Joint Workshop on Scalable and High-Performance Semantic Web Systems (SSWS + HPCSW 2012)

At the 11th International Semantic Web Conference (ISWC2012), Boston, USA, November, 2012

SSWS + HPCSW 2012 PC Co-chairs' Message

For 2012, the 8th International Workshop on Scalable Semantic Web Knowledge Base Systems (SSWS2012) and the 2nd Workshop on High-Performance Computing for the Semantic Web (HPCSW2012) were merged together. This joint workshop focused on addressing broader 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 application's 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 11 submissions. Each paper was carefully evaluated by three workshop Program Committee members. Based on these reviews, we accepted seven papers for presentation. We sincerely thank the authors for all the submissions and are grateful for the excellent work by the Program Committee members.

November 2012

Achille Fokoue
Thorsten Liebig
Eric Goodman
Jesse Weaver
Jacopo Urbani
David Mizell

Program Committee

Jans Aasman
Franz, Inc.

Robert Adolf
Pacific Northwest Nat. Lab., USA

Sinan Al-Saffar
Pacific Northwest Nat. Lab., USA

Alexey Cheptsov
High Performance Computing Center Stgt, Germany

Oscar Corcho
Univ. Politecnica de Madrid, Spain

Mike Dean
BBN Technologies, USA

Achille Fokoue
IBM Watson Research Center, USA

Raúl García-Castro
Univ. Politecnica de Madrid, Spain

Eric Goodman
Sandia National Laboratories, USA

Yuanbo Guo
Microsoft, USA

Volker Haarslev
Condordia University, Canada

David Haglin
Pacific Northwest Nat. Lab., USA

Pascal Hitzler
Wright State University, Ohio, USA

Aidan Hogan
DERI Galway, Ireland

Bill Howe
University of Washington, USA

Cliff Joslyn
Pacific Northwest Nat. Lab., USA

Anastasios Kementsietsidis
IBM Watson Research Center, USA

Pavel Klinov
Ulm University, Germany

Spyros Kotoulas
IBM Watson Research Center, USA

Thorsten Liebig
derivo GmbH, Germany

David Mizell
YarcData, Inc, USA

Ralf Möller
Hamburg Univ. of Techn., Germany

Jeff Z. Pan
University of Aberdeen, UK

Axel Polleres
Siemens AG, Österreich

Mariano Rodriguez
Free University of Bolzano, Italy

Sebastian Rudolph
Karlsruhe Inst. of Techn., Germany

Andy Seaborne
Epimorphics, UK

Kavitha Srinivas
IBM Watson Research Center, USA

Jacopo Urbani
Vrije Universiteit Amsterdam, Netherlands

Jesse Weaver
Rensselaer Polytechnic Institute, USA

Gregory Todd Williams
Rensselaer Polytechnic Institute, USA

Takahira Yamaguchi
Keio University, Japan

Additional Reviewers

Cong Wang
Wright State University, Ohio, USA

Kevin Lee
University of Aberdeen, UK

Table of Contents

FishMark: A Linked Data Application Benchmark

Samantha Bail[1], Sandra Alkiviadous[1], Bijan Parsia[1], David Workman[2], Mark Van Harmelen[2], Rafael S. Goncalves[2], and Cristina Garilao[3]

[1] The University of Manchester, Manchester M13 9PL, United Kingdom
[2] HedTek, Manchester M4 1LZ, United Kingdom
[3] GEOMAR Helmholtz-Zentrum fr Ozeanforschung, 24148 Kiel, Germany

Abstract. FishBase is an important species data collection produced by the FishBase Information and Research Group Inc (FIN), a not-for-profit NGO with the aim of collecting comprehensive information (from the taxonomic to the ecological) about all the world's finned fish species. FishBase is exposed as a MySQL backed website (supporting a range of canned, although complex queries) and serves over 33 million hits per month. FishDelish is a transformation of FishBase into LinkedData weighing in at 1.38 billion triples. We have ported a substantial number of FishBase SQL queries to FishDelish SPARQL query which form the basis of a new linked data application benchmark (using our derivative of the Berlin SPARQL Benchmark harness). We use this benchmarking framework to compare the performance of the native MySQL application, the Virtuoso RDF triple store, and the Quest OBDA system on a fishbase.org like application.

1 Introduction

The Linked Open Data (LOD) movement promises much, indeed, nothing less than a new World Wide Web with comparable success to the Web as it is. The amount of LOD is growing at an interesting pace and the underlying technologies are constantly improving. Off the shelf, untuned RDF triple stores handle data sets on normal hardware that would have been unthinkable 5 years ago. However, there is scant evidence about the benefits and drawbacks of converting applications to use linked data. Given that the conversion from "native" data models (such as XML or relational databases) typically involves a large blow up in size and loss of tuned structures, e.g. indexes or (de)normalization, achieving comparable performance post-triplification is a common concern. While there may be other benefits to triplification, such as easier integration with other LOD, this needs to be weighed against the costs imposed.

To help assess the costs of triplification, we have developed FishMark, an application benchmark for linked data systems. FishMark consists of two components: The Manchester University Multi-Benchmarking (MUM-benchmark) framework, and a set of data, queries, and query frequences derived from FishBase, a comprehensive database about the world's finned fish species, and fish-

base.org, a popular web front end to FishBase.[4] FishBase is a LAMP application with a large number of SQL queries being invoked against a MySQL backend to generate various web pages. We have triplified FishBase and ported its most popular queries to SPARQL. FishMark thus allows a precise comparison between the RDBMS infrastructure and various linked data competitors.

Further, we have created an OWL ontology and respective mappings for the FishBase data, which allows us to measure the performance of an ontology-based data access (OBDA) system [6,4] compared to the RDMBS and RDF triple store versions. OBDA systems offer a different approach to the problem of describing domain knowledge at an abstract level while granting efficient and scalable access to large amounts of data. In the OBDA approach the data are held in external data sources, such as a standard relational database, which are connected to information in an OWL ontology using *mappings*. Ontologies used for this purpose are in the OWL 2 QL profile[5] which is based on the *DL-Lite* [1] family of description logics.

In this paper, we present a first approach to comparing the query performance of native RDBMS, RDF triple stores, and OBDA systems using a single set of—real—data and queries. In particular, we compare an Extract Transform Load (ETL) approach using D2R and the Virtuoso RDF store, and Quest, an OBDA system which executes the SPARQL queries via mappings against the original MySQL database.

2 Related Work

There have been a number of benchmarks for measuring the performance of SPARQL query answering, and countless approaches to SQL benchmarking; we therefore focus on a selection of the most prevalent RDF benchmarking frameworks.

One of the most well-known performance benchmarks for RDF stores is the Lehigh University Benchmark (LUBM) [5]. LUBM consists of a small, hand-built ontology containing information about university departments, students, etc., with a large number of instances for each class. The dataset is scaled by increasing the number of universities, which creates a randomised number of instances for the new university. This method generates entirely disjoint sets of data, a problem which the University Ontology Benchmark (UOBM) [7] seeks to rectify by generating *interrelations* between the instances across different universities.

The LUBM Benchmark is used in [11] as a benchmark for SparqlEngineDB, a SPARQL-to-SQL system which translates SPARQL queries to SQL queries and executes them against a relational database. This translation approach is thought to have an advantage over querying native RDF stores with SPARQL, as it does not require to hold the entire data in memory (as is the case with SPARQL), while also making use of the query optimisation techniques used in RDBMS.

[4] In general, we will use "FishBase" to refer to both parts.
[5] http://www.w3.org/TR/2008/WD-owl2-profiles-20081008/

The authors perform an evaluation of SparqlEngineDB against Virtuoso and Jena-SDB with the data scaling up to 200,000 triples. They find that Virtuoso exceeds the performance of both SparqlEngineDB and Jena-SDB.

The Berlin SPARQL Benchmark (BSBM) [2] uses a hand-built e-commerce dataset and a fixed set of queries to measure the performance of RDF triple stores. It provides a data generation tool which generates a data set of custom size, as well as a test driver framework for performing query mixes for several use cases, such as "explore", "update and explore", and "Business Intelligence", which emulate the *"search and navigation pattern of a consumer looking for a product"* [2]. The authors present the results of applying BSBM to measure the SPARQL query performance of several RDF triple stores, as well as the SQL translations of the SPARQL queries using Virtuoso RDF View and D2R Server.[6] While Sesame performs best for the smallest instance of the dataset (1 million items), Virtuoso's RDF View outperforms the triple stores on large-scale datasets. Even though no direct comparisons between the RDB and RDF stores' performance are made, it can be seen that the native SQL queries outperform the SPARQL queries by an order of magnitude on the smallest dataset.

One of the RDF benchmarks that use real test data and queries is the DBpedia SPARQL Benchmark (DBPSB) [8]. DBPSB uses data from the DBpedia[7] knowledge base and queries extracted from DBpedia's query logs. While implementing standard benchmarking techniques such as clearing the cache and performing warm-up runs, DBPSB also provides a method for scaling the size of the benchmarking dataset. In [8] the authors use DBPSB to test the performance of four triple stores: Virtuoso,[8] Sesame,[9] BigOWLIM,[10] and Jena-TDB.[11] They find that Virtuoso is the fastest of the four triple stores, handling large amounts of data significantly better than the other systems.

A similar approach to BSBM is the Social Network Intelligence Benchmark (SIB) [3], which uses a schema of a social network similar to Facebook, and data dictionaries in order to generate artificial RDF data. The benchmark also includes several query mixes, such as the "Interactive query mix" of 20 queries that simulate typical user interaction in a social network. The SP^2Bench benchmark [10] is set in a scenario similar to DBLP, a large computer science bibliography database which indexes over 200,000 citations. SP^2Bench includes a data generator to generate "large DBLP-like models" in RDF, which is based on a study of the features of DBLP and 12 hand-built SPARQL queries which vary in their characteristics.

[6] http://d2rq.org/d2r-server
[7] http://dbpedia.org/
[8] http://virtuoso.openlinksw.com/
[9] http://www.openrdf.org/index.jsp
[10] http://www.ontotext.com/owlim/editions
[11] http://jena.apache.org/documentation/tdb/index.html

3 Materials and Methods

3.1 Data

There are currently two FishMark data sets: A MySQL database representing a snapshot of FishBase from 2011, and an RDF graph that is the result of applying a D2R conversion to that database.

The conversion of the complete FishBase dump via D2R consumed several hours and resulted in an RDF graph with 1.38 billion triples (which was stored as a 250GB file). Initial tests with various triple stores, however, were unsuccessful when attempting to load the data. In order to deal with this issue, we generated another MySQL dump of FishBase which only included the tables needed for the given queries. This reduced the data resulting from the D2R conversion to approximately 20 million (20,186,776) triples. According to the Virtuoso statistics generator, this data set contains 31,927 fish species.

The FishBase OWL ontology which was manually created using the Protégé 4 ontology editor contains 10 classes, 10 object properties, 84 data properties, and 206 logical axioms. The manually created OBDA model contains 20 mappings, which map data from the RDB to the OWL ontology.

3.2 Queries and Query Mix

While, eventually, the FishMark query set should include at least all queries which drive FishBase.org, currently we have a select set of 22 SQL queries with a range of complexity and corresponding SPARQL translations. Short descriptions of all 22 queries are listed in Table 2 alongside the number of joins for each query; sample instances of a selected set of queries can be found in Appendix B. The queries are of restricted complexity due to the limited number of SPARQL features supported by the Quest OBDA system at the time of performing the benchmark.

We obtained a server log from the fishbase.org server in Kiel, Germany, for June 2012. The logs indicated that, on average, only a small number of distinct queries were performed on FishBase, the most frequent being the generation of a species page for a fish (2440 queries per day), followed by the search for a fish species by common name (1034 queries per day). The total numbers in June 2012, as well as the average daily and hourly frequency of the most frequently used queries (according to the FishBase server logs) are given in Table 3.

We use the following notation to describe aspects of the benchmarking framework:

- Query type: A parameterised named query, e.g. “Query 1: CommonName”, “Query 2: SpeciesPage”, etc.
- Query instance: An instance of a query type with randomly selected values for the parameters.
- Query set: A set of query instances of all query types in the query mix.

We have defined three distinct benchmarks which are intended to test various aspects of the systems:

ID	Query name	Description	Joins
1	CommonName	Find species for a given common name.	2
2	SpeciesPage	Find information about a specific species.	5
3	Genus	Find species matching a given genus.	1
4	Species	Find species matching a given species.	1
5	FamilyInformation	Find information about a family of species.	1
6	FamilyAllFish	Find all fish for a given family.	1
7	FamilyNominalSpecies	Find all nominal species for a given family.	1
8	FamilyListOfPictures	Find all pictures for a given family.	2
9	CollaboratorPage	Retrieve information about a collaborator.	0
10	PicturePage	Retrieve information about a picture.	1
11	CAllFish	Find all fish for a given country.	3
12	CSpeciesInformation	Find information about a species of a country.	2
13	CFreshwater	Find all freshwater species for a country.	3
14	CIntroduced	Find all introduced species for a country.	3
15	CEndemic	Find all endemic species for a country.	3
16	CReefAssociated	Find all reef-associated species for a country.	3
17	CPelagic	Find all pelagic species for a country.	3
18	CGameFish	Find all game fish for a country.	2
19	CCommercial	Find all commercial fish for a country.	4
20	CUsedAquaculture	Find all species used for a. c. for a country.	3
21	CPotAquaculture	Find species w/ potential use for a.c. for a country.	2
22	CAquariumTrade	Find all species used for a. t. for a country.	3

Table 2. FishBase Queries with short descriptions and number of joins.

Benchmark 1: Individual Queries The first query mix for the benchmark consists of a simple performance test for each individual query, i.e. the query mix is used as a simple means to run each query exactly once. We generated multiple query sets for this test, each with a new, randomly selected set of parameters. Generating multiple query sets seemed necessary, as the data in FishBase are fairly heterogeneous: For example, a common name search for "Shark" returns 24 different species, while the same search for "Borna Snakehead" returns exactly 1 result. Therefore, running the query mix with the same query set may skew the results towards "good" or "bad" parameters for the queries. The Berlin SPARQL Benchmark employs the same principle to generate different parameter values for queries.

Benchmark 2: Randomised Weighted Query Mix A second variant of the query mix is the randomised weighted query mix based on the FishBase server access logs. The query mix contains the 5 most frequent query types, each of the queries being instantiated n times, where n is the frequency of the query according to the server access logs. The final query mix contains 175 query instances of 5 query types. Note that the queries are instantiated with random parameter values, therefore it is possible that some of the query instances are identical. This seems realistic, as FishBase users might perform the same query several times in a row.

Query name	Month	Day	Hour
Species Page	73213	2440.43	101.68
Common Name	31008	1033.60	43.07
Genus	13331	444.37	18.52
Country Species Information	4429	147.63	6.15
Collaborator Page	4138	137.93	5.75

Table 3. Frequency of the most common FishBase queries per month (total), day (mean), and hour (mean), June 2012.

Benchmark 3: Typical User Scenario The log files also allow us to draw conclusions as to how users commonly navigate on the fishbase.org site. As the species page for a fish species is the central point of information, the log files show a usage pattern which focuses heavily on accessing species pages from various other points on the site, most frequently the common name search (which is a prominent feature on the fishbase.org start page). From this usage pattern, we can construct a query mix which emulates the route a typical user takes during one visit to fishbase.org, similar to BSBM's *explore* use case.

3.3 Datastores- and Access

Virtuoso Open Source 6.1.5 Virtuoso is a "multi-model data server" which supports data storage and management of relational data, XML data, and RDF data, amongst others. Through several prior benchmarks, Virtuoso emerged as one of the best performing RDF triple stores. We installed Virtuoso (Open Source edition) following the instructions on the Virtuoso wiki[12] for a default install. As recommended by Virtuoso, the following parameters were modified in the virtuoso.ini file to match our hardware setup:

- NumberOfBuffers: 1360000
- MaxDirtyBuffers: 1000000
- MaxCheckpointRemap: 60GB (= 1/4 of the DB size)

These were the only measures taken to tune the datastore. The FishBase RDF triples were then loaded into the database from the n-triples file using the SQL command DB.DBA.TTLP_MT.

MySQL 5.5 Relational DBMS The current live version of FishBase uses a MySQL RDBMS as data store. As described above, we generated a smaller snapshot of the FishBase database dump which contained all the information required by the queries in the query mix. The data was loaded into an "out-of-the-box" install of the MySQL 5.5 RDBMS running on our test machine.

[12] http://www.openlinksw.com/dataspace/dav/wiki/Main

Quest 1.7 OBDA System (using a MySQL database) The Quest OBDA system [9] defines "virtual ABoxes" over data which is stored in a relational DBMS. It provides access to the data via SPARQL queries which the system rewrites into SQL, thereby delegating the query execution entirely to the RDBMS. Quest currently supports several RDBMS, including PostgreSQL, MySQL, DB2 and Oracle. For our benchmark, we used the existing MySQL database with the manually created OWL ontology and mappings described above.

3.4 Benchmarking Framework

We developed a multi-query language benchmarking framework which is a derivative of the Berlin SPARQL Benchmark (BSBM) code, extending its functionality by a query generation module and additional connectors for SQL queries and OBDA access. The Manchester University Multi-Benchmarking (MUM-benchmark) framework is released as an open source project under the Apache License 2.0.[13]

The query generator accepts an XML file containing parameterised queries (in SPARQL or SQL) and the respective queries to select a random parameter value from the data store for each parameter. This allows us to generate queries based on data from existing data stores, rather than relying on artificially generated data and queries. Examples of such a parameterised query can be found in Appendix A.

The BSBM TestDriver has been extended to measure the performance of relational databases queries via SQL queries and OBDA. The benchmark TestDriver measures the query performance of a *query mix*, which is created manually by specifying a series of queries in a plain text file. The benchmarking process includes a warm-up phase of several runs (default: 50 runs) of the query mix, which is followed by multiple runs (default: 500 runs) whose performance is measured. The results of the benchmark are then output as an XML result file, including aggregated metrics for the query mix, as well as individual metrics for each query in the query mix.

3.5 What We Measure

Due to time constraints, we focused on obtaining results for Benchmark 1 described in section 3.2, which measures the performance of each individual query. We generated 20 distinct query sets in order to ensure that the query performance was not affected by the choice of parameters. Each of the query mixes in the benchmark was run 50 times as a warm-up phase, followed by 100 timed runs;[14] this results in a total of 2,000 measured runs per query.

[13] http://code.google.com/p/mum-benchmark/

[14] Note that the default number of timed runs in the BSBM framework is 500; due to the large number of query sets and the relatively stable times after the warm-up, we reduced the number of timed runs to 100.

In preliminary tests we found that the Quest system performed reasonably well on the query mix, with one significant drawback: The query rewriting consumed a large amount of time for some of the queries, in particular for query 2 (SpeciesPage). This is due to the large number of SQL queries being generated in the SPARQL-to-SQL rewriting process, which then have to be pruned in a time-consuming process. In the case of query 2, the rewriting algorithm generated 7,200 queries, of which 7,198 were pruned, whereas most other queries in the mix generated not more than 120 queries. The time required to rewrite query 2 was nearly 10 seconds on average, which caused extreme delays in the benchmark. Due to a simple caching mechanism, however, the system did not have to perform any rewriting after the first execution of the query, which lead to a significant improvement in the query execution time. We therefore performed the rewriting and caching in the warm-up phase of the benchmark, only measuring the execution time of the SQL query generated by Quest. While this may seem to paint an unrealistic picture of the total execution time of the queries, it does provide us with a measure of the quality of the SPARQL-to-SQL rewriting.

The results are returned using an extended version of the BSBM result XML file, which includes metrics for the query mix (fastest, slowest, and average query mix run time, query mixes per hour), and metrics for the individual queries: Average / min / max query execution time, queries per second, average / min / max number of results per query execution, and number of query timeouts.

