser, and K. Aouiche. Sorting improves word-aligned bitmap indexes. *Data Knowl. Eng.*, 69(1):3–28, 2010.
17. B. Liu and B. Hu. HPRD: A High Performance RDF Database. In K. Li et al., editors, *Int. Conference Network and Parallel Computing*, volume 4672 of *LNCS*, pages 364–374. Springer, September 2007.
18. L. Ma, Z. Su, Y. Pan, L. Zhang, and T. Liu. RStar: an RDF storage and query system for enterprise resource management. In *Int. Conference on Information and Knowledge Management*, pages 484–491, New York, NY, USA, 2004. ACM Press.
19. A. Matono, T. Amagasa, M. Yoshikawa, and S. Uemura. A path-based relational RDF database. In *Australasian conference on database technologies*, pages 95–103, Darlinghurst, Australia, Australia, 2005. Australian Computer Society, Inc.
20. T. Neumann and G. Weikum. The RDF-3X engine for scalable management of RDF data. *The VLDB Journal*, 19(1):91–113, 2010.
21. OpenLink Software. Virtuoso, 2010. Zuletzt besucht am 16.3.2010.
22. E. Prud'hommeaux and A. Seaborne. SPARQL Query Language for RDF, Januar 2008. W3C Recommendation.
23. M. Schmidt, T. Hornung, G. Lausen, and C. Pinkel. *sp*^{2}bench: A sparql performance benchmark. In *ICDE*, pages 222–233. IEEE, 2009.
24. C. Weiss, P. Karras, and A. Bernstein. Hexastore: sextuple indexing for semantic web data management. *Proc. of the VLDB Endowment*, 1(1):1008–1019, 2008.
25. T. Weithöner, T. Liebig, and G. Specht. Storing and Querying Ontologies in Logic Databases. In Cruz et al. [9].
26. K. Wilkinson, C. Sayers, H. Kuno, and D. Reynolds. Efficient RDF Storage and Retrieval in Jena2. In Cruz et al. [9].
27. D. Wood, P. Gearon, and T. Adams. Kowari: a platform for semantic web storage and analysis. In *Proceedings of XTech 2005*, 2005.
28. G. Wu, J. Li, J. Hu, and K. Wang. System *Pi*: A Native RDF Repository Based on the Hypergraph Representation for RDF Data Model. *Journal of Computer Science and Technology*, 24(4):652–664, July 2009.
29. K. Wu, E. J. Otoo, and A. Shoshani. Optimizing bitmap indices with efficient compression. *ACM Trans. Database Syst.*, 31(1):1–38, 2006.

Author Index

A
Alexander, Paul R. 14
C
Collier, Suzanne 1
Cure, Olivier 122
D
Dean, Mike 1
E
Emmons, Ian 1
F
Fokoue, Achille 106
G
Garlapati, Mounika 1
H
Harth, Andreas 30
Heese, Ralf 138
Heflin, Jeff 90
L
Ladwig, Günter 30
Lamolle, Myriam 122
Le Duc, Chan 122
Li, Yingjie 90
M
Musen, Mark A. 14
Möller, Ralf 75
N
Noy, Natalya F. 14
P
Pan, Jeff Z. 106
R
Risch, Tore 43
S
Salvadores, Manuel 14
Sensoy, Murat 106
Srivatsa, Mudhakar 106
Stamou, Giorgos 59
Stefanova, Silvia 43
Stoilos, Giorgos 59
V
Venetis, Tassos 59
W
Wandelt, Sebastian 75

Z
Znamirowski, Martin 138