3.6 Hardware

The data stores were installed on an "out-of-the-box" Mac Mini with a 2.7 GHz Intel Core i7 dual-core processor, 16 GB (1333 MHz DDR3) memory, and a 750 GB HDD, running Mac OS X 10.7.4 (Lion). The benchmark queries were performed remotely, using an identical machine.

4 Results

The results of the individual query benchmark are shown in Table 4, the unit being queries per second (qps).[15] The SQL query results are given only as a baseline to compare the other datastores against. As stated previously, the results for Quest do not exclude the query re-writing times, but only compare the performance of the SQL queries the OBDA system generates from the SPARQL queries. The SPARQL queries against Virtuoso perform consistently worse than against Quest, with Quest outperforming Virtuoso by roughly an order of magnitude on most queries.

Across all 22 queries, there are large differences in performance for each data store: For 7 of the 22 queries, Virtuoso's performance lies in the single-digit range, a further 10 only range between 12 and 68, and only 5 queries (CommonName,

[15] Please note that query 9 was wrongly generated for Quest, therefore we did not obtain any meaningful results for this query.

FamilyAllfish, FamilyNominalSpecies, Genus, Species) perform better than 100 qps, with the best query execution time for query 4 (Species). The performance even drops to only 1 query per second on average for query 2 (SpeciesPage).

ID	Query name	Virtuoso	Quest	MySQL
1	CommonName	132	850	1262
2	SpeciesPage	1	552	840
3	Genus	167	753	1025
4	Species	192	870	1249
5	FamilyInformation	17	572	840
6	FamilyAllfish	141	704	1113
7	FamilyNominalSpecies	161	773	1060
8	FamilyListOfPictures	50	578	742
9	CollaboratorPage	27		824
10	PicturePage	68	711	1166
11	CAllFish	24	316	629
12	CSpeciesInformation	7	598	1009
13	CFreshwater	6	201	212
14	CIntroduced	9	303	479
15	CEndemic	13	656	985
16	CReefAssociated	4	266	378
17	CPelagic	7	337	532
18	CGameFish	9	580	845
19	CCommercial	12	556	748
20	CUsedForAquaculture	12	779	1229
21	CPotentialAquaculture	34	694	957
22	CAquariumTrade	66	815	1251

Table 4. Query results of the 22 queries in qps (queries per second)

The behaviour of Quest seems more stable compared to Virtuoso: 16 of the 22 queries perform at between 550 and 870 qps, with only query 13 (Country Freshwater, i.e. retrieve all freshwater species of a given country, an instantiation of which is shown in Appendix B) and 16 (Country Reef Associated) causing a significant drop in performance across all three systems. This is surprising, as the Country queries (query 11 to 22) are all similar, but rank among the best and worst performing queries for the Quest system. The search for a Species (query 4) as well as the Common Name query (query 1) are executed at the highest speed, with 870 and 850 qps, respectively.

In summary, it can be said that the SQL queries generated by Quest perform surprisingly well compared to the Virtuoso RDF store. As Virtuoso has consistently performed well in existing benchmarks (see Section 2), we have reason to assume that the comparatively weak performance is not restricted to Virtuoso, but rather a general issue of triple stores. While the results of the native SQL queries against the MySQL database are only given as a baseline, it is clear to

see that the results of the queries against Quest come fairly close to those of the native SQL queries.

5 Conclusions and Future Work

In this paper, we have presented a multi-purpose benchmarking framework, which allows benchmark users to generate randomised queries from existing data. We have used this framework to generate a benchmark using data from FishBase, a large collection of information about the world's fish. The *FishMark* has been employed to measure the performance of several data stores. While the work on this benchmark is in its early stages, we have found that the combination of real data, query mixes inferred from server logs, automated query instantiation and query benchmarking over several different systems (RDF triple store, SQL RDBMS, OBDA using an RDBMS), makes for a promising approach to performance measurement. The OBDA system we tested outperformed the RDF store by approximately an order of magnitude, while being surprisingly close in performance to FishBase's native relational database.

For future work, there are a number of possible ways to extend and improve the MUM-benchmarking framework. The current version only generates independent parameter values for the queries in a query mix rather than a *sequence* of queries in which each parameter value depends on the parameter values in previous queries. Sequencing would generate a more natural set of queries for a query mix.

We are planning to develop strategies for scaling the FishBase data based on a single fish species. This will allow us to test the performance of various data engines using self-contained subsets of the FishBase data. Another next step is the generation of more realistic query mixes based on the information extracted from the FishBase server logs. Additionally, we are aiming to make another attempt at using the complete FishBase data set (1.38 billion triples) for the benchmark.

The main purpose of the benchmarking results in this report is to demonstrate the FishMark benchmark; therefore we only tested a very restricted number of systems on a single query mix. We aim to perform more extensive tests with FishMark using a larger set of RDF stores, SPARQL-to-SQL rewriters, and OBDA systems. Finally, we did not attempt to measure the query performance under the load of multiple clients, which is a natural next step in the development of the FishMark benchmark.

References

1. Artale, A., Calvanese, D., Kontchakov, R., Zakharyaschev, M.: The DL-Lite family and relations. J. of Artificial Intelligence Research 36, 1–69 (2009)
2. Bizer, C., Schultz, A.: The Berlin SPARQL benchmark. Int. J. Semantic Web Inf. Syst. 5(2), 1–24 (2009)

3. Boncz, P., Pham, M.D., Erling, O., Mikhailov, I., Rankka, Y.: Social network intelligence benchmark (SIB) - version 0.8. http://www.w3.org/wiki/Social_Network_Intelligence_BenchMark (2011)
4. Calvanese, D., Giacomo, G.D., Lembo, D., Lenzerini, M., Poggi, A., Rodriguez-Muro, M., Rosati, R., Ruzzi, M., Savo, D.F.: The MASTRO system for ontology-based data access. Semantic Web J. 2(1), 43–53 (2011)
5. Guo, Y., Pan, J.Z., Heflin, J.: LUBM: A benchmark for OWL knowledge base systems. J. of Web Semantics 3(2-3), 158–182 (2005)
6. Kontchakov, R., Lutz, C., Toman, D., Wolter, F., Zakharyaschev, M.: The combined approach to ontology-based data access. In: Proc. of IJCAI-11. pp. 2656–2661 (2011)
7. Ma, L., Yang, Y., Qiu, Z., Xie, G., Pan, Y., Liu, S.: Towards a complete OWL ontology benchmark. In: Proc. of ESWC-06. pp. 125–139 (2006)
8. Morsey, M., Lehmann, J., Auer, S., Ngomo, A.C.N.: DBpedia SPARQL benchmark - performance assessment with real queries on real data. In: Proc. of ISWC-11. pp. 454–469 (2011)
9. Rodriguez-Muro, M., Calvanese, D.: Quest, an OWL 2 QL reasoner for ontology-based data access. In: Proc. of OWLED-12 (2012)
10. Schmidt, M., Hornung, T., Lausen, G., Pinkel, C.: SP2Bench: A SPARQL performance benchmark. In: Proc. of ICDE-09. pp. 222–233 (2009)
11. Weiske, C., Auer, S.: Implementing SPARQL support for relational databases and possible enhancements. In: Proc. of CSSW-07. pp. 69–80 (2007)

Appendix A: Sample Query Template

Query Template for Query 2: SpeciesPage (SPARQL)

```
<bmquery name="SpeciesPage">
     <query>
         <![CDATA[
             PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
             PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>
             PREFIX fish: <http://fishdelish.cs.man.ac.uk/rdf/vocab/resource/>
             SELECT ?common ?code ?refno
             ?author  ?demerspelag ?anacat
             ?family ?order ?class ?entered
             ?pic ?picid ?description ?refauthor ?refyear
             ?collaborator  ?comments
             WHERE {
             ?x fish:species_Genus "%genus%" .
             ?x fish:species_SpecCode ?code.
             ?x fish:species_Species "%species%" .
             ?x fish:species_Comments ?comments .
             OPTIONAL {?x fish:species_Author ?author}.
             OPTIONAL {?x fish:species_FBname ?common}.
             OPTIONAL {?x fish:species_SpeciesRefNo ?refno}.
             OPTIONAL {?ref fish:refrens_RefNo ?refno}.
             OPTIONAL {?ref fish:refrens_Author ?refauthor}.
             OPTIONAL {?ref fish:refrens_Year ?refyear}.
             OPTIONAL {?x fish:species_Comments ?biology.}
             OPTIONAL {
             ?x fish:species_FamCode ?famcode.
             ?famcode fish:families_Family ?family.
             ?famcode fish:families_Order ?order.
             ?famcode fish:families_Class ?class.
             }
             OPTIONAL {?morph fish:morphdat_Speccode ?x.
             ?morph fish:morphdat_AddChars ?description.}
             OPTIONAL {?x fish:species_DemersPelag ?demerspelag.}
             OPTIONAL {?x fish:species_AnaCat ?anacat.}
             OPTIONAL {?x fish:species_PicPreferredName ?pic.
             ?pic_node fish:picturesmain_SpecCode ?x.
             ?pic_node fish:picturesmain_PicName ?pic.
             ?pic_node fish:picturesmain_autoctr ?picid.
             ?pic_node fish:picturesmain_Entered ?entered.
             ?pic_node fish:picturesmain_AuthName ?collaborator.
             }
             }
         ]]>
     </query>
     <parameterquery>
       <paramname>genus</paramname>
       <paramname>species</paramname>
         <paramvalues>
             <query>
                 <![CDATA[
                     PREFIX fd: <http://fishdelish.cs.man.ac.uk/rdf/vocab/resource/>
                     SELECT ?genus ?species
                     WHERE {
                     ?code fd:species_Genus ?genus .
                     ?code fd:species_Species ?species .
                     }
                 ]]>
             </query>
         </paramvalues>
     </parameterquery>
 </bmquery>
```

Appendix B: Sample Benchmark Queries

Instance of Query 1: Common Name (SPARQL)

```
PREFIX fd: <http://fishdelish.cs.man.ac.uk/rdf/vocab/resource/>
SELECT ?type ?species ?genus ?country ?language
WHERE {
?nameID fd:comnames_ComName "Banded wormfish" .
?nameID fd:comnames_NameType ?type .
?nameID fd:comnames_SpecCode ?code .
?nameID fd:comnames_C_Code ?ccode .
?code fd:species_Species ?species .
?code fd:species_Genus ?genus .
?ccode fd:countref_PAESE ?country .
}
```

Instance of Query 2: SpeciesPage (SPARQL)

```
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>
PREFIX fish: <http://fishdelish.cs.man.ac.uk/rdf/vocab/resource/>
SELECT ?common ?code ?refno
?author  ?demerspelag ?anacat
?family ?order ?class ?entered
?pic ?picid ?description ?refauthor ?refyear
?collaborator  ?comments
WHERE {
?x fish:species_Genus "Sebastes" .
?x fish:species_SpecCode ?code.
?x fish:species_Species "nigrocinctus" .
?x fish:species_Comments ?comments .
OPTIONAL {?x fish:species_Author ?author}.
OPTIONAL {?x fish:species_FBname ?common}.
OPTIONAL {?x fish:species_SpeciesRefNo ?refno}.
OPTIONAL {?ref fish:refrens_RefNo ?refno}.
OPTIONAL {?ref fish:refrens_Author ?refauthor}.
OPTIONAL {?ref fish:refrens_Year ?refyear}.
OPTIONAL {?x fish:species_Comments ?biology.}
OPTIONAL {
?x fish:species_FamCode ?famcode.
?famcode fish:families_Family ?family.
?famcode fish:families_Order ?order.
?famcode fish:families_Class ?class.
}
OPTIONAL {?morph fish:morphdat_Speccode ?x.
?morph fish:morphdat_AddChars ?description.}
OPTIONAL {?x fish:species_DemersPelag ?demerspelag.}
OPTIONAL {?x fish:species_AnaCat ?anacat.}
OPTIONAL {?x fish:species_PicPreferredName ?pic.
?pic_node fish:picturesmain_SpecCode ?x.
?pic_node fish:picturesmain_PicName ?pic.
?pic_node fish:picturesmain_autoctr ?picid.
?pic_node fish:picturesmain_Entered ?entered.
?pic_node fish:picturesmain_AuthName ?collaborator.
}
}
```

Instance of Query 3: Genus

```
PREFIX fd: <http://fishdelish.cs.man.ac.uk/rdf/vocab/resource/>
SELECT ?species ?author ?family ?ename
WHERE {
?code fd:species_Species ?species .
?code fd:species_Genus "Parachondrostoma" .
OPTIONAL {?code fd:species_FBname ?ename .}
?code fd:species_Author ?author .
?code fd:species_FamCode ?fcode .
?fcode fd:families_Family ?family .
}
```

Instance of Query 4: Species

```
PREFIX fd: <http://fishdelish.cs.man.ac.uk/rdf/vocab/resource/>
SELECT ?genus ?author ?family ?ename
WHERE {
?code fd:species_Species "ocellatum" .
?code fd:species_Genus ?genus .
OPTIONAL {?code fd:species_FBname ?ename .}
?code fd:species_Author ?author .
?code fd:species_FamCode ?fcode .
?fcode fd:families_Family ?family .
}
```

Instance of Query 5: Family Information

```
 PREFIX fd: <http://fishdelish.cs.man.ac.uk/rdf/vocab/resource/>
SELECT ?order ?class ?noOfGenera ?noOfSpecies ?marine ?brackish ?freshwater ?fossil
?aquarium ?remark  ?division ?activityLevel ?author ?year  ?repguild  ?SpeciesCount
WHERE {
 ?familiesID fd:families_Family "Ipnopidae" .
 ?familiesID fd:families_Order ?order .
 ?familiesID fd:families_Class ?class .
 ?familiesID fd:families_SpeciesCount ?SpeciesCount .
 ?familiesID fd:families_Genera ?noOfGenera .
 ?familiesID fd:families_Species ?noOfSpecies .
 OPTIONAL {?familiesID fd:fossilReference ?fossil }.
 ?familiesID fd:families_Marine ?marine .
 ?familiesID fd:families_Brackish ?brackish .
 ?familiesID fd:families_Freshwater ?freshwater .
 ?familiesID fd:families_Aquarium ?aquarium .
 ?familiesID fd:families_Remark ?remark .
 ?familiesID fd:families_Remark ?remark .
 ?familiesID fd:families_Division ?division .
 ?familiesID fd:families_Activity ?activityLevel .
 ?familiesID fd:families_ReprGuild ?repguild .
 ?familiesID fd:families_FamiliesRefNo ?code .
 ?x fd:refrens_RefNo ?code .
 ?x fd:refrens_Author  ?author .
 ?x fd:refrens_Year ?year .
}
```

Instance of Query 6: Family All Fish

```
PREFIX fd: <http://fishdelish.cs.man.ac.uk/rdf/vocab/resource/>
SELECT ?species ?genus ?author ?englishName
WHERE { ?SpeciesID fd:species_Author ?author ;
fd:species_Species ?species;
fd:species_Genus ?genus ;
fd:species_FamCode ?code .
?code  fd:families_Family "Stromateidae" .
OPTIONAL {?SpeciesID fd:species_FBname ?englishName } .
}
```

Instance of Query 7: Family Nominal Species

```
PREFIX fd: <http://fishdelish.cs.man.ac.uk/rdf/vocab/resource/>
SELECT ?species ?author ?genus ?ref
WHERE { ?SpeciesID fd:species_Author ?author ;
fd:species_Species ?species;
fd:species_Genus ?genus ;
fd:species_FamCode ?code .
OPTIONAL {?SpeciesID fd:species_ImportanceRef ?ref }.
?code  fd:families_Family "Gobiesocidae" .
}
```

Instance of Query 8: Family List of Pictures

```
PREFIX fd: <http://fishdelish.cs.man.ac.uk/rdf/vocab/resource/>
SELECT ?genus ?species ?englishname ?picture ?photographer ?location
WHERE {?picID fd:picturesmain_SpecCode ?code ;
fd:picturesmain_PicName ?picture ;
fd:picturesmain_AuthName ?photographer .
OPTIONAL { ?picID fd:picturesmain_Locality ?location }.
OPTIONAL { ?code fd:species_FBname ?englishname } .
?code fd:species_Species ?specie;
fd:species_Genus ?genus ;
fd:species_FamCode ?fcode .
?fcode fd:families_Family "Moronidae" .
}
```

Instance of Query 9: Collaborator Page

```
PREFIX fd: <http://fishdelish.cs.man.ac.uk/rdf/vocab/resource/>
SELECT ?prename ?surname  ?email ?photo ?webpage ?fax ?institute ?street ?city ?country ?comments ?keywords ?year
WHERE {
?x fd:collaborators_Personnel "1952" .
OPTIONAL {?x fd:collaborators_Prename ?prename }.
OPTIONAL {?x fd:collaborators_Surname ?surname }.
OPTIONAL {?x fd:collaborators_E-mail ?email }.
OPTIONAL {?x fd:collaborators_StaffPhoto ?photo }.
OPTIONAL {?x fd:collaborators_WebPage ?webpage }.
OPTIONAL {?x fd:collaborators_FAX ?fax }.
OPTIONAL {?x fd:collaborators_Institute ?institute }.
```

```
OPTIONAL {?x fd:collaborators_Street ?street }.
OPTIONAL {?x fd:collaborators_City ?city }.
OPTIONAL {?x fd:collaborators_Country ?country }.
OPTIONAL {?x fd:collaborators_Comments ?comments }.
OPTIONAL {?x fd:collaborators_Keywords ?keywords }.
OPTIONAL {?x fd:collaborators_Year ?year }.
}
```

Instance of Query 10: Picture Page

```
PREFIX fd: <http://fishdelish.cs.man.ac.uk/rdf/vocab/resource/>
SELECT ?genus ?species ?photographer ?size ?location ?stage  ?reference ?remark
WHERE {
?pcode fd:picturesmain_PicName "Danav_u0.jpg" .
?pcode fd:picturesmain_AuthName ?photographer .
OPTIONAL {?pcode fd:picturesmain_Size ?size }.
OPTIONAL {?pcode fd:picturesmain_Locality ?location }.
?pcode fd:picturesmain_LifeStage ?stage .
OPTIONAL {?pcode fd:picturesmain_Reference ?reference }.
OPTIONAL {?pcode fd:picturesmain_Remark ?remark }.
?pcode fd:picturesmain_SpecCode ?scode .
?scode fd:species_Genus ?genus .
?scode fd:species_Species ?species .
}
```

Instance of Query 10: Family All Fish

```
PREFIX fd: <http://fishdelish.cs.man.ac.uk/rdf/vocab/resource/>
SELECT ?species ?genus ?author ?englishName
WHERE { ?SpeciesID fd:species_Author ?author ;
fd:species_Species ?species;
fd:species_Genus ?genus ;
fd:species_FamCode ?code .
?code  fd:families_Family "Stromateidae" .
OPTIONAL {?SpeciesID fd:species_FBname ?englishName } .
}
```

Instance of Query 13: Country Freshwater

```
PREFIX fd: <http://fishdelish.cs.man.ac.uk/rdf/vocab/resource/>
SELECT ?order ?family ?genus ?species ?occurrence ?fbname ?name
WHERE {
?nameID fd:comnames_ComName ?name .
?nameID fd:comnames_C_Code ?ccode .
?nameID fd:comnames_SpecCode ?x.
?x fd:species_Genus ?genus .
?x fd:species_Species ?species .
OPTIONAL {?x fd:species_FBname ?fbname }.
?x fd:species_FamCode ?f .
?f fd:families_Family  ?family .
?f fd:families_Order ?order .
?c fd:country_SpecCode ?x.
?c fd:country_Status ?occurrence .
?c fd:country_Freshwater 1 .
?c fd:country_C_Code ?cf .
?cf fd:countref_PAESE "Ghana".
}
```

Instance of Query 16: Country Reef Associated

```
PREFIX fd: <http://fishdelish.cs.man.ac.uk/rdf/vocab/resource/>
SELECT ?order ?family ?genus ?species ?occurrence ?fbname ?name ?dangerous
WHERE {
?nameID fd:comnames_ComName ?name .
?nameID fd:comnames_C_Code ?ccode .
?nameID fd:comnames_SpecCode ?x.
?x fd:species_Genus ?genus .
?x fd:species_Species ?species .
?x fd:species_Dangerous ?dangerous .
?x fd:species_DemersPelag "reef-associated" .
OPTIONAL {?x fd:species_FBname ?fbname }.
?x fd:species_FamCode ?f .
?f fd:families_Family  ?family .
?f  fd:families_Order ?order .
?c fd:country_SpecCode ?x.
?c fd:country_Status ?occurrence .
?c fd:country_C_Code ?cf .
?cf fd:countref_PAESE "Trinidad Tob" .
}
```

The Combined Approach to OBDA: Taming Role Hierarchies using Filters

Carsten Lutz[1], İnanç Seylan[1], David Toman[2], and Frank Wolter[3]

[1]Universität Bremen, Germany
{clu,seylan}@informatik.uni-bremen.de
[2]Cheriton School of CS, University of Waterloo, Canada
david@cs.uwaterloo.ca
[3]University of Liverpool, United Kingdom
wolter@liverpool.ac.uk

Abstract. There are several approaches to implementing query answering over instance data in the presence of an ontology that target conventional relational database systems (SQL databases) as their back-end. They all share the limitation that, for ontologies formulated in OWL2 QL or versions of the description logic DL-Lite that admit both role hierarchies and inverse roles, it seems impossible to avoid an exponential blowup of the query (and sometimes this is even provable). We consider the *combined approach* and propose to replace its query rewriting part with a *filtering technique*. This is natural from an implementation perspective and allows us to handle both inverse roles and role hierarchies without an exponential blowup. We also carry out an experimental evaluation that demonstrates the scalability of this approach.

1 Introduction

In recent years, ontology-based data access (OBDA) has emerged as a promising and challenging application of ontologies. The idea is to enrich instance data with a 'semantic layer' in the form of an ontology, used as an interface for querying and to derive additional answers. A central research problem in this area is to design query answering engines that can deal with sufficiently expressive ontology languages yet scale to large data sets. The most popular ontology languages that have been considered for OBDA include the three profiles OWL2 RL, OWL2 QL, and OWL2 EL, as well as various description logics and datalog variants related to these profiles [2, 3, 5, 11, 15].

Currently, there are two major methodologies for answering queries in an OBDA setting: rewriting-based approaches (also called backward chaining) and materialization-based approaches (also called forward chaining). In the former, one compiles the ontology $\mathcal{T}$ and the query q into a new query $q_{\mathcal{T}}$ that contains the relevant knowledge from the ontology, i.e., the answers to q over $\mathcal{A}$ and $\mathcal{T}$ coincide with the answers to $q_{\mathcal{T}}$ over $\mathcal{A}$. One can thus store $\mathcal{A}$ in a relational database management system RDBMS and execute $q_{\mathcal{T}}$ over $\mathcal{A}$. In materialization

approaches, the instance data $\mathcal{A}$ is completed with the relevant knowledge from the ontology $\mathcal{T}$, i.e. for any query q, the answers given to q over $\mathcal{A}$ and $\mathcal{T}$ coincide with the answers given to q over the completed instance data $\mathcal{A}_\mathcal{T} \supseteq \mathcal{A}$ without any ontology. Thus, one can store $\mathcal{A}_\mathcal{T}$ in a relational database management system (RDBMS) and execute q over $\mathcal{A}_\mathcal{T}$.

A technical problem that arises in materialization approaches is that the completed data $\mathcal{A}_\mathcal{T}$ easily becomes infinite; in particular, this may happen when the ontology expresses cyclic dependencies and has existential quantifiers in the heads of its concept inclusions, which is allowed in most ontology languages including the ones mentioned above. To overcome this problem, an economic way of reusing individuals introduced for existential quantifiers has been proposed in [8, 9] for the case where ontologies are formulated in description logics from the $\mathcal{EL}$ and DL-Lite families, which are the logical cores of the OWL2 EL and OWL2 QL ontology languages. While the resulting completed data sets are finite and give correct answers to instance queries, they can give spurious answers to conjunctive queries (CQs). To recover soundness for CQs, it is thus necessary to include an additional step, resulting in the *combined approach* to query answering: the original query is rewritten in a way that eliminates spurious answers. In sharp contrast to pure rewriting, the *auxiliary query rewriting* required in the combined approach turns out to be rather simple—an additional *selection condition* applied to the results of the original CQ over the completed data—and often of polynomial size. Indeed, experiments indicate that the combined approach admits very efficient query execution for expressive variants of $\mathcal{EL}$ and DL-Lite [8, 9].

Unfortunately, there are certain combinations of logical operators that are important from an application perspective, but for which an exponential blowup of the query seems to be unavoidable both in the query rewriting approach and in the combined approach. In particular, this is the case for the combination of inverse roles and role hierarchies as found in DL-Lite$_\mathcal{R}$ [3], the extension of basic DL-Lite with role hierarchies that underpins OWL2 QL. It has been shown that, in the query rewriting approach, an exponential blowup of the query size is unavoidable when the ontology is formulated in DL-Lite$_\mathcal{R}$ [7]. For the combined approach, an auxiliary query rewriting strategy for DL-Lite$_\mathcal{R}$ ontologies and CQs is presented in [8], but it incurs an exponential blowup and it seems unlikely that the rewriting can be improved to a poly-sized one (although this question is yet to be resolved).

In this paper, we present a new variation on the combined approach that can handle CQs and DL-Lite$_\mathcal{R}$ ontologies and eliminates the need for auxiliary query rewriting altogether, thus also eliminating the need to deal with exponentially sized queries. Specifically, we replace auxiliary query rewriting with a *filtering component*: spurious answers are eliminated by a polynomial-time filtering procedure (called a *filter* in the rest of the paper) that is installed as a user-defined function in the underlying RDBMS. Our main contributions are as follows.

(1) We develop a polynomial time procedure for filtering out spurious answers to CQs for ontologies formulated in DL-Lite$_\mathcal{R}$. Interestingly, the existence of such

a filtering procedure appears to be quite sensitive to how exactly the instance data is completed. Compared to the data completion for the original combined approach [8], the filtering technique requires subtle modifications to the data completion in order to obtain a polytime filter.

(2) To analyze the performance of our approach and to compare it with the query rewriting approach, we modify the Lehigh University Benchmark (LUBM) [6] by introducing additional existential restrictions and subconcepts into the LUBM ontology and incompleteness into the LUBM Data Generator. Additional queries that are designed to stress-test the performance of the filtering procedure are introduced as well.

(3) Finally, the resulting ontologies, data, and queries are used to demonstrate the feasibility of our approach, and to show that it scales to large amounts of instance data.

Some technical proofs and details of our experimental evaluation are presented in the appendix of the full version of this paper, available at `http://informatik.uni-bremen.de/~clu/combined/`

2 Preliminaries

We introduce DL-Lite$_\mathcal{R}$-TBoxes, ABoxes, and conjunctive queries. Let $\mathsf{N_I}$, $\mathsf{N_C}$, and $\mathsf{N_R}$ be countably infinite sets of *individual names*, *concept names* and *role names*. *Roles* R, *simple concepts* C, and *concepts* D are built according to the following syntax rules, where P ranges over $\mathsf{N_R}$ and A over $\mathsf{N_C}$:

$$R ::= P \mid P^-, \qquad C ::= A \mid \exists R, \qquad D ::= C \mid \neg C \mid \exists R.A.$$

As usual, we use $\mathsf{N_R^-}$ to denote the set of all roles and identify $(P^-)^-$ with P. In DL-Lite$_\mathcal{R}$, a *TBox* is a finite set $\mathcal{T}$ of *concept inclusions (CIs)* $C \sqsubseteq D$ with C a simple concept and D a concept, and *role inclusions (RIs)* $R_1 \sqsubseteq R_2$ with R_1, R_2 roles.[1]

An *ABox* is a finite set of *concept assertions* $A(a)$ and *role assertions* $P(a, b)$, where $A \in \mathsf{N_C}$, $P \in \mathsf{N_R}$ and $a, b \in \mathsf{N_I}$. We denote by $\mathsf{Ind}(\mathcal{A})$ the set of individual names that occur in $\mathcal{A}$, and write $P^-(a, b) \in \mathcal{A}$ instead of $P(b, a) \in \mathcal{A}$ whenever convenient. A *knowledge base* (*KB*) is a pair $(\mathcal{T}, \mathcal{A})$ with $\mathcal{T}$ a TBox and $\mathcal{A}$ an ABox.

The semantics of TBoxes and ABoxes is defined in the standard way based on interpretations $\mathcal{I} = (\Delta^\mathcal{I}, \cdot^\mathcal{I})$, where $\Delta^\mathcal{I}$ is a non-empty *domain* and $\cdot^\mathcal{I}$ an *interpretation function* that maps each $A \in \mathsf{N_C}$ to a subset $A^\mathcal{I} \subseteq \Delta^\mathcal{I}$, each $P \in \mathsf{N_R}$ to a relation $P^\mathcal{I} \subseteq \Delta^\mathcal{I} \times \Delta^\mathcal{I}$, and each $a \in \mathsf{N_I}$ to an element $a^\mathcal{I} \in \Delta^\mathcal{I}$; for details consult [1, 3]. An interpretation is a *model* of a TBox $\mathcal{T}$ if it satisfies all inclusions in $\mathcal{T}$; models of ABoxes and knowledge bases are defined analogously. A knowledge base is *consistent* if it has a model. For a CI or RI α, we write $\mathcal{T} \models \alpha$ when α is a consequence of $\mathcal{T}$ (satisfied in all models of $\mathcal{T}$). Instead of

[1] A set of role inclusions is also called a *role hierarchy*.

$\mathcal{T} \models R \sqsubseteq S$, we will usually write $R \sqsubseteq^*_{\mathcal{T}} S$ to clearly distinguish consequences of this form (which are RIs) from consequences of the form $\mathcal{T} \models \exists R \sqsubseteq \exists S$ (which are CIs). Note that, in DL-Lite$_{\mathcal{R}}$, deciding consistency and logical consequence amounts to computing a form of transitive closure [3].

Let $\mathsf{N_V}$ be a countably infinite set of *variables*. Taken together, the sets $\mathsf{N_V}$ and $\mathsf{N_I}$ form the set $\mathsf{N_T}$ of *terms*. A *first-order* (*FO*) *query* is a first-order formula $q = \varphi(\boldsymbol{x})$ in the signature $\mathsf{N_C} \cup \mathsf{N_R}$ and with terms from $\mathsf{N_T}$, where the concept and role names are treated as unary and binary predicates, respectively, and $\boldsymbol{x}$ are the free variables of φ called the *answer variables*; we say that q is *k-ary* if $\boldsymbol{x}$ comprises k variables. If $k = 0$, then q is a *Boolean* query. A *conjunctive query (CQ)* is an FO query of the form $q = \exists \boldsymbol{y}\, \psi(\boldsymbol{y}, \boldsymbol{x})$, where ψ is a conjunction of *concept atoms* $A(t)$ and *role atoms* $P(t, t')$ where $t, t' \in \mathsf{N_T}$. As in the case of ABox assertions, we do not distinguish between $P^-(t, t')$ and $P(t', t)$. We denote by $\mathsf{term}(q)$ the set of terms in q.

Let $q = \varphi(\boldsymbol{x})$ be a k-ary FO query with $\boldsymbol{x} = x_1, \dots, x_k$, and $\mathcal{I}$ an interpretation. A mapping $\pi \colon \mathsf{term}(q) \to \Delta^{\mathcal{I}}$ with $\pi(a) = a^{\mathcal{I}}$ for all $a \in \mathsf{term}(q) \cap \mathsf{N_I}$ is a *match* for q in $\mathcal{I}$ if $\mathcal{I}$ satisfies q under the variable assignment that maps each $t \in \mathsf{term}(q)$ to $\pi(t)$; in this case, we write $\mathcal{I} \models^{\pi} q$. For a k-tuple of individual names $\boldsymbol{a} = a_1, \dots, a_k$, a match π for q in $\mathcal{I}$ is an *$\boldsymbol{a}$-match* if $\pi(x_i) = a_i^{\mathcal{I}}$ for $i \leq k$. We say that $\boldsymbol{a}$ is an *answer* to q in an interpretation $\mathcal{I}$ if there is an $\boldsymbol{a}$-match for q in $\mathcal{I}$ and use $\mathsf{ans}(q, \mathcal{I})$ to denote the set of all answers to q in $\mathcal{I}$. Finally, $\boldsymbol{a}$ is a *certain answer* to q over a KB $\mathcal{K} = (\mathcal{T}, \mathcal{A})$ if $\boldsymbol{a} \subseteq \mathsf{Ind}(\mathcal{A})$ and $\mathcal{I} \models q[\boldsymbol{a}]$ for all models $\mathcal{I}$ of $\mathcal{K}$. The set of all certain answers to q over $\mathcal{K}$ is denoted by $\mathsf{cert}(q, \mathcal{K})$. The query answering problem considered in this paper is: given a DL-Lite$_{\mathcal{R}}$ knowledge base $\mathcal{K}$ and a CQ q, compute $\mathsf{cert}(q, \mathcal{K})$.

To simplify notation, throughout the paper we adopt the *unique name assumption* (UNA), i.e., require that $a^{\mathcal{I}} \neq b^{\mathcal{I}}$ for distinct $a, b \in \mathsf{N_I}$. This assumption has no impact on the query answering problem.

3 ABox Completion

As explained in the introduction, the central idea of the combined approach is to materialize consequences of the TBox in the ABox as a preprocessing step, and then to execute queries over the completed data stored in an RDBMS as a plain table. We illustrate this using two examples from the university domain, similar in spirit to the LUBM ontology from [6] used in the experimental evaluation.

Example 1. For any ABox $\mathcal{A}$, the concept inclusions

$$\mathsf{Student} \sqsubseteq \mathsf{Person} \tag{1}$$

$$\mathsf{Student} \sqsubseteq \exists \mathsf{takesCourse} \tag{2}$$

lead to the following additions: (1) for every assertion $\mathsf{Student}(a) \in \mathcal{A}$, add (1) $\mathsf{Person}(a)$ and (2) $\mathsf{takesCourse}(a, b)$ for some fresh individual b (unless such assertions are already present). After this completion, a CQ such as

$$q_1(x) = \exists y\, \mathsf{Person}(x) \wedge \mathsf{takesCourse}(x, y)$$

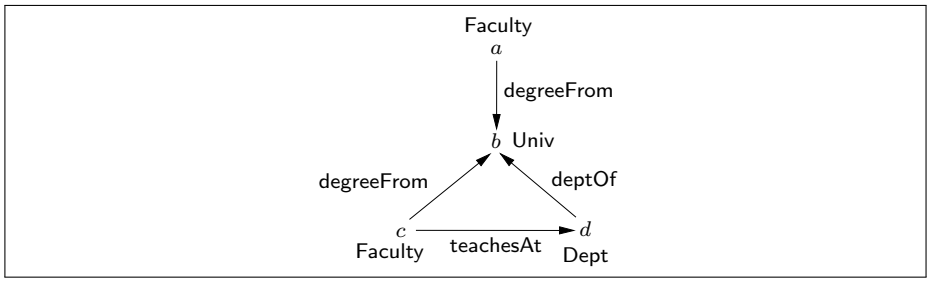


Fig. 1. Completed ABox for Example 3.

correctly returns each a with $\mathsf{Student}(a) \in \mathcal{A}$ *as a certain answer.*

The following example shows that naive completion can result in infinite ABoxes.

Example 2. Completed naively, the ABox $\{\mathsf{Faculty}(a)\}$ *and LUBM inclusions*

$$\mathsf{Faculty} \sqsubseteq \exists \mathsf{degreeFrom} \qquad \exists \mathsf{degreeFrom}^- \sqsubseteq \mathsf{Univ} \tag{3}$$

$$\mathsf{Univ} \sqsubseteq \exists \mathsf{deptOf}^- \qquad \exists \mathsf{deptOf} \sqsubseteq \mathsf{Dept} \tag{4}$$

$$\mathsf{Dept} \sqsubseteq \exists \mathsf{teachesAt}^- \qquad \exists \mathsf{teachesAt} \sqsubseteq \mathsf{Faculty} \tag{5}$$

result in an infinite role chain that indefinitely repeats the roles $\mathsf{degreeFrom}$, deptOf^-, *and* $\mathsf{teachesAt}^-$.

The problem can be overcome by reusing fresh individuals in an economic way.

Example 3. Consider again the TBox (3)-(5). By reusing individuals, the ABox $\{\mathsf{Faculty}(a)\}$ *can be completed as shown in Figure 1, replacing the infinite role chain with a cycle. Individual reuse compromises soundness of query answering as some queries now have spurious answers; for example, the CQ*

$$q_2(x) = \exists y, z\ \mathsf{Faculty}(x) \land \mathsf{degreeFrom}(x, y) \land \mathsf{Univ}(y) \land \mathsf{deptOf}(z, y) \land \mathsf{Dept}(z) \land \mathsf{teachesAt}(x, z)$$

returns c as an answer when executed over the completed ABox shown in Figure 1. This answer is spurious for two reasons: first, the cycle in Figure 1 is present only due to individual reuse and thus should be disregarded for query matches; and second, the freshly introduced individuals b, c, d are 'labeled nulls' and thus can never be returned as answers.

To recover soundness, it is necessary to eliminate the spurious answers. In the original combined approach, this was achieved by query rewriting [8, 9]. In this paper, the spurious answers are eliminated by a filtering procedure that is installed as a user-defined function in the RDBMS. In the remainder of this section, we introduce ABox completion in full detail. In the subsequent section, we describe the filtering procedure.

From a conceptual perspective, the ABox completion step can be viewed as replacing the original ABox with the *canonical model* $\mathcal{I}_\mathcal{K}$ of the knowledge

base $\mathcal{K}$ [8]. To define $\mathcal{I}_\mathcal{K}$, we need a few preliminaries. From now on, we will generally disallow concepts of the form $\exists R.C$. This can be done without loss of generality since each CI $D \sqsubseteq \exists R.C$ can be replaced with $D \sqsubseteq \exists R_C$, $R_C \sqsubseteq R$, and $\exists R_C^- \sqsubseteq C$, where R_C is a fresh role name.

Let $\mathcal{K} = (\mathcal{T}, \mathcal{A})$ be a DL-Lite$_\mathcal{R}$ KB. We use $\mathsf{rol}(\mathcal{T})$ to denote the set of all role names in $\mathcal{T}$ plus their inverses. The canonical model comprises at most two fresh individuals for every role in $\mathsf{rol}(\mathcal{T})$. However, we only want to introduce the fresh individuals for a given role when necessary. Formally, we call a role $R \in \mathsf{rol}(\mathcal{T})$ *generating in* $\mathcal{T}$ if there exist an $a \in \mathsf{Ind}(\mathcal{A})$ and $R_0, \dots, R_n \in \mathsf{rol}(\mathcal{T})$ such that $R_n = R$ and the following conditions hold:

(agen) $\mathcal{K} \models \exists R_0(a)$ and $R_0(a, b) \notin \mathcal{A}$ for all $b \in \mathsf{Ind}(\mathcal{A})$ (written $a \rightsquigarrow \exists R_0$),

(rgen) for $i < n$, $\mathcal{T} \models \exists R_i^- \sqsubseteq \exists R_{i+1}$ and $R_i^- \neq R_{i+1}$ (written $\exists R_i^- \rightsquigarrow \exists R_{i+1}$).

To facilitate the implementation of efficient filters, we refine the definition of canonical models as given in [8]: in some cases, we introduce two fresh individuals for a given role instead of only a single one. The need for the additional individual is related to particular role configurations in the TBox called a loop: a set $\{R, S\} \subseteq \mathsf{rol}(\mathcal{T})$ (where potentially $R = S$) is a *loop in* $\mathcal{T}$ if $R \neq S^-$, $\mathcal{T} \models \exists R^- \sqsubseteq \exists S$, $\mathcal{T} \models \exists S^- \sqsubseteq \exists R$, and there is some $T \in \mathsf{rol}(\mathcal{T})$ such that $S^- \sqsubseteq^*_\mathcal{T} T$ and $R \sqsubseteq^*_\mathcal{T} T$. Let $\mathfrak{L}_\mathcal{T}$ denote the set of all roles that occur in a loop in $\mathcal{T}$. The canonical model $\mathcal{I}_\mathcal{K}$ is then based on the domain

$$\begin{aligned}\Delta^{\mathcal{I}_\mathcal{K}} = \mathsf{Ind}(\mathcal{A}) \cup \{c_{R,0} \mid R \in \mathsf{rol}(\mathcal{T}) \setminus \mathfrak{L}_\mathcal{T} \text{ is generating in } \mathcal{K}\} \\ \cup \{c_{R,0}, c_{R,1} \mid R \in \mathfrak{L}_\mathcal{T} \text{ is generating in } \mathcal{K}\}.\end{aligned}$$

To define the extension of roles in $\mathcal{I}_\mathcal{K}$, we need some additional preparation. Let "$\prec$" be an arbitrary, but fixed total ordering on $\mathsf{rol}(\mathcal{T})$. For all $d, d' \in \Delta^{\mathcal{I}_\mathcal{K}}$ and each role R, we write $d \rightsquigarrow_R d'$ whenever there is an S such that $S \sqsubseteq^*_\mathcal{T} R$ and one of the following cases applies:

- $d = a \in \mathsf{Ind}(\mathcal{A})$, $a \rightsquigarrow \exists S$, and $d' = c_{S,0}$;
- $d = c_{T,i}$, $\exists T^- \rightsquigarrow \exists S$, $d' = c_{S,j}$, and one of the following holds
  - $i = j$ and $\{S, T\}$ is not a loop in $\mathcal{T}$;
  - $i = j$, $\{S, T\}$ is a loop in $\mathcal{T}$, and $S \prec T$;
  - $i = \overline{j}$, $\{S, T\}$ is a loop in $\mathcal{T}$, and $T = S$ or $T \prec S$ (for $\overline{0} = 1$ and $\overline{1} = 0$).

The *canonical model* $\mathcal{I}_\mathcal{K}$ *for* $\mathcal{K}$ is now defined as follows, based on the domain $\Delta^{\mathcal{I}_\mathcal{K}}$ introduced above:

$$\begin{aligned}A^{\mathcal{I}_\mathcal{K}} &= \{a \in \mathsf{Ind}(\mathcal{A}) \mid \mathcal{K} \models A(a)\} \cup \{c_{R,i} \in \Delta^{\mathcal{I}_\mathcal{K}} \mid \mathcal{T} \models \exists R^- \sqsubseteq A\}, \\ R^{\mathcal{I}_\mathcal{K}} &= \{(a, b) \in \mathsf{Ind}(\mathcal{A}) \times \mathsf{Ind}(\mathcal{A}) \mid \exists S : S(a, b) \in \mathcal{A} \text{ and } S \sqsubseteq^*_\mathcal{T} R\} \cup \\ &\quad \{(d, d') \in \Delta^{\mathcal{I}_\mathcal{K}} \mid d \rightsquigarrow_R d' \text{ or } d' \rightsquigarrow_{R^-} d\}, \\ a^{\mathcal{I}_\mathcal{K}} &= a.\end{aligned}$$

The ABox completion consists of replacing the ABox $\mathcal{A}$ originally stored in the RDBMS with its canonical model $\mathcal{I}_\mathcal{K}$. This can be achieved by executing a set of FO/SQL-queries whose size is polynomial in the size of $\mathcal{T}$ [8].

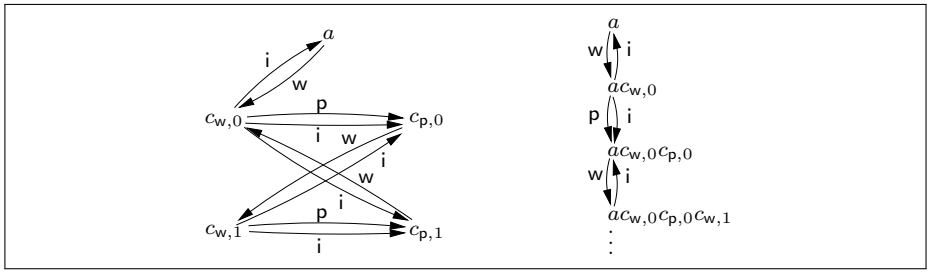


Fig. 2. Canonical model $\mathcal{I}_\mathcal{K}$ and unraveled canonical model $\mathcal{U}_\mathcal{K}$ for Example 5.

It can be shown that $\mathcal{I}_\mathcal{K}$ is a model of $\mathcal{K}$ whenever $\mathcal{K}$ is consistent. Note that one can find a Boolean CQ $q_\mathcal{T}$ of size polynomial in the size of $\mathcal{T}$ such that for any ABox $\mathcal{A}$ stored in the RDBMS, $q_\mathcal{T}$ gives a positive answer iff $\mathcal{K} = (\mathcal{T}, \mathcal{A})$ is consistent [8]. We can thus safely assume that the knowledge base has been tested for consistency before query answering.

Example 4. Reconsider Examples 2 and 3. The canonical model $\mathcal{I}_\mathcal{K}$ for the ABox $\{\mathsf{Faculty}(a)\}$ and TBox (3)-(5) is the structure displayed in Figure 1. Following our construction above, the fresh individuals b, c, d are named $c_{\mathsf{degreeFrom},0}$, $c_{\mathsf{teachesAt}^-,0}$, and $c_{\mathsf{deptOf}^-,0}$. Note that the TBox (3)-(5) does not give rise to any loops, and thus all $c_{R,i}$ have index $i = 0$.

Example 5. The following TBox gives rise to the loop $\{\mathsf{worksFor}, \mathsf{paysSalaryOf}\}$:

$$\mathsf{Employee} \sqsubseteq \exists \mathsf{worksFor} \qquad \exists \mathsf{worksFor}^- \sqsubseteq \mathsf{Employer} \tag{6}$$

$$\mathsf{Employer} \sqsubseteq \exists \mathsf{paysSalaryOf} \qquad \exists \mathsf{paysSalaryOf}^- \sqsubseteq \mathsf{Employee} \tag{7}$$

$$\mathsf{worksFor}^- \sqsubseteq \mathsf{isAffiliatedWith} \qquad \mathsf{paysSalaryOf} \sqsubseteq \mathsf{isAffiliatedWith}. \tag{8}$$

The canonical model $\mathcal{I}_\mathcal{K}$ for the ABox $\{\mathsf{Employee}(a)\}$ and the TBox (6)-(8) with $\mathsf{paysSalaryOf} \prec \mathsf{worksFor}$ is shown on the left-hand side of Figure 2, where concept names are omitted and role names are abbreviated by their first letter.

Note that a more straightforward version of the canonical model could be obtained by identifying all elements $c_{R,0}$ and $c_{R,1}$. Section 4 explains why this leads to problems for efficient filtering. Also note that real world ontologies seem to contain only very few loops, if any, and thus having two domain elements per role that participates in a loop should not significantly increase the size of canonical models in practical cases.

To characterize the spurious answers that have to be filtered out, it is useful to introduce an unraveled (infinite) version of canonical models. Let $\mathcal{K}$ be a knowledge base. A *path* is a finite sequence $ad_1 \cdots d_n$, $n \geq 0$, such that $a \in \mathsf{Ind}(\mathcal{A})$, $d_1, \ldots, d_n \in \Delta^{\mathcal{I}_\mathcal{K}} \setminus \mathsf{Ind}(\mathcal{A})$, $a \rightsquigarrow_R d_1$ for some $R \in \mathsf{N}_\mathsf{R}^-$, and $d_i \rightsquigarrow_R d_{i+1}$ for some $R \in \mathsf{N}_\mathsf{R}^-$, $1 \leq i < n$. We denote by $\mathsf{tail}(\sigma)$ the last element of the path σ.

The unraveled canonical model $\mathcal{U}_\mathcal{K}$ is then defined by taking:

$$\begin{aligned} \Delta^{\mathcal{U}_\mathcal{K}} &\text{ is the set of all paths in } \mathcal{I}_\mathcal{K}, \\ a^{\mathcal{U}_\mathcal{K}} &= a, \text{ for all } a \in \mathsf{Ind}(\mathcal{A}), \\ A^{\mathcal{U}_\mathcal{K}} &= \{\sigma \in \Delta^{\mathcal{U}_\mathcal{K}} \mid \mathsf{tail}(\sigma) \in A^{\mathcal{I}_\mathcal{K}}\}, \end{aligned}$$

$$\begin{aligned} R^{\mathcal{U}_\mathcal{K}} = {} & \{(a,b) \in \mathsf{Ind}(\mathcal{A}) \times \mathsf{Ind}(\mathcal{A}) \mid \exists S : S(a,b) \in \mathcal{A} \text{ and } S \sqsubseteq^*_\mathcal{T} R\} \cup \\ & \{(\sigma, \sigma d) \mid \sigma d \in \Delta^{\mathcal{U}_\mathcal{K}} \text{ and } \mathsf{tail}(\sigma) \rightsquigarrow_R d\} \cup \\ & \{(\sigma d, \sigma) \mid \sigma d \in \Delta^{\mathcal{U}_\mathcal{K}} \text{ and } \mathsf{tail}(\sigma) \rightsquigarrow_{R^-} d\}. \end{aligned}$$

As an example, the canonical model $\mathcal{U}_\mathcal{K}$ for the KB from Example 4 is shown on the right-hand side of Figure 2. The following result shows that, as one would expect, $\mathcal{U}_\mathcal{K}$ does not suffer from spurious answers.

Theorem 1. *For every consistent DL-Lite$_\mathcal{R}$-KB $\mathcal{K}$ and every CQ q, we have* $\mathsf{cert}(q,\mathcal{K}) = \mathsf{ans}(q,\mathcal{U}_\mathcal{K})$.

The proof of Theorem 1 is standard and omitted, see [8] for a similar proof.

4 Filtering

To remove spurious answers, we install a filtering procedure as a user-defined function in the RDBMS. The procedure takes as input a match of the query in the canonical model $\mathcal{I}_\mathcal{K}$ stored in the RDBMS and returns "false" if this match is spurious and "true" otherwise. We assume that the filtering procedure has access to the query and the TBox, but not to the data. To define its behavior more precisely, we formally define spurious matches based on unraveled canonical models $\mathcal{U}_\mathcal{K}$ and Theorem 1.

Let $\mathcal{K}$ be a KB and $q(\boldsymbol{x})$ a CQ. A match π of q in $\mathcal{I}_\mathcal{K}$ *is reproduced by* a match τ of q in $\mathcal{U}_\mathcal{K}$ if for all $t \in \mathsf{term}(q)$, we have $\pi(t) = \mathsf{tail}(\tau(t))$. We say that π is *spurious* if it is not reproduced by any match τ of q in $\mathcal{U}_\mathcal{K}$. The following lemma, which is an immediate consequence of Theorem 1, shows that $\mathcal{I}_\mathcal{K}$ can be used for query answering when spurious matches are filtered out.

Lemma 1. $\boldsymbol{a} \in \mathsf{cert}(q,\mathcal{K})$ *iff there is an $\boldsymbol{a}$-match π of q in $\mathcal{I}_\mathcal{K}$ that is not spurious.*

We want to show that it can be decided in time polynomial in the size of q and $\mathcal{T}$ whether a given match in $\mathcal{I}_\mathcal{K}$ is spurious. Clearly, it is enough to test for each maximally connected component of q whether the given match is spurious on that component. We can thus w.l.o.g. assume that q is connected.

We need a few preliminaries. An *anonymous path* is a path without the leading individual name, i.e., it is a finite sequence $d_1 \cdots d_n$, $n \geq 1$, such that $d_1, \ldots, d_n \in \Delta^{\mathcal{I}_\mathcal{K}} \setminus \mathsf{Ind}(\mathcal{A})$ and $d_i \rightsquigarrow_R d_{i+1}$ for some $R \in \mathsf{N}_\mathsf{R}^-$, $1 \leq i < n$. We use Paths to denote the set of all paths, both anonymous and non-anonymous. A *root configuration for q given π* is a set $\rho \subseteq \mathsf{term}(q)$ such that one of the following conditions is true:

- ρ is the set of those $t \in \mathsf{term}(q)$ such that $\pi(t) \in \mathsf{N}_\mathsf{I}$ and this set is non-empty;

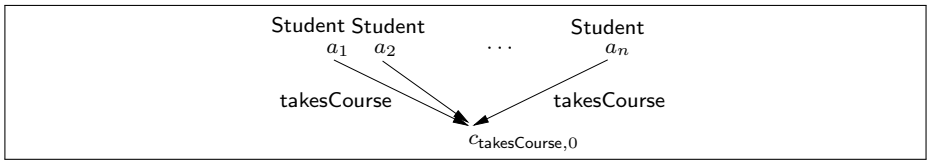


Fig. 3. Canonical model $\mathcal{I}_\mathcal{K}$ for Example 6.

– the above set is empty and ρ contains exactly one term (actually a variable).

The filtering procedure first checks whether all answer variables are mapped to elements of $\mathsf{Ind}(\mathcal{A})$, i.e., individuals of the original ABox $\mathcal{A}$. If this is not the case, it immediately returns "false". Then the procedure iterates through all root configurations ρ. For each ρ, it constructs a sequence $S^0_\rho, S^1_\rho, \dots$ of relations $S^i_\rho \subseteq \mathsf{term}(q) \times \mathsf{Paths}$ as follows:

– S^0_ρ contains all pairs $(t, \pi(t))$ with $t \in \rho$;
– S^{i+1}_ρ is S^i_ρ extended with the following pairs:
  (a) $(t, \sigma\pi(t))$ for all $R(s,t) \in q$ with $(s,\sigma) \in S^i_\rho$ and $\pi(s) \rightsquigarrow_R \pi(t)$;
  (b) $(t, \sigma\pi(t))$ for all $R(s,t) \in q$ with $(s, \sigma\pi(t)\pi(s)) \in S^i_\rho$ and $\pi(t) \rightsquigarrow_{R^-} \pi(s)$.

The computation stops as soon as the sequence stabilizes or S^i_ρ becomes non-functional which happens after at most $|\mathsf{term}(q)|$ iterations. The procedure returns "true" if the final S^i_ρ is a function with domain $\mathsf{term}(q)$ for some root configuration ρ, and "false" otherwise.

Example 6. Consider the TBox (1)-(2) from Example 1, the query

$$q_3(x,y) = \exists z\, \mathsf{Student}(x) \wedge \mathsf{Student}(y) \wedge \mathsf{takesCourse}(x,z) \wedge \mathsf{takesCourse}(y,z), \quad (9)$$

and the ABox

$$\{\mathsf{Student}(a_1), \dots, \mathsf{Student}(a_n)\}. \quad (10)$$

The canonical model $\mathcal{I}_\mathcal{K}$ is shown in Figure 3. Suppose the filter gets as input the match $\pi = \{x \mapsto a_1, y \mapsto a_2, z \mapsto c_{\mathsf{takesCourse},0}\}$. There is only one possible root configuration for π, which is $\rho = \{x, y\}$. The procedure computes

$$S_\rho = \{(x, a_1), (y, a_2), (z, a_1 c_{\mathsf{takesCourse},0}), (z, a_2 c_{\mathsf{takesCourse},0})\}$$

which is not a function; thus, the match is identified as spurious and "false" is returned.

Example 7. Consider the ABox $\{\mathsf{Faculty}(a)\}$, TBox (3)-(5), and query q_2 from Example 3. To make things a bit more interesting, assume that x is a quantified variable in q_2 rather than an answer variable. Recall that the canonical model $\mathcal{I}_\mathcal{K}$ is shown in Figure 1, modulo the names of fresh individuals. Given the match $\pi = \{x \mapsto c, y \mapsto b, z \mapsto d\}$ and considering the root configuration $\rho = \{x\}$, the procedure computes

$$S_\rho = \{(x, c), (y, cb), (z, cbd), (x, cbdc)\}$$

and stops because of non-functionality. For the other root configurations $\rho = \{y\}$ and $\rho = \{z\}$, the procedure fails in a similar way and thus returns "false".

Similar to the "tree witnesses" from [8], the filtering procedure follows a simple idea for reproducing the input match π in $\mathcal{I}_\mathcal{K}$ as a match τ in $\mathcal{U}_\mathcal{K}$: when we have already decided that $\tau(x) = \sigma \notin \mathsf{Ind}(\mathcal{A})$ and $R(x, y) \in q$, then there is a *uniquely determined* individual σ' to which y can be matched. This follows from requiring $\pi(y) = \mathsf{tail}(\tau(y))$ and the following property of $\mathcal{U}_\mathcal{K}$:

$$\text{if } (\sigma, \sigma') \in R^{\mathcal{U}_\mathcal{K}} \text{ and } (\sigma, \sigma'') \in R^{\mathcal{U}_\mathcal{K}} \text{ with } \sigma' \neq \sigma'', \text{ then } \mathsf{tail}(\sigma') \neq \mathsf{tail}(\sigma'').$$

In fact, it is this determinism of matches that is made explicit by Conditions (a) and (b) of the filtering procedure. Note that, without introducing two individual names $c_{R,0}$ and $c_{R,1}$ whenever R is involved in a loop, the above crucial property of $\mathcal{U}_\mathcal{K}$ fails. In fact, we do not know whether polytime filtering is possible based on the variation of the canonical model where all individuals $c_{R,0}$ and $c_{R,1}$ are identified. The problem is illustrated by the following example.

Example 8. Consider the ABox $\{\mathsf{Employee}(a)\}$ *and TBox (6)-(8) from Example 5 and the CQ*

$$q_4(x) = \exists y, z, u\, \mathsf{w}(x, y) \land \mathsf{p}(y, z) \land \mathsf{i}(u, z).$$

Let $\pi = \{x \mapsto a, y \mapsto c_{\mathsf{w},0}, z \mapsto c_{\mathsf{p},0}, u \mapsto c_{\mathsf{w},1}\}$*. The only root configuration is* $\rho = \{x\}$*. During the first two iterations, the filtering procedure produces*

$$S^2_\rho = \{(x, a), (y, ac_{\mathsf{w},0}), (z, ac_{\mathsf{w},0}c_{\mathsf{p},0})\}.$$

S^2_ρ *says that* z *has to be mapped to* $ac_{\mathsf{w},0}c_{\mathsf{p},0}$*. Due to the atom* $\mathsf{i}(z, u) \in q_4$ *and the two* i*-edges outgoing from* z *in* $\mathcal{U}_\mathcal{K}$*, the possible targets for* u *are* $ac_{\mathsf{w},0}$ *and* $ac_{\mathsf{w},1}$*. However, to produce a match in* $\mathcal{U}_\mathcal{K}$ *that is compatible with* π*, we can only choose a target that ends with* $\pi(u) = c_{\mathsf{w},1}$ *and obtain*

$$S^3_\rho = \{(x, a), (y, ac_{\mathsf{w},0}), (z, ac_{\mathsf{w},0}c_{\mathsf{p},0}), (z, ac_{\mathsf{w},0}c_{\mathsf{p},0}c_{\mathsf{w},1})\}$$

which is functional, showing that the match π *is not spurious. In a canonical model* $\mathcal{I}_\mathcal{K}$ *where* $c_{\mathsf{w},0}$ *and* $ac_{\mathsf{w},1}$ *are identified, there are indeed two choices for mapping of* u*. This makes it non-obvious how to find a polytime filtering procedure in this case, if one exists at all.*

We now analyze the runtime and correctness of the filtering procedure. First note that, in Conditions (a) and (b), the filtering procedure has to check whether $\pi(s) \rightsquigarrow_R \pi(t)$ and $\pi(t) \rightsquigarrow_{R^-} \pi(s)$, respectively. As required, both conditions can be tested without access to the ABox $\mathcal{A}$. For example, in Condition (a) we have:

- if $\pi(t) \in \mathsf{Ind}(\mathcal{A})$, then $\pi(s) \rightsquigarrow_R \pi(t)$ does not hold and checking whether $\pi(t) \in \mathsf{Ind}(\mathcal{A})$ requires only to check whether or not $\pi(t)$ is of the form $c_{R,i}$;
- if $\pi(s) \in \mathsf{Ind}(\mathcal{A})$ and $\pi(t) \notin \mathsf{Ind}(\mathcal{A})$, then $\pi(s) \rightsquigarrow_R \pi(t)$ holds by the construction of $\mathcal{I}_\mathcal{K}$ since π is a match of q in $\mathcal{I}_\mathcal{K}$, and;
- if $\pi(s) \notin \mathsf{Ind}(\mathcal{A})$ and $\pi(t) \notin \mathsf{Ind}(\mathcal{A})$, then $\pi(s) \rightsquigarrow_R \pi(t)$ can be checked by using only π and $\mathcal{T}$ based on the definition of "$\rightsquigarrow_R$".

It is not hard to see that the algorithm runs in polynomial time. The runtime is quadratic in the size of q because we first have to iterate over all root configurations ρ and then need to compute S_ρ, essentially a breadth-first search

of (the graph of) q. We conjecture that iterating over all root configurations is avoidable at the cost of a less transparent filtering procedure, improving the runtime to linear in the size of q. The runtime also depends on $\mathcal{T}$ as checking the applicability of Conditions (a) and (b) involves testing consequences of the forms $\mathcal{T} \models \exists R \sqsubseteq \exists S$ and $S \sqsubseteq^*_{\mathcal{T}} R$. Since it is efficient to pre-compute all these consequences in practical cases, this amounts to a simple lookup.

The following lemma asserts correctness of the filtering procedure. It is proved in Appendix A of the full version.

Lemma 2. *Given a match π of q in $\mathcal{I}_\mathcal{K}$, the filtering procedure returns "true" iff π is not spurious.*

5 Experiments

We carry out an experimental evaluation to analyze the performance of the combined approach with filtering, based on the IBM DB2 relational database system. We use a modified version of the ontology from the Lehigh University Benchmark (LUBM) [6] and produce test ABoxes using a modified version of the LUBM data generator. We execute five queries from the LUBM suite as well as six handcrafted ones that were designed to test the proposed approach more fully. Note that LUBM was not specifically designed for evaluating OBDA with DL-Lite$_\mathcal{R}$ and in its original form is not too useful for this purpose, for reasons explained in more detail below. Our modifications of the LUBM ontology, data generator, and query set all aim at making the LUBM suite more realistic for OBDA evaluation. We believe that this material might be interesting also for future experiments and provide it online at `http://informatik.uni-bremen.de/~clu/combined/`.

5.1 Ontology, Data, and Queries

The LUBM ontology comprises 42 concept names and 25 role names and is formulated in the description logic $\mathcal{ELI}$ extended with transitive roles, role hierarchies, and datatypes. The TBox contains concept inclusions of the form $A \sqsubseteq C$, concept definitions $A \equiv C$ as abbreviations for $A \sqsubseteq C$, $C \sqsubseteq A$, and domain and range restrictions of the form $\exists R \sqsubseteq A$ and $\exists R^- \sqsubseteq A$. We converted this ontology to DL-Lite$_\mathcal{R}$ by dropping all datatypes, treating the only transitive role subOrganizationOf as a standard role, replacing concept equations $A \equiv C$ with $A \sqsubseteq C$, and breaking up conjunctions $A \sqsubseteq C_1 \sqcap C_2$ into $A \sqsubseteq C_1, A \sqsubseteq C_2$.

While the resulting TBox is formulated in DL-Lite$_\mathcal{R}$ as required, it is only moderately interesting for evaluating query answering techniques: first, there is a lack of existential restrictions $\exists R$ and $\exists R.C$ on the right-hand side of concept inclusions, which leads to extremely few fresh *anonymous* individuals being generated during the ABox completion, and consequently to very few role edges between those individuals (from now on, we call this part of the canonical model $\mathcal{I}_\mathcal{K}$ the *anonymous part*); second, the overall size of the TBox is too small to be representative for real-world ontologies. To attenuate these deficiencies while still

being able to use the LUBM data generator, we extended the DL-Lite$_\mathcal{R}$-version of LUBM in two directions:

(1) We added 26 concept inclusions, many of which have existential restrictions on the right-hand side, to generate a more interesting anonymous part of canonical models. A complete list of these CIs can be found in Appendix B of the full version of this paper.

(2) With reasonable effort, it does not seem possible to significantly increase the size of LUBM (to hundreds or thousands of concepts) while retaining a careful modeling. One particularly unrealistic aspect of LUBM and a striking difference to more comprehensive ontologies is its limited concept hierarchy, where each concept has only very few subconcepts. To alleviate this shortcoming, we added subconcepts to each of the LUBM concepts Course, Department, Professor, and Student by introducing subject areas, such as MathCourse, BioCourse, and CSCourse for courses, MathProfessor, BioProfessor for professors, etc.

We call the resulting TBox LUBM$_n^\exists$ with n indicating the number of subconcepts introduced in Point 2 above (20 by default). For example, LUBM$_{20}^\exists$ contains 106 concept names and 27 role names.

To generate ABoxes, we use the LUBM Data Generator (UBA) version 1.7, modified so as to complement our modifications to the TBox. Specifically, the original UBA generates data that is complete w.r.t. existential restrictions in the LUBM ontology: it produces ABoxes $\mathcal{A}$ such that for every assertion $A(a) \in \mathcal{A}$ and CI $A \sqsubseteq \exists R$ (and $A \sqsubseteq \exists R.B$) in LUBM$_n^\exists$, there is already an r-successor of a in $\mathcal{A}$. Our modifications introduce a controlled amount of incompleteness: the modified data generator takes a probability p as a parameter and, in selected parts of the data, drops generated role assertions with probability p. More information can be found in Appendix C of the full version. The second modification of the data generator is linked to the subconcepts introduced in Point 2 above. Whenever the original generator produces an instance a of Student, the new generator randomly chooses a value between 1 and n and generates an assertion for the i-th subject, Subj$i$Student(a); similarly for Course, Department, and Professor.

The main aim of our experiments is to show that our approach is feasible on realistic ontologies, data, and queries. Additionally, we also provide a preliminary comparison with the query rewriting approach, using the Requiem tool for producing those rewritings [10]. We use 11 queries, six of which we have handcrafted specifically for our experiments and five originating from the evaluation of Requiem presented in [10]. The latter queries are extremely simple and do in most cases neither pose a serious challenge for the filtering approach nor for pure rewriting. The former are shown in Figure 4. Note that q_3 is very similar to the query discussed in Examples 3, 4, and 7; and is designed in such a way that spurious cycles in the anonymous part of canonical models produce spurious matches that have to be filtered out. Query q_2 is essentially the query discussed in Example 6 and is designed to stress-test the filtering approach: based on the data generation scheme, it is expected to produce a very large number of spurious answers.

```
q1(x,y) <- Student(x), takesCourse(x,z), Course(z), teacherOf(y,z),
           Faculty(y), worksFor(y,u), Department(u), memberOf(x,u)
q2(x,y) <- Subj3Student(x), Subj4Student(y),
           takesCourse(x,z), takesCourse(y,z)
q3(x)   <- Faculty(x), degreeFrom(x,y), University(y),
           subOrganizationOf(z,y), Department(z), memberOf(x,z)
q4(x,y) <- Subj3Department(x), Subj4Department(y),
           Professor(z), memberOf(z,x), publicationAuthor(u,z),
           Professor(v), memberOf(v,y), publicationAuthor(u,v)
q5(x)   <- Publication(x), publicationAuthor(x,y), Professor(y),
           publicationAuthor(x,z), Student(z)
q6(x,y) <- University(x), University(y), memberOf(z,x), Student(z),
           memberOf(u,y), Professor(u), advisor(z,u)
```

Fig. 4. Queries $\mathtt{q}_1$ to $\mathtt{q}_6$.

Universities	CA	CA (compl)	RA	RA (compl)	Inds	Inds (compl)
10	373K	636K	593K	1.3M	201K	201K
25	984K	1.6M	1.5M	3.6M	528K	528K
50	1.9M	3.3M	3.1M	7.2M	1M	1M
75	3M	5.1M	4.7M	10.9M	1.6M	1.6M
100	4M	6.8M	6.3M	14.6M	2.1M	2.1M
125	5M	8.5M	7.9M	18M	2.7M	2.7M
150	6M	10.1M	9.5M	21.8M	3.2M	3.2M
200	8M	13.5M	12.6M	29M	4.3M	4.3M

Fig. 5. Number of concepts, roles, and individuals in original and completed ABoxes.

5.2 Results

We report on three experiments, in each experiment varying a different parameter: in experiment one, we vary the size of the ABox via the number of universities generated by the data generator; in experiment two, we vary the degree of incompleteness of the data; and in experiment three, we vary the number of subclasses, i.e., the parameter n of the ontology $\text{LUBM}_n^{\exists}$. All experiments were carried out on a Linux (3.2.0) machine with a 3.5Ghz quad-core processor and 8GB of RAM, using IBM DB2 version 9.7.5.

The size of the test data is detailed in Figure 5 and query execution times for the first experiment are reported in Figure 7. Here we use 5% incompleteness of the data and $n = 20$ subclasses in $\text{LUBM}_n^{\exists}$. The orange curves are for the combined approach with filtering while the blue ones indicate the pure rewriting approach for those cases in which Requiem succeeded generating a rewriting. Using the combined approach with filtering, all queries were answered within very reasonable time. To better understand the results, it is interesting to consider the number of spurious and valid answers for each query shown in Figure 6 for the 200 universities experiment. Query $\mathtt{q}_2$, designed specifically to stress-test the filter, as expected produces a huge number of spurious answers. Indeed, the

	q_1	q_2	q_3	q_4	q_5	q_6	req_1	req_2	req_3	req_4	req_5
spurious answers	2	28M	2	24K	0	0	0	22K	0	163K	0
valid answers	4.6M	0	0	0	8.3M	0	0	410K	48K	137K	0

Fig. 6. Number of answers for 200 universities, 5% incompleteness, 20 subclasses.

comparably long execution time of this query appears to be mainly due to the fact that DB2 has to handle a large number of spurious answers before the filtering takes place, and not to a poor performance of the filter itself. The execution times of q_1 and q_5 can also be explained by a large number of answers. Note that the number of filter calls is actually the sum of the numbers of answers, both spurious and valid. Also note that, in principle, it is possible to avoid an extremely large number of spurious answers in q_2 (and any other query) at the cost of slightly increasing the size of the canonical model: duplicate the anonymous part of the canonical model so that no two individuals in the original ABox 'share' an anonymous part of the canonical model. Analyzing this further in experiments is left for future work.

Experiments two and three are reported about in Figures 8 and 9. Here, we only tested the filtering approach. In both cases, we use 100 universities. In experiment two, the number of subclasses is fixed to 20 while in experiment three, the degree of incompleteness is fixed to 5%. In general, the degree of incompleteness has virtually no effect on the execution time of queries. Again, q_2 is an exception as the number of spurious answers increases dramatically from 3M for 1% incompleteness to 125M for 20% incompleteness. The number of subclasses also has essentially no effect on query execution times (in contrast to the pure query rewriting approach for which a non-trivial number subclasses can dramatically increase the size of the rewritten query). Note that the execution time of q_2 becomes shorter with an increasing number of subclasses because the number of spurious matches decreases from 6.5M for 5 subclasses to 211K for 100 subclasses: this is due to the atoms Subj3Student and Subj4Student in q_2 and the fact that the number of assertions for these two concepts decreases as the number of subject areas increases.

6 Conclusion

We have modified the combined approach to OBDA by replacing the query rewriting part with a filtering technique. This step is natural from an implementation perspective and allows to circumvent an exponential blowup of the query. Based on experiments with an improved version of the LUBM ontology, we have demonstrated the scalability of our approach.

As future work, we plan to extend the combined approach with filtering to other description logics for which, until now, it is unknown how to avoid an exponential blowup. For example, we believe that polytime filtering is possible for the extension of $\mathcal{EL}$ with transitive roles, as found in the OWL2 EL profile. It would also be interesting to better understand the impact of modifying the canonical model on query answering, both from a theoretical and from a

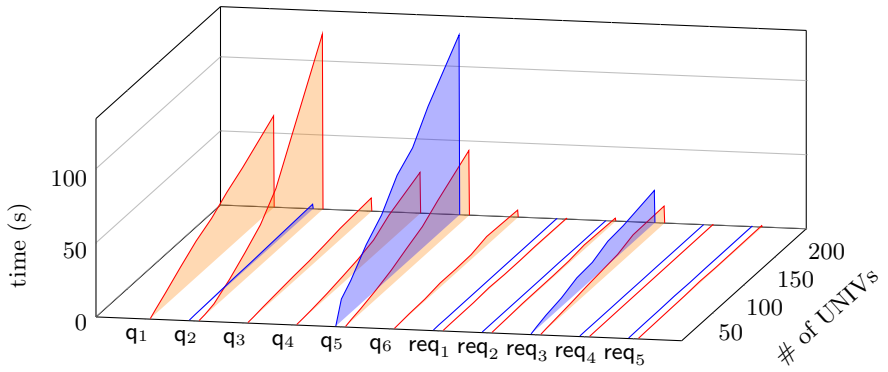


Fig. 7. Query run times for varying numbers of Universities.

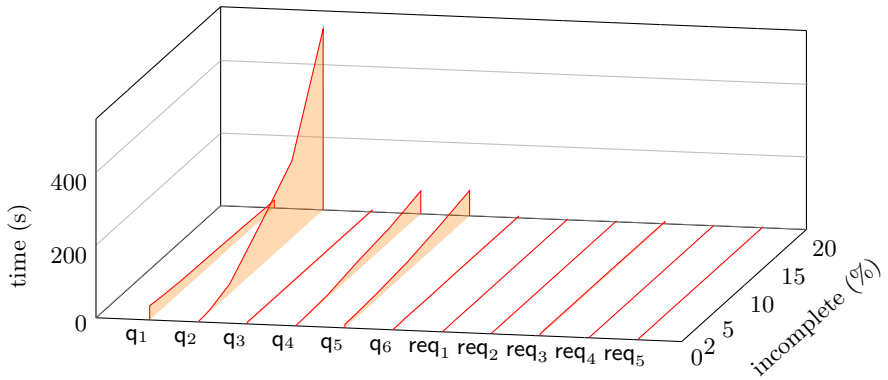


Fig. 8. Query run times for varying incompleteness (in %).

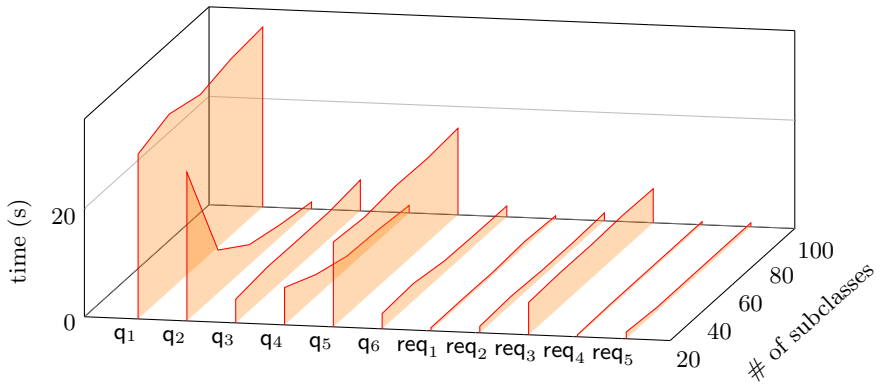


Fig. 9. Query run times for varying number of subclasses.

practical perspective. For example, we do not know whether the more natural canonical model obtained by identifying all individuals $c_{R,0}$ and $c_{R,1}$ admits polytime filtering. Moreover, as discussed above it is conceivable that versions of the canonical model that are *less* economic regarding individual reuse result in better runtime in practice.

We also plan to compare the performance of our approach more thoroughly with the performance of pure query rewriting, using other state-of-the-art query rewriting tools such as Quest [12], Presto [13], OWLgres [14], CLIPPER [4]. In this context, it is interesting to note that promising new optimization techniques have recently been developed in [11] and implemented in the Quest system. While some of them (such as the exploitation of ABox integrity constraints) aim specifically at the query rewriting approach, others (such as semantic indexing) can easily be combined with the filtering approach proposed in this paper.

References

1. Baader, F., Calvanese, D., McGuinness, D., Nardi, D., Patel-Schneider, P.F. (eds.): The Description Logic Handbook: Theory, Implementation and Applications. Cambridge University Press (2003)
2. Calì, A., Gottlob, G., Pieris, A.: New expressive languages for ontological query answering. In: AAAI (2011)
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. J. of Automated Reasoning 39(3), 385–429 (2007)
4. Eiter, T., Ortiz, M., Simkus, M., Tran, T.K., Xiao, G.: Query rewriting for Horn-$\mathcal{SHIQ}$ plus rules. In: AAAI (2012)
5. Eiter, T., Ortiz, M., Simkus, M., Tran, T.K., Xiao, G.: Towards practical query answering for Horn-$\mathcal{SHIQ}$. In: Description Logics (2012)
6. Guo, Y., Pan, Z., Heflin, J.: LUBM: A benchmark for OWL knowledge base systems. J. Web Sem. 3(2-3), 158–182 (2005)
7. Kikot, S., Kontchakov, R., Podolskii, V.V., Zakharyaschev, M.: Exponential lower bounds and separation for query rewriting. In: ICALP (2). pp. 263–274 (2012)
8. Kontchakov, R., Lutz, C., Toman, D., Wolter, F., Zakharyaschev, M.: The combined approach to query answering in DL-Lite. In: KR (2010)
9. Lutz, C., Wolter, F., Toman, D.: Conjunctive query answering in the description logic $\mathcal{EL}$ using a relational database systems. In: IJCAI. pp. 2070–2075 (2009)
10. Pérez-Urbina, H., Horrocks, I., Motik, B.: Efficient query answering for OWL 2. In: International Semantic Web Conference. pp. 489–504 (2009)
11. Rodriguez-Muro, M., Calvanese, D.: High performance query answering over DL-Lite ontologies. In: KR (2012)
12. Rodriguez-Muro, M., Calvanese, D.: Quest, an OWL 2 QL reasoner for ontology-based data access. In: OWLED (2012)
13. Rosati, R., Almatelli, A.: Improving query answering over DL-Lite ontologies. In: KR (2010)
14. Stocker, M., Smith, M.: Owlgres: A scalable OWL reasoner. In: OWLED (2008)
15. Thomazo, M., Baget, J.F., Mugnier, M.L., Rudolph, S.: A generic querying algorithm for greedy sets of existential rules. In: KR (2012)

Evaluation of Query Rewriting Approaches for OWL 2

Héctor Pérez-Urbina, Edgar Rodríguez-Díaz, Michael Grove,
George Konstantinidis, and Evren Sirin

Clark & Parsia, LLC
United States
{hector,edgar,mike,george,evren}@clarkparsia.com

Abstract. Query answering over ontologies is a crucial feature in contexts such as ontology-based data access and semantic information integration. There is considerable research interest in using *query rewriting* for efficient and scalable query answering: instead of evaluating a given query over the ontology with the (potentially very large) data directly, one rewrites the query with respect to the relevant knowledge in the ontology, and delegates the evaluation of the computed rewriting to a (possibly deductive) database system where the data resides. In this paper we examine the performance and scalability of producing unions of conjunctive queries versus datalog queries as rewritings. We present an empirical comparison between two representative approaches that consider very expressive ontology languages.

1 Introduction

The use of ontologies for query answering allows for the extraction of both explicit and *implicit* knowledge from the underlying data. Query answering over ontologies is a crucial problem in contexts such as ontology-based data access [12] and semantic information integration [10].

The main query language considered in the literature is that of *conjunctive queries* (which captures the core of SPARQL queries). In contrast, several ontology languages of various levels of expressivity have been considered. Query answering for very expressive languages such as OWL 2 DL is known to be intractable [7]. Fortunately, three *profiles* of OWL 2 with good computational properties have been identified: QL, RL, and EL.[1] In fact, query answering over QL ontologies is known to be only as hard as evaluating SQL queries over a relational database [3].[2]

Query answering in QL can be performed via *query rewriting* in two steps: first, the query and the *terminological* part of the ontology (i.e., the schema or TBox) are transformed into a so-called *rewriting*; and then the rewriting is evaluated over the *assertional* part of the ontology (i.e., the data or ABox) only.

[1] http://www.w3.org/TR/owl2-profiles/
[2] With respect to data complexity.

In this case, the rewriting is an expanded version of the original query in the form of a union of conjunctive queries (UCQ). Therefore, reasoners implementing query rewriting not only avoid keeping potentially very large ABoxes in memory, but may delegate evaluation of the rewriting to off-the-shelf, highly optimized RDBMSs.

Various UCQ-producing rewriting algorithms have been devised for (variants of) QL [3, 11, 15, 5, 8]; alas, the size of the rewritings has been shown to be exponential with respect to the size of the original query and the TBox [3]. In practice, this means that the computed rewriting might contain hundreds or thousands of queries, rendering it too big to evaluate efficiently (or at all) over existing technology. In order to address this problem, alternative approaches have been devised [16, 6] in which, instead of producing a potentially large UCQ, the original query and the TBox are rewritten into a more succinct *datalog query* (DQ). Datalog queries, however, are harder to evaluate than UCQs [1], which suggests there is a trade-off between the size of the rewriting and the complexity of its evaluation.

In this paper, we consider the advantages and disadvantages of producing DQs over UCQs in terms of scalability of query answering. We begin by presenting the query rewriting approach in more detail in Section 2. We then present a general overview of existing approaches (of the two kinds) in Section 3. The main contribution of the paper is an empirical evaluation in which we compare the (DQ-producing) approach of Eiter et al. [6] against Blackout—a highly optimized version of the (UCQ-producing) approach of Pérez-Urbina et al. [11]. We present Blackout in more detail in Section 4. The results of our evaluation are presented in Section 5. We present our conclusions in Section 6, and discuss our plans for future work in Section 7.

2 Query Answering via Rewriting

In this section, we introduce the notion of query rewriting informally by means of an example; we then discuss the advantages and disadvantages of the approach, and we finish with relevant formal definitions.

Query rewriting is a technique that can be used to solve the problem of query answering over ontologies—that is, given a *conjunctive* query and an ontology, composed of a TBox and an ABox, compute the set of *certain answers* of the query with respect to the ontology. The main idea behind query rewriting is to transform the given query and *TBox* into an expanded query that can be later evaluated over the ABox *only*. Intuitively, the expanded query contains all the relevant information captured in the TBox, making the latter unnecessary for query evaluation.

Example 1. Suppose we have an ontology $\mathcal{O} = \langle \mathcal{T}, \mathcal{A} \rangle$ that talks about universities, students, professors, and so on. The TBox $\mathcal{T}$ contains the following axioms

(shown in Manchester syntax[3]):

`Class:` Teacher `SubClassOf:` teaches `some Thing` (1)

`Class:` Professor `SubClassOf:` Teacher (2)

`ObjectProperty:` hasTutor `Range:` Professor (3)

where axiom (1) states that teachers teach at least someone, axiom (2) states that professors are teachers, and axiom (3) states that all tutors are professors.

Suppose that we want to retrieve the list of individuals who teach according to $\mathcal{O}$ using the query Q (shown in datalog syntax [1]):

$$Q(x) \leftarrow \text{teaches}(x, y) \tag{4}$$

Before considering the data in $\mathcal{A}$, we can rewrite the query with respect to the TBox—that is, expand Q with the relevant knowledge in $\mathcal{T}$. According to the meaning of axioms (1)–(3), we conclude that all teachers, professors, and tutors teach; therefore, we expand Q with:

$$Q(x) \leftarrow \text{Teacher}(x) \tag{5}$$

$$Q(x) \leftarrow \text{Professor}(x) \tag{6}$$

$$Q(y) \leftarrow \text{hasTutor}(x, y) \tag{7}$$

We can now evaluate the resulting *union* of queries (4)–(7) over $\mathcal{A}$ without further consideration of $\mathcal{T}$.

The query rewriting approach has important advantages over the 'direct' query answering approach implemented in reasoners like Pellet,[4] HermiT,[5] or FaCT++.[6] Since the query and the TBox only are considered, reasoners implementing query rewriting need not maintain potentially large ABoxes in memory, a crucial feature for some applications in terms of scalability. Once the query has been rewritten, its evaluation can be delegated to existing highly optimized (deductive) database systems. Moreover, as the rewriting is independent from the ABox, one does not need to recompute it every time the data changes, but only when the TBox does. This is important in many application domains where data tends to change much more often than the schema.

The specification of OWL 2 includes the definition of various fragments or profiles that have been tailored with specific use cases in mind. In particular, the QL profile was designed so as to benefit from the advantages of query rewriting, both in terms of scalability and performance. It has been shown that queries posed over OWL 2 QL TBoxes can be rewritten into unions of conjunctive queries (UCQs) [3]. Producing UCQs is particularly desirable as their evaluation can be delegated to RDBMSs [12].

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

[4] `http://clarkparsia.com/pellet/`

[5] `http://hermit-reasoner.com/`

[6] `http://owl.man.ac.uk/factplusplus/`

Query rewriting is, alas, not a silver bullet. Depending of the nature of the expanded query, it might turn out to be too big and/or complex to evaluate efficiently (or at all). In particular, for instance, the size of a UCQ computed from a query and an OWL 2 QL TBox has been shown to be worst-case exponential (with respect to the size of the inputs) [3]. This means that we might compute a UCQ containing hundreds or thousands of queries, which would compromise the feasibility of its evaluation. Regarding complexity, once we consider more expressive fragments of OWL than QL, the resulting expanded query may not longer be a UCQ. Depending on how far we go with respect to ontology expressivity, we might need to produce *recursive* or even *disjunctive* datalog queries in order to ensure the soundness and completeness of the results. As one might expect, these types of query are harder to evaluate than UCQs. In such cases, one needs a more sophisticated machinery, such as that implemented in a deductive database system, for query evaluation.

An alternative approach to query rewriting is a technique based on forward chaining [4], known as *materialization*. This approach consists of expanding the ABox, instead of the query, with respect to the TBox, to effectively make all the implicit knowledge explicit. This approach might be preferable to query rewriting in cases where there are no changes to the ontology; queries may be executed as they are, without the need to rewrite them into potentially larger, more difficult to answer ones. Materialization, however, has significant drawbacks when changes to the data are frequent. This is due to the fact that materialized inferences need to be maintained in order for query answering to remain sound and complete. Thus, materialization may not be efficient in domains where data changes frequently. In contrast, query rewritings are independent of the ABox; therefore, one does not need to recompute them every time the data changes, but only when the TBox does. Materialization may also not be feasible as the expanded ABox might be prohibitively large. In contrast, query rewriting requires no modifications to the ABox.

Other alternatives include hybrid approaches where both the query *and* the ABox are expanded with respect to the TBox. The objective of these approaches is to avoid the potential exponential explosion in the size of the expanded query by using certain types of axiom to expand the ABox, while still retaining a manageable size. Unfortunately, similarly to materialization, these approaches might not be very efficient in scenarios where the data changes frequently.

We finish this section by giving a formal definition of the various notions described thus far. We use the well-known notions of constants, variables, function symbols, terms, and atoms of first-order logic [4].

Definition 1. *A* Horn clause *is an expression of the form* $H \leftarrow B_1 \wedge \cdots \wedge B_m$*, where* H *is a possibly empty atom and* $\{B_i\}$ *is a set of atoms. The atom* H *is called the* head *and the set* $\{B_i\}$ *is called the* body*. A Horn clause* C *is* safe *if all variables occurring in the head also occur in the body.*

A datalog *program* P *is a set of function-free, safe Horn clauses. The* extensional database (EDB) predicates *of* P *are those that do not occur in the head atom of any Horn clause in* P*; all other predicates are called* intensional

database (IDB) predicates. *A* datalog query *(DQ) Q is a tuple $\langle Q_P, P\rangle$, where Q_P is a* query predicate *and P is a datalog program. $Q = \langle Q_P, P\rangle$ is a* union of conjunctive queries *(UCQ) if Q_P is the only IDB predicate in P and the body of each clause in P does not contain Q_P, and Q is a* conjunctive query *(CQ) if it is a union of conjunctive queries and P contains exactly one Horn clause.*

An ontology *$\mathcal{O}$ is a tuple $\langle \mathcal{T}, \mathcal{A}\rangle$, where $\mathcal{T}$ is the terminological box or TBox, and $\mathcal{A}$ is the assertional box or ABox [2]. A tuple of constants $\vec{a}$ is an* answer *of a datalog query $Q = \langle Q_P, P\rangle$ on an ontology $\mathcal{O} = \langle \mathcal{T}, \mathcal{A}\rangle$ if and only if $\mathcal{O} \cup P \models Q_P(\vec{a})$, where P is considered to be a set of universally quantified implications with the usual first-order semantics; the set of all answers of Q on $\mathcal{O}$ is denoted by $\mathsf{ans}(Q, \mathcal{O})$.*

Given a conjunctive query Q and a TBox $\mathcal{T}$, a datalog query $Q_\mathcal{T}$ is said to be a rewriting *of Q w.r.t. $\mathcal{T}$ if and only if $\mathsf{ans}(Q, \mathcal{T} \cup \mathcal{A}) = \mathsf{ans}(Q_\mathcal{T}, \mathcal{A})$ for every $\mathcal{A}$.*

3 State of the Art

The success of query rewriting depends on algorithms that produce manageable rewritings, both in terms of size and complexity. Since the seminal work of Calvanese et al. on DL-Lite—the Description Logic that provides the logical underpinning for OWL 2 QL—many rewriting algorithms aimed at efficient query answering via query rewriting have been proposed, most of which have been implemented in prototypes or commercial systems.

We limit ourselves to approaches implementing query rewriting as defined in Definition 1; that is, approaches where only the query gets expanded and the ABox is considered to be independent. Materialization-based and hybrid approaches—such as those by Kontchakov et al. [9], and Rodríguez-Muro and Calvanese [14]—are out of the scope of this paper.

Table 1. Overview of existing rewriting algorithms

	DL-Lite	Beyond DL-Lite
UCQ	PerfectRef Requiem Prexto Rapid Nyaya	Nyaya (Datalog$^{\pm}$)
DQ	Presto Clipper	Requiem ($\mathcal{ELHIO}^{\neg}$) Clipper (Horn-$\mathcal{SHIQ}$)

Notable approaches include that of Calvanese et al. (PerfectRef) [3], Pérez-Urbina et al. (Requiem) [11], Chortaras et al. (Rapid) [5], Rosati and Almatelli

(Presto) [16], Rosati (Prexto) [15], Gottlob et al. (Nyaya) [8], and Eiter et al. (Clipper) [6]. In Table 1 these approaches are classified by the type of rewritings they produce (either UCQ or DQ) and by the Description Logic (DL) they support. Unsurprisingly, most approaches have been proposed for DL-Lite; however, note that there are a few algorithms that support DL-Lite as well as more expressive logics (shown in parentheses). Among the approaches that go beyond DL-Lite, Requiem is the only one that produces UCQs for DL-Lite and DQs for more expressive logics. In fact, Requiem will only produce DQs when the rewriting *has* to be recursive in order to ensure correct results; therefore, in many cases Requiem will produce UCQs even for logics more expressive than DL-Lite.

Most of the papers cited previously include an empirical evaluation of the approach with respect to others. We have summarized these results in Table 2, which shows the comparison of the approaches with respect to the rewritings size (number of clauses), their structural complexity, the time it takes to compute them, and the time it takes to *evaluate* them over some ABox.

Table 2. Comparison of existing rewriting algorithms

Size	[Clipper $\approx$ Presto], Prexto $<$ [PerfectRef = Requiem = Rapid = Nyaya]
Complexity	[PerfectRef $\approx$ Requiem $\approx$ Prexto $\approx$ Rapid $\approx$ Nyaya] $<$ [Clipper $\approx$ Presto]
Time	Rapid, Nyaya, [Clipper $\approx$ Presto] $<$ Requiem $<$ PerfectRef
Eval time	[Requiem $\approx$ Clipper $\approx$ Presto] $<$ PerfectRef

Note 1. All these comparison were made over DL-Lite ontologies. The relationship between approaches separated with commas is not discussed in the literature.

The approaches that produce DQs (see Table 1) produce smaller rewritings than their UCQ-producing counterparts, with the exception of Prexto. This is due to the fact that UCQs are larger than semantically equivalent (non UCQ) DQs (conjunctive normal form versus disjunctive normal form). Prexto stands apart because, unlike other UCQ-producing approaches, it implements an optimization that considers the ABox to reduce the size of the rewritings. Regarding complexity, we see that UCQ-producing approaches do better than DQ-producing ones. With respect to time, as it is related to size, it is not surprising that producing DQs is faster than producing UCQs; it is important to mention, however, that among those algorithms that produce UCQs, some approaches are much more efficient than others (hours versus seconds). Finally, with respect to evaluation time, we see that Presto outperforms PerfectRef, and, interestingly, that Requiem, Clipper, and Presto perform similarly, in spite of the fact that Requiem's rewritings are larger.

The comparison between Requiem, Clipper, and Presto regarding evaluation times was carried out by evaluating the computed rewritings of each system using DLV.[7] Using such a system was necessary as both Presto and Clipper require a deductive database of this type for query evaluation even for DL-Lite ontologies; note, however, that Requiem produces UCQs in this scenario. Therefore, these results suggest that evaluating semantically equivalent UCQs and DQs in DLV takes approximately the same time, so there is no substantial gain in evaluation time by producing (smaller but more complex) DQs versus UCQs. It would be interesting to see, however, whether UCQs can be evaluated more efficiently in an RDBMS or an RDF database.

Another important aspect to consider is the time it takes to compute the rewritings. According to Table 2, both Clipper and Presto outperform Requiem with respect to this metric (in fact, Requiem is outperformed by every algorithm except PerfectRef). It would be interesting to see whether Requiem can be made faster by enhancing it with the various optimization techniques used in the other, more efficient approaches.

In order to address these two questions, we present an empirical evaluation of Clipper and Blackout—an optimized version of Requiem—in Section 5. We chose these two approaches as they are the ones that support the most expressive logic within their respective rewriting type (UCQ vs DQ) categories. The optimizations implemented in Blackout are described in Section 4.

4 Blackout Optimizations

In this section we describe Blackout, a highly optimized version of Requiem. Blackout is part of the state-of-the-art triplestore Stardog.[8] Besides careful software engineering for efficiency, Blackout improves Requiem with two core optimizations.

First, Blackout implements an eager query containment optimization, as opposed to Requiem's lazy approach that computes query containment as the last step. As observed in most of the papers referenced in Section 3, this is one of Requiem's major drawbacks as the final containment step could take a very long time. Eager containment prunes redundant queries earlier in the rewriting process; thus, it prevents additional rewritings from being generated from these redundant queries, which themselves would be redundant. Even though this does not ultimately change the number of rewritings, it does significantly minimize the time spent on query containment checks. Thus, Blackout does not waste time generating redundant queries that will eventually be pruned, and it reduces the total number of containment checks performed.

Second, Blackout implements an optimization technique known as *data oracle*. This optimization is related to the so-called extensional constraints technique presented in [13]. If a derived query contains an atom which is empty with respect to a given ABox, the query is discarded as it would obviously produce empty

[7] `http://www.dlvsystem.com/`
[8] `http://stardog.com/`

results when evaluated. For instance, consider the rewriting obtained in Example 1, if we knew via the *data oracle* that the class Professor had no instances in $\mathcal{A}$, then there would be no need to include query (6) in the final rewriting.

The effectiveness of this optimization lies in the fact that even when a TBox might contain a large number of classes and properties, the assertions in the ABox typically use a much smaller number of classes and properties. This is frequently the case for deep class hierarchies where asserted types use leaf classes of the hierarchy, instead of more general classes higher in the hierarchy. For this reason, querying for a generic class might produce many rewritings because of the class hierarchy, but we might not need to execute all of those rewritings depending on the specific ABox at hand.

In Section 2 we pointed out that one advantage of the query rewriting approach is that it is independent of the ABox, whereas clearly the data oracle optimization introduces a dependency. This dependency, however, is a very weak one and requires the data oracle to only check the existence of an atom, a class or a property, in the data. Stardog, like other RDF databases, maintains special index structures that make this very efficient. Therefore, the query rewriting component can still be loosely-coupled from the storage system and does not need to maintain special in-memory data structures for this optimization. Moreover, as will be discussed further, the data oracle implementation can be crucial to the success of query rewriting in practice.

5 Evaluation

In this section we present an empirical evaluation of Clipper and Blackout. Our evaluation is based on the LUBM benchmark—a well-known standard that provides customizable data generation capabilities.[9] We first examined the performance of the two approaches with respect to size/complexity of the rewritings, including the time it took to compute them; and then, we looked into how these rewritings perform when evaluated.

All experiments were performed on Ubuntu 3.0.0 with a 3.2Ghz AMD Phenom processor, 8GB of RAM running Java $1.6.0_3 3$ with 8GB allocated to the JVM for each run. We recorded the average of 10 runs after 5 warmups for each experiment.

5.1 Computing Rewritings

The first part of the evaluation consisted of rewriting the 14 LUBM queries with respect to three TBoxes: $\mathcal{T}_{\text{QL}}$, $\mathcal{T}_{\text{RL}}$, and $\mathcal{T}_{\text{EL}}$, which correspond to QL, RL, and EL versions of the LUBM TBox, respectively. Table 3 summarizes our results. On the left-hand-side, we show the time in milliseconds that each system took to produce the rewritings for the 14 queries, whereas on the right-hand-side we show the number of clauses that each system produced overall.

[9] `http://swat.cse.lehigh.edu/projects/lubm/`

Table 3. Computation of Rewritings

	Time (ms)			Size (clauses)		
	$\mathcal{T}_{\text{QL}}$	$\mathcal{T}_{\text{RL}}$	$\mathcal{T}_{\text{EL}}$	$\mathcal{T}_{\text{QL}}$	$\mathcal{T}_{\text{RL}}$	$\mathcal{T}_{\text{EL}}$
Clipper	8.4	4.3	14.0	145	138	866
Blackout	25.0	21.0	108.8	20	21	102

As can be seen, Blackout is generally slower than Clipper, but it produces smaller rewritings. We believe the rewritings with few clauses produced by Blackout are the result of the data oracle optimization. In order to verify the benefits of this optimization, we ran Blackout without it and obtained 66 clauses for $\mathcal{T}_{\text{QL}}$, 65 clauses for $\mathcal{T}_{\text{RL}}$, and 253 clauses for $\mathcal{T}_{\text{EL}}$. The most significant gain was in query 5 over $\mathcal{T}_{\text{EL}}$, in which Blackout produced a rewriting containing 51 clauses, whereas Blackout with no data oracle produced 117. These results suggest that the data oracle optimization, when applicable, may have a big impact.

As can be seen in Table 4, Clipper produced DQs exclusively for all the queries and profiles, whereas Blackout produced UCQs most of the time, even for RL and EL. These results suggest that it is not always necessary to produce DQs as rewritings even for RL or EL ontologies. The type of the rewriting impacts evaluation time as UCQs are structurally less complex than DQs, and might be easier to evaluate, depending on their size.

Table 4. Rewritings Type

	$\mathcal{T}_{\text{QL}}$	$\mathcal{T}_{\text{RL}}$	$\mathcal{T}_{\text{EL}}$
Clipper	DQ (100%)	DQ (100%)	DQ (100%)
Blackout	UCQ (100%)	UCQ (86%)	UCQ (79%)

5.2 Evaluating Rewritings

The second part of our evaluation consisted of evaluating the rewritings produced by the two approaches. We used DLV—a state-of-the-art deductive database system (datalog engine) maintained by DLVSYSTEM s.r.l.—to evaluate Clipper and Blackout rewritings, and we used Stardog to evaluate Blackout rewritings only.[10] We considered four ABoxes of increasing size: $\mathcal{A}_1$, $\mathcal{A}_{10}$, $\mathcal{A}_{100}$, and $\mathcal{A}_{1000}$, which contain approximately 138K, 1.38M, 13.8M, and 138M triples, respectively.[11]

[10] Note that Stardog cannot presently evaluate DQs.

[11] Since the current implementation of Clipper does not support data property assertions, we ignored this type of assertion in our tests.

Tables 5 and 6 summarize our results. Table 5 shows the time in milliseconds that it took DLV to evaluate the rewritings produced by Clipper and Blackout over the various ABoxes.

Table 5. DLV Evaluation Performance (ms)

	Clipper			Blackout		
	$\mathcal{T}_{\text{QL}}$	$\mathcal{T}_{\text{RL}}$	$\mathcal{T}_{\text{EL}}$	$\mathcal{T}_{\text{QL}}$	$\mathcal{T}_{\text{RL}}$	$\mathcal{T}_{\text{EL}}$
$\mathcal{A}_1$	23.3	19.9	24.0	15.2	15.5	17.4
$\mathcal{A}_{10}$	20.8	28.3	30.2	16.3	18.5	15.6
$\mathcal{A}_{100}$	-	-	-	-	-	-
$\mathcal{A}_{1000}$	-	-	-	-	-	-

As can be seen, DLV was able to execute Blackout rewritings faster than those of Clipper, which is not surprising as the former are smaller and less complex than the latter. Unfortunately, DLV was unable to execute the rewritings over $\mathcal{A}_{100}$ and $\mathcal{A}_{1000}$ due to lack of memory since it maintains all the clauses in memory. Clearly, in order to be able to evaluate rewritings and queries in general over large ABoxes, we would need a system that makes use of secondary storage.

Table 6. Stardog Evaluation Performance (ms)

	$\mathcal{T}_{\text{QL}}$	$\mathcal{T}_{\text{RL}}$	$\mathcal{T}_{\text{EL}}$
$\mathcal{A}_1$	1.3	1.0	0.9
$\mathcal{A}_{10}$	13.8	14.4	14.1
$\mathcal{A}_{100}$	133.5	137.2	133.2
$\mathcal{A}_{1000}$	334.0	1036.0	69558.0

Table 6 shows the time in milliseconds that it took Stardog to evaluate the rewritings produced by Blackout over the various ABoxes. Stardog allows the creation of in-memory and disk databases. The experiments over $\mathcal{A}_1$, $\mathcal{A}_{10}$, and $\mathcal{A}_{100}$ were carried out using in-memory databases, whereas those over $\mathcal{A}_{1000}$ were performed over a disk database. As can be seen, Stardog outperformed DLV both with respect to Clipper and Blackout rewritings regarding $\mathcal{A}_1$ and $\mathcal{A}_{10}$. Moreover, it was able to scale to $\mathcal{A}_{100}$ and $\mathcal{A}_{1000}$.

5.3 Computation and Evaluation

In this section we summarize the results from previous sections with respect to Blackout and Stardog. Table 7 shows the overall performance of Stardog in

milliseconds counting the time it took Blackout to compute the rewritings and the time it took Stardog to evaluate them. It also shows the percentage of time that was spent on evaluating the produced rewritings.

Table 7. Blackout/Stardog Overall Performance

	$\mathcal{T}_{\text{QL}}$		$\mathcal{T}_{\text{RL}}$		$\mathcal{T}_{\text{EL}}$	
	Eval	Total (ms)	Eval	Total (ms)	Eval	Total (ms)
$\mathcal{A}_1$	4.94%	26.3	4.54%	22.0	0.82%	109.7
$\mathcal{A}_{10}$	35.57%	38.8	40.64%	35.4	11.48%	122.9
$\mathcal{A}_{100}$	84.23%	158.5	86.71%	158.2	55.05%	242.0
$\mathcal{A}_{1000}$	93.04%	359.0	98.01%	1057.0	99.84%	69666.8

Our results show that the larger the ABox, the longer time is spent on evaluating the rewritings. Therefore, we believe it is important to produce the simplest and smallest rewritings possible, even if this means spending a bit more time on the rewriting phase.

6 Conclusions

In this section we present a summary of our results.

DQ-producing approaches, such as Clipper, do not necessarily produce smaller rewritings than UCQ-producing approaches as formerly thought. Taking into consideration the underlying data might result in optimizations, such as the data oracle optimization, that can significantly reduce the size of the produced UCQs. This type of optimization should apply to DQs as well; it would be interesting to see to what extent it does and the impact it has.

It is not necessary to produce DQs for RL and EL in many cases. This is important as users can benefit from several of the languages features that are not included in QL without needing a datalog engine for query evaluation. As shown in this paper, the size of UCQs can be reduced and, importantly, UCQs are amenable to straightforward parallel evaluation. Therefore, we believe that one should only have to deal with DQs, and datalog engines, when necessary.

Evaluating the rewritings dominates the overall query answering time at large scales. Therefore, even though it may not be beneficial at small scales, it is worth investing time on optimizing the computation of rewritings in order to produce smaller and simpler rewritings that can be evaluated more efficiently.

7 Future Work

We are currently working on adding datalog evaluation to Stardog so that it can correctly evaluate Blackout rewritings even when they are DQs. Our ongoing

implementation is based on the well-known algorithm Query-SubQuery [1]. Once it is ready, it will be interesting to compare its performance with that of DLV and other state-of-the-art datalog engines (e.g., IRIS[12]).

Additionally, we plan to work on the parallelization of UCQ evaluation within Stardog. Currently, Stardog executes the results of the query rewriting as a single query; it takes the UCQ produced by Blackout and creates a single query by unioning each of the queries. However, the queries composing the UCQ tend to be relatively small, simple, and easy to evaluate. Crucially, they are also independent: the results of one conjunct are not needed to produce the results of another. This lends itself very nicely to evaluation of each query in parallel, which can be done by taking advantage of existing architecture within Stardog.

We are also working on the implementation of SPARQL 1.1 with Stardog. There are new features in SPARQL 1.1 such as sub-queries and property paths that have an interesting overlap with features provided, or planned, within Stardog and Blackout. First, we will explore what the performance implications are for rewriting sub-queries in SPARQL 1.1, and if there are advantages the QSQ approach can provide during evaluation, or even if rewriting sub-queries is a feasible design. Additionally, with some OWL language features, such as transitivity, now available in SPARQL, we will determine whether Blackout can be used to handle queries utilizing these features.

References

1. S. Abiteboul, R. Hull, and V. Vianu. *Foundations of Databases*. Addison-Wesley, 1995.
2. F. Baader and W. Nutt. *Basic Description Logics*, chapter 2, pages 47–100. Cambridge University Press, 2003.
3. D. Calvanese, G. De Giacomo, D. Lembo, M. Lenzerini, and R. Rosati. Tractable Reasoning and Efficient Query Answering in Description Logics: The DL-Lite Family. *Journal of Automated Reasoning*, 9:385–429, 2007.
4. C.-L. Chang and R. C.-T. Lee. *Symbolic Logic and Mechanical Theorem Proving*. Academic Press, Inc., Orlando, FL, USA, 1997.
5. A. Chortaras, D. Trivela, and G. B. Stamou. Optimized Query Rewriting for OWL 2 QL. In N. Bjørner and V. Sofronie-Stokkermans, editors, *CADE*, volume 6803 of *Lecture Notes in Computer Science*, pages 192–206. Springer, 2011.
6. T. Eiter, M. Ortiz, M. Simkus, T.-K. Tran, and G. Xiao. Towards Practical Query Answering for Horn-SHIQ. In Y. Kazakov, D. Lembo, and F. Wolter, editors, *Description Logics*, volume 846 of *CEUR Workshop Proceedings*. CEUR-WS.org, 2012.
7. B. Glimm, I. Horrocks, C. Lutz, and U. Sattler. Conjunctive Query Answering for the Description Logic SHIQ. *CoRR*, abs/1111.0049, 2011.
8. G. Gottlob, G. Orsi, and A. Pieris. Ontological Queries: Rewriting and Optimization. In S. Abiteboul, K. Böhm, C. Koch, and K.-L. Tan, editors, *ICDE*, pages 2–13. IEEE Computer Society, 2011.

[12] `http://www.iris-reasoner.org/`

9. R. Kontchakov, C. Lutz, D. Toman, F. Wolter, and M. Zakharyaschev. The combined approach to query answering in dl-lite. In F. Lin and U. Sattler, editors, *Proceedings of the 12th International Conference on Principles of Knowledge Representation and Reasoning (KR2010)*. AAAI Press, 2010.
10. M. Lenzerini. Data Integration: a theoretical perspective. In *PODS '02: Proceedings of the twenty-first ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*, pages 233–246, New York, NY, USA, 2002. ACM Press.
11. H. Pérez-Urbina, B. Motik, and I. Horrocks. Tractable Query Answering and Rewriting under Description Logic Constraints. *J. Applied Logic*, 8(2):186–209, 2010.
12. A. Poggi, D. Lembo, D. Calvanese, G. De Giacomo, M. Lenzerini, and R. Rosati. Linking Data to Ontologies. *J. on Data Semantics*, X:133–173, 2008.
13. M. Rodriguez-Muro and D. Calvanese. Dependencies: Making Ontology Based Data Access Work. In P. Barceló and V. Tannen, editors, *AMW*, volume 749 of *CEUR Workshop Proceedings*. CEUR-WS.org, 2011.
14. M. Rodriguez-Muro and D. Calvanese. High performance query answering over dl-lite ontologies. In G. Brewka, T. Eiter, and S. A. McIlraith, editors, *KR*. AAAI Press, 2012.
15. R. Rosati. Prexto: Query Rewriting under Extensional Constraints in DL-Lite. In E. Simperl, P. Cimiano, A. Polleres, O. Corcho, and V. Presutti, editors, *ESWC*, volume 7295 of *Lecture Notes in Computer Science*, pages 360–374. Springer, 2012.
16. R. Rosati and A. Almatelli. Improving Query Answering over DL-Lite Ontologies. In F. Lin, U. Sattler, and M. Truszczynski, editors, *KR*. AAAI Press, 2010.

Triangle Finding: How Graph Theory can Help the Semantic Web

Edward Jimenez, Eric L. Goodman

Sandia National Laboratories, Albuquerque, NM, USA
{esjimen,elgoodm}@sandia.gov

Abstract. RDF data can be thought of as a graph where the subject and objects are vertices and the predicates joining them are edge attributes. Despite decades of research in graph theory, very little of this work has been applied to RDF data sets and it has been largely ignored by the Semantic Web research community. We present a case study of triangle finding, where existing algorithms from graph theory provide excellent complexity bounds, growing at a significantly slower rate than algorithms used within existing RDF triple stores. In order to scale to large volumes of data, the Semantic Web community should look to the many existing graph algorithms.

1 Introduction

The Semantic Web continues to evolve and grow with data sizes becoming increasingly large and unwieldy. As such, we need to utilize the most efficient and effective algorithms in order to scale to meet the growing data requirements. We believe that the graph theory body of research has much to offer in terms of formal analysis and understanding of the Semantic Web. Also, graph theory has much to offer in terms of efficient algorithms that can be employed for the Semantic Web. In particular we examine triangle finding. For several decades there have been algorithms for finding triangles that have a temporal complexity of $O(m^{3/2})$ where m is the number of edges [5]. As such, we believe it incumbent of any SPARQL engine to use these methods whenever a triangle appears as part of a query.

In Section 2 we introduce formally the notion of triangle finding and the associated notation. In Section 3 we discuss the frequency of triangles in SPARQL query benchmarks. In Section 4 we discuss how SPARQL can be used to find all the triangles in a graph. We then outline the particular triangle finding algorithm we employ in Section 5 that is $O(m^{3/2})$. Section 6 compares experimentally the triangle finding method we employ against Jena[1] and Sesame[2], two common open-source RDF/SPARQL engines. Section 7 outlines related work. We then conclude in Section 8.

[1] jena.apache.org
[2] www.openrdf.org

2 Formalisms

RDF data can be modeled as a graph, $G = (V, E)$ where V is a set of vertices and E is a set of edges connecting the vertices $v \in V$. We use n to refer to the number of vertices, $|V|$, and m as the number of edges, $|E|$. For RDF, the subjects and objects are vertices in V. The predicates can be thought of as attributes associated with each edge. We enumerate the vertices so that each has a unique id in $[1, n]$. Also, we enumerate the edges of G to be in $[1, m]$. The notation e_i refers to the i^{th} edge under the enumeration. Since RDF is described directionally, i.e. there is a subject uni-directionally related to an object, edges in the graph are also directed. We will use $source(e_i)$ to refer to the source or subject of the edge. $target(e_i)$ refers to the target or object of the edge. We call two vertices v_1, $v_2 \in V$ *adjacent* if there is an edge $e \in E$ where $(source(e) = v_1 \wedge target(e) = v_2) \vee (source(e) = v_2 \wedge target(e) = v1)$. We use $\delta(v_i)$ to denote the total degree, both incoming and outgoing, of a vertex. Below we formally define the notion of a triangle.

Definition 1. *A* triangle *in a graph G is a set of three edges, e_i, e_j, $e_k \in E$ such that the set of vertices $\hat{v} = \{v | v = source(e_i) \vee target(e_i) \vee v = source(e_j) \vee v = target(e_j) \vee v = source(e_k) \vee v = target(e_k)\}$ have the following properties:*

1. $|\hat{v}| = 3$
2. *All $v \in \hat{v}$ are adjacent to one another.*

We also refer to *triads*, which we define as pair of edges with a common vertex.

It is also useful to define the notion of triangle equality.

Definition 2. *Two triangles are considered* equal *if their edge sets are the same.*

While triangle equality may seem obvious, it is important point to consider when generating result sets via SPARQL. Without care, duplicate triangles can be generated that are not prunable with the `DISTINCT` keyword. Duplicate triangles can be produced that differ in the order in which the nodes and edge labels are presented (i.e. to which variables they are bound), and thus `DISTINCT` will have no effect on these duplicates. More of this will be discussed in Section 4.

3 Applicability to Current Benchmarks

An obvious question when optimizing for triangles in SPARQL queries is how often triangles occur in practice. We examine two popular benchmarks, LUBM [11] and SP2Bench [15]. Two out of the 14 queries in LUBM have triangles in them, namely queries 2 and 9. For brevity and to emphasize the triangle portion of the queries, we omit the prefixes and the type constraints on each of the variables.

Query 2

```
SELECT ?X, ?Y, ?Z WHERE
{?X ub:memberOf ?Z .
 ?Z ub:subOrganizationOf ?Y .
 ?X ub:undergraduateDegreeFrom ?Y}
```

Query 9

```
SELECT ?X, ?Y, ?Z WHERE
{?X ub:advisor ?Y .
 ?Y ub:teacherOf ?Z .
 ?X ub:takesCourse ?Z}
```

These two queries are also some of the more complicated and time intensive ones as reported by various vendors and researchers (e.g. AllegroGraph[3] OWLIM[4]), pointing to the need for efficient processing.

SP2Bench does not have any queries with triangles in them, but there are several queries that with a natual extension of one more constaint form a triangle. For example, Query 4 requests *all distinct pairs of article author names for authors that have published in the same journal.* A natural extension is to define a constraint between the two authors, either directly, forming a triangle, or to another common vertex. The latter case forms what is called a quadrangle, and the algorithmic approach is similar and has the same complexity as finding triangles [5]. In conclusion, there appears to be a small but significant portion of queries that contain triangle structures within them.

4 Finding Triangles with SPARQL

Expressing a SPARQL query to find all triangles in a graph is surprisingly convoluted. Figure 1 is a query that finds all unique triangles in an RDF graph, but perhaps more importantly it finds the triangles with no duplication, meaning that no two solution triangles in the result set are equal. We will refer to this query as the *Triangle-finding SPARQL query.* Note that this query only works on data involving only IRIs as the subjects and objects. According to the standard the `STR` function is not defined for blank nodes and the function also removes typing and language modifiers on literals which may cause the comparison filter to be incorrect.

Before we discuss how this query finds the triangles without duplication, we must first discuss two concepts, that of *graph isomorphism* and *graph automorphism.* Two graphs G and H are isomorphic if there is a bijection, f, mapping the vertex sets of G and H such that two vertices v_i and v_j are adjacent in G if and only if $f(v_i)$ and $f(v_j)$ are also adjacent in H. A graph automorphism is

[3] `http://www.franz.com/agraph/allegrograph/agraph_bench_lubm.lhtml`
[4] `http://www.ontotext.com/owlim/benchmark-results/lubm`

```
SELECT ?X ?Y ?Z
WHERE {
  { ?X ?a ?Y .
    ?Y ?b ?Z .
    ?Z ?c ?X
    FILTER (STR(?X) < STR(?Y))
    FILTER (STR(?Y) < STR(?Z))
  }
  UNION
  {
    ?X ?a ?Y .
    ?Y ?b ?Z .
    ?Z ?c ?X
    FILTER (STR(?Y) > STR(?Z))
    FILTER (STR(?Z) > STR(?X))
  }
  UNION
  {
    ?X ?a ?Y .
    ?Y ?b ?Z .
    ?X ?c ?Z
  }
}
```

Fig. 1. The above query finds all unique triangles and each is represented once in the result set.

when there is a isomorphism of a graph G onto itself, and generally we will be concerned with non-identity mappings.

The reason this is important can be understood from the query below:

```
SELECT ?X ?Y ?Z
WHERE {
  { ?X ?a ?Y .
    ?Y ?b ?Z .
    ?Z ?c ?X }
```

For this query, every triangle found will appear three times. For instance, a triangle with solution $\langle s_1, s_2, s_3 \rangle$ will also appear as $\langle s_2, s_3, s_1 \rangle$ and $\langle s_3, s_1, s_2 \rangle$. The reason for this is that each of the vertices s_1, s_2, and s_3 each bind to each of the three variables ?X, ?Y, and ?Z under different circumstances. All three solutions satisfy the query, but each solution is the same triangle. `DISTINCT` does not help because the bindings are different for each duplicate solution. The problem arises because the query graph represented above is automorphic. Each non-trivial automorphism is a mapping to translate from one solution to another solution using the same set of edges.

Thus we arrive at the convoluted nature of the query in Figure 1. When constructing the SPARQL query, we need to account for all possible triangles

but not generate duplicates. There are eight types of triangles, shown in Figure 2. However, all eight of these types can be collapsed down to the three unioned clauses in Figure 1. We present this formally as a proof.

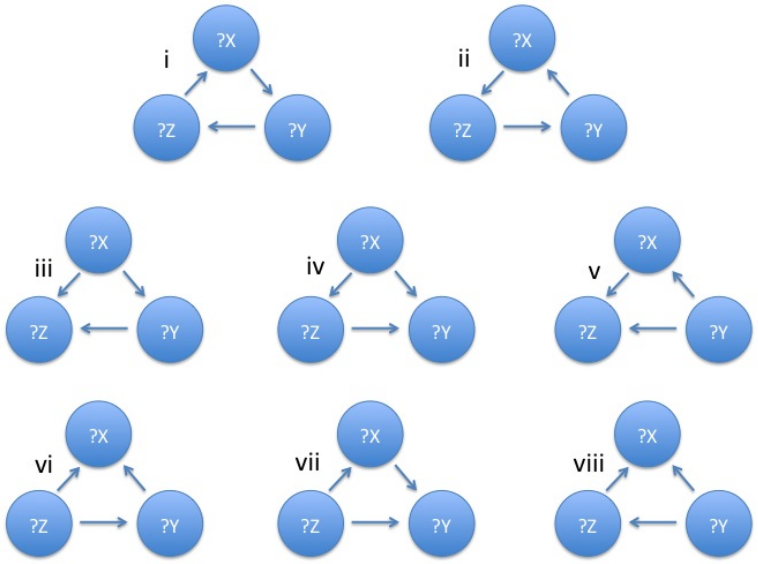


Fig. 2. The eight possible triangle patterns.

Theorem 1. *The* Triangle-finding SPARQL query *finds all triangles in the queried graph and each solution is unique.*

Proof. First consider that triangle type *iii* is isomorphic to types *iv* through *viii*. The below table outlines the functions needed to show the isomorphisms. As they are isomorphic, type *iii* is sufficient to represent all the other types *iv* - *viii*. Also, type *iii* is not automorphic, and thus will not produce any duplicate triangles. Finally, the solutions resulting from *iii* are disjoint from *i* and *ii* as there is not mapping from *iii* to either *i* or *ii*. Thus we can deal with the solution sets separately.

Mapping *iii* to ...	?X	?Y	?Z
iv	?X	?Z	?Y
v	?Y	?X	?Z
vi	?Z	?Y	?X
vii	?Z	?X	?Y
vii	?Y	?Z	?X

Concerning types i and ii, they are isomorphic under the mapping $f(?X) = ?Z$, $f(?Y) = ?Y$, and $f(?Z) = ?X$. Thus, we need only include one of the patterns in the query. We arbitrarily select type i. However, type i is automorphic (as is ii), and we must concoct a way of avoiding duplicate triples with the available SPARQL language features. We solve the issue by enforcing an ordering on the bindings. The first clause of the Triangle-finding SPARQL query enforces that the string representation of the bindings must obey $?X < ?Y < ?Z$ under an alphanumeric ordering. The second clause enforces $?Y > ?Z > ?X$. It remains to show that these two orderings will find all triangles of type i without duplication. The table below outlines all possible relations between the variables assuming no self loops.

?X ? ?Y	?Y ? ?Z	?Z ? ?X	
$<$	$<$	$<$	Empty since $?X < ?Y \wedge ?Y < ?Z \implies ?X < ?Z$, contradicting the third constraint.
$<$	$<$	$>$	Fulfilled directly by first clause
$<$	$>$	$<$	Use automorphism $f(?X) = ?Y, f(?Y) = ?Z, f(?Z) = ?X$. Fulfilled by first clause.
$<$	$>$	$>$	Fulfilled directly by second clause.
$>$	$<$	$<$	Use automorphism $f(?X) = ?Z, f(?Y) = ?X, f(?Z) = ?Y$. Fulfilled by first clause.
$>$	$<$	$>$	Use automorphism $f(?X?) = ?Z, f(?Y) = ?X, f(?Z) = ?Y)$. Fulfilled by second clause.
$>$	$>$	$<$	Use automorphism $f(?X) = ?Y, f(?Y) = ?Z, f(?Z) = ?X$. Fulfilled by second clause.
$>$	$>$	$>$	Empty since $?X > ?Y \wedge ?Y > ?Z \implies ?X > ?Z$, contradicting the third constraint.

Two of the possibilities are invalid because the constraints are contradictory. Another two possibilities directly match the first two clauses of the triangle-finding query. The remaining four cases match the two clauses through an automorphism. Thus, we may conclude that the Triangle-finding SPARQL query does in fact find all triangles in the graph with no duplicates. □

5 An $O(m^{3/2})$ Triangle Finding Algorithm

There are many triangle finding algorithms that are $O(m^{3/2})$. For the experiments we employ an algorithm presented by Cohen [6]. This algorithm has the benefit of already being described in a parallel fashion in terms of mappers and reduces of the MapReduce paradigm [8]. Also, there is a version implemented in the MultiThreaded Graph Library[5] (MTGL) [2]. However, a formal complexity analysis was not outlined in [6], so we perform that here.

[5] `https://software.sandia.gov/trac/mtgl`

Cohen's algorithm operates on what he calls a *simplified graph*. Namely, a graph in which self-loops are eliminated, directionality is ignored, and there are no duplicate edges. In our later experiments we do not allow self-loops, but we account for directionality and do allow duplicate edges. This is due to the fact that RDF is directional and multiple edges can be defined between vertices with different edge types (predicate types). We do not want to collapse all of these edge types down into one edge. We do not formally account for these different assumptions in our analysis.

Theorem 2. *Given a simplified graph G, Cohen's triangle finding algorithm is $O(m^{3/2})$.*

Proof. We assume the worst case, that G is completely connected. Relating m to n, we have

$$m = \sum_{i=1}^{n-1} i = \frac{(n-1)(n)}{2} \tag{1}$$

Cohen's algorithm is composed of two MapReduce phases. The input to the first map is a list of edges. Each edge has been previously been augmented with the degree of each vertex. If one were to include this preprocessing step in the overall complexity, it is $O(m)$ which is also in $O(m^{3/2})$.

Map 1 Map each edge to its low-degree vertex. According to our assumptions, $\delta(v_x) = \delta(v_y)\ \forall x, y \leq n$. Cohen suggests a tie-breaker based on vertex ordering; we'll use $v_1 < v_2 < \cdots < v_n$. Below is the composition of the bins. Since directionality is ignored, we'll use a canonical representation of each edge, $\langle v_i, v_j \rangle$, such that $i < j$.

Bin 1	Bin 2	$\cdots$	Bin $n-1$
$\langle v_1, v_2 \rangle$	$\langle v_2, v_3 \rangle$	$\cdots$	$\langle v_n - 1, v_n \rangle$
$\langle v_1, v_3 \rangle$	$\langle v_2, v_4 \rangle$		
$\vdots$	$\vdots$		
$\vdots$	$\langle v_2, v_n \rangle$		
$\langle v_1, v_n \rangle$			

Thus, for m edges, perform m mappings; hence, $O(m)$.

Reduce 1 Emit a record for each pair of edges in a bin (one for every open triad). For the graph G, the first Map phase created $n-1$ bins, and bin i contains

$n-i$ edges. Therefore the number of triads created is

$$\sum_{i=1}^{n-2}\binom{n-i}{2} = \sum_{i=1}^{n-2}\frac{(n-i)!}{2(n-(i+2))!} \tag{2}$$

$$= \frac{1}{2}\sum_{i=1}^{n-2}(n-i)(n-(i+1)) \tag{3}$$

$$< \frac{1}{2}\sum_{i=1}^{n-2}(n-1)(n-(i+1)) \tag{4}$$

$$= \frac{n-1}{2}\sum_{i=1}^{n-2}(n-(i+1)) \tag{5}$$

$$= \frac{n-1}{2}\sum_{i=1}^{n-2} i \tag{6}$$

$$= \frac{n-1}{2}\frac{(n-1)(n-2)}{2} \tag{7}$$

$$< \frac{\sqrt{2}(n-1)^3}{4} \tag{8}$$

$$= \left(\frac{(n-1)^2}{2}\right)^{3/2} \tag{9}$$

$$< \left(\frac{n(n-1)}{2}\right)^{3/2} \tag{10}$$

$$= m^{3/2} = O(m^{3/2}) \tag{11}$$

Hence, the first MapReduce task has complexity $O(m^{3/2})$.

For the second MapReduce phase the input is the emitted records of the first MapReduce phase ($O(m^{3/2})$) as well as the augmented edge list that was used as the input for the first MapReduce phase ($O(m)$). For the second MapReduce phase, let p be the size of the combined input, note:

$$O(p) = O(m) + O(m^{3/2}) = O(m^{3/2}).$$

Map 2 Combine degree-augmented file and output from Reduce 1. For the augmented edge list we have the following mapping to remove the vertex valences:

$$\begin{array}{c} \overline{key_1 = [e_1,\ \delta(v_x),\ \delta(v_y)]} \\ \overline{key_2 = [e_2,\ \delta(v_w),\ \delta(v_z)]} \\ \vdots \\ \overline{key_n = [e_n,\ \delta(v_u),\ \delta(v_t)]} \end{array} \quad \Rightarrow \quad \begin{array}{c} \overline{key_{e_1} = [e_1]} \\ \overline{key_{e_2} = [e_2]} \\ \vdots \\ \overline{key_{e_n} = [e_n]} \end{array},$$

and for the records emitted by Reduce 2, we have the identity operation. Therefore, this task is $O(p)$.

Reduce 2 Each bin corresponds with a vertex pair. A bin will contain at most one edge record and any number of triad records. With our assumptions of a completely connected graph, bin i contains $\binom{n-i}{2}$ triad records and one edge record, and the reducer will emit $\binom{n-i}{2}$ triangles. From our previous analysis, this is $O(m^{3/2})$ and therefore the overall complexity is $O(m^{3/2})$. □

6 Experiments

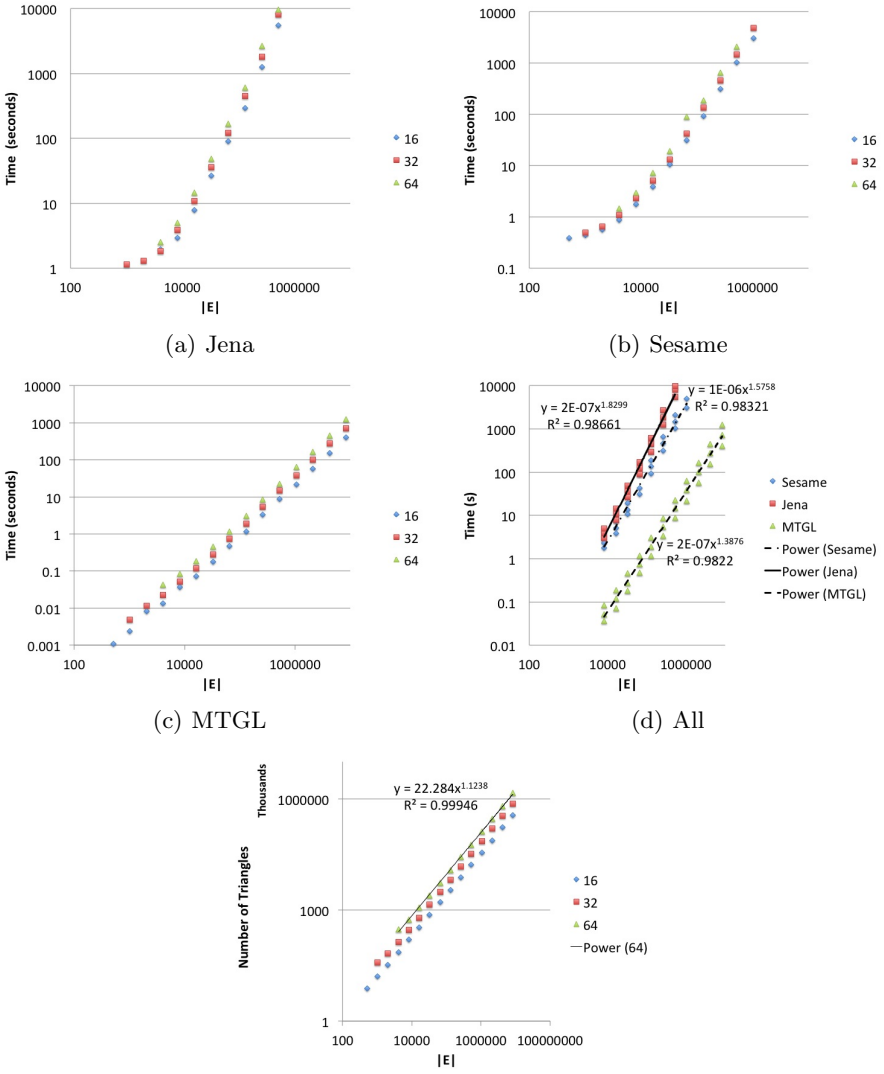
We experimentally compare open source RDF/SPARQL engines, Jena, version 2.7.2, and Sesame, version 2.6.8, with MTGL's implementation of Cohen's algorithm. For both Jena and Sesame we use the in-memory backend version of each. We did need to make some modifications to the MTGL version in order to allow for directionality and duplicate edges. Namely, we created a multimap that gives a the list of edges connecting any two vertices. This allowed us to create multiple triangles from single instances of a triad in the last reduce phase. We used a workstation with 8 GB of memory and a 2.2 GHz Intel Core i7 processor. Detailed times for our experiments can be found in the Appendix.

For our experiments we create R-MAT [4] graphs to simulate real-world graph properties such as power-law distributions on degree, small-world graphs, and small diameter. We varied the size of the graph from between $n = 2^5$ to $n = 2^{19}$. Also, we tried three different edge factors (average degree per vertex), namely 16, 32, and 64. R-MAT has four other parameters, a, b, c, and d. These four parameters are probabilities used recursively to determine where edges exist within the adjacency matrix. We set these to the values of the Graph500[6] search benchmark: $a = 0.57$, $b = 0.19$, $c = 0.19$, and $d = 0.05$. To enable the SPARQL engines the ability to process the data, we created IRI's of the form *<http://i>* where i is the vertex id given by the R-MAT generator. Also we made all edges of the same type.

Figures 3(a), 3(b), and 3(c) show the individual performance of each of the three platforms as a function of the number of edges. All of the plots have a log-log scale. Figure 3(d) shows the three platforms side by side. For this Figure, we exclude graphs below 8192 edges to give a better idea of the scaling behavior for larger graphs. Both Jena and Sesame exhibit a fair amount of constant overhead per query that dominates the times in the smaller graphs. When excluding this data, the fit of trendlines using power regression is quite good, with R^2 all exceeding 0.98. As can be seen from Figure 3(d), Jena has a complexity of around $O(m^{1.83})$, Sesame has $O(m^{1.58})$, and MTGL's version of Cohen's algorithm is around $O(m^{1.39})$. The best possible for this data would be around $O(m^{1.12})$, which is rate of growth of triangles for the data as determined experimentally and shown in Figure 3(e).

It is clear that both Jena and Sesame are not employing a triangle finding algorithm as their rate of growth is significantly larger than $O(m^{1.5})$. While the differences in powers may seem small between the three, consider that extrapolating out to a billion edges, the difference in times between an $O(m^{1.39})$

[6] www.graph500.org



(a) Jena (b) Sesame

(c) MTGL (d) All

(e) Triangle Growth

Fig. 3.

algorithm and one with $O(m^{1.58})$ with the same constant is about a 50x difference. And the difference between $O(m^{1.39})$ and $O(m^{1.83})$ is around 9000x. While not the focus of this paper, it is interesting to note that the MTGL version computed the triangles in about two-orders of magnitude less time than either Jena or Sesame.

7 Related Work

There is much work within the graph theory community where there is a straightforward application to the Semantic Web. For the most part, interfacing with the Semantic Web takes place through SPARQL. This largely confines interactions to subgraph matching tasks. Under this limitation, we can find work on clique-finding [5,10], or generalizations of cliques such as trusses [6]. An excellent overview of research in subgraph pattern matching over a thirty year timespan can be found in Conte et al. [7].

There are also algorithms outside of subgraph matching that can be of help in analyzing the Semantic Web. Probably one of the most fundamental algorithms is breadth-first search. In recent years, the Graph500 list has significantly increased the competitiveness and visibility of the task, and scalability and performance have increased concomitantly with the greater attention. Other algorithms include single source shortest path [13], betweeness centrality [1], connected components [12], and many others.

Related to all of these efforts is the task of graph partitioning. Graphs have historically been difficult to partition in a distributed memory setting, where the interconnectedness of the graph make it difficult to divide the data in such a way to minimize communication overhead. Notable efforts include Buluç and Madduri [3] and Devine et al. [9].

Programming models have also been created to aid development of graph-centric algorithms. Notable among them are Google's Pregel [14] and Signal/Collect by Stutz et al. [16]. These may prove to be valuable paradigms for implementing efficient graph-oriented code that can scale on large distributed systems.

8 Conclusions

In order for Semantic Web applications to scale, the community needs to adopt efficient algorithms to use as the computational kernels underlying analytics. In this paper we've demonstrated how long existing triangle finding algorithms can be employed to speed up SPARQL queries. We believe there are other algorithms and lessons from graph theory that can utilized to speed up Semantic Web applications and also open up other avenues for analysis.

References

1. D. A. Bader, S. Kintali, K. Madduri, and M. Mihail. Approximating betweenness centrality. In *Proceedings of the 5th international conference on Algorithms and models for the web-graph*, WAW'07, pages 124–137, Berlin, Heidelberg, 2007. Springer-Verlag.
2. J. W. Berry, B. Hendrickson, S. Kahan, and P. Konecny. Software and algorithms for graph queries on multithreaded architectures. *Parallel and Distributed Processing Symposium, International*, 0:495, 2007.

3. A. Buluç and K. Madduri. Graph partitioning for scalable distributed graph computations. In *Proc. 10th DIMACS Implementation Challenge Workshop – Graph Partitioning and Graph Clustering*, Feb. 2012.
4. D. Chakrabarti, Y. Zhan, and C. Faloutsos. R-mat: A recursive model for graph mining. In M. W. Berry, U. Dayal, C. Kamath, and D. B. Skillicorn, editors, *SDM*. SIAM, 2004.
5. N. Chiba and T. Nishizeki. Arboricity and subgraph listing algorithms. *SIAM Journal on Computing*, 14(1):210–223, 1985.
6. J. Cohen. Graph twiddling in a mapreduce world. *Computing in Science Engineering*, 11(4):29 –41, july-aug. 2009.
7. D. Conte, P. Foggia, C. Sansone, and M. Vento. Thirty years of graph matching in pattern recognition. *IJPRAI*, pages 265–298, 2004.
8. J. Dean and S. Ghemawat. Mapreduce: simplified data processing on large clusters. *Commun. ACM*, 51:107–113, January 2008.
9. K. D. Devine, E. G. Boman, R. T. Heaphy, R. H. Bisseling, and U. V. Catalyurek. Parallel hypergraph partitioning for scientific computing. In *Proceedings of the 20th international conference on Parallel and distributed processing*, IPDPS'06, pages 124–124, Washington, DC, USA, 2006. IEEE Computer Society.
10. F. Eisenbrand and F. Grandoni. On the complexity of fixed parameter clique and dominating set. *Theor. Comput. Sci.*, 326(1-3):57–67, Oct. 2004.
11. Y. Guo, Z. Pan, and J. Heflin. Lubm: A benchmark for owl knowledge base systems. *J. Web Sem.*, 3(2-3):158–182, 2005.
12. A. Krishnamurthy, S. S. Lumetta, D. E. Culler, and K. Yelick. Connected components on distributed memory machines. In *Parallel Algorithms: 3rd DIMACS Implementation Challenge October 17-19, 1994, volume 30 of DIMACS Series in Discrete Mathematics and Theoretical Computer Science*, pages 1–21. American Mathematical Society, 1994.
13. K. Madduri, D. Bader, J. Berry, and J. Crobak. An experimental study of a parallel shortest path algorithm for solving large-scale graph instances. In *ALENEX*, 2007.
14. G. Malewicz, M. H. Austern, A. J. Bik, J. C. Dehnert, I. Horn, N. Leiser, and G. Czajkowski. Pregel: a system for large-scale graph processing. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of data*, SIGMOD '10, pages 135–146, New York, NY, USA, 2010. ACM.
15. M. Schmidt, T. Hornung, G. Lausen, and C. Pinkel. Sp2bench: A sparql performance benchmark. *CoRR*, abs/0806.4627, 2008.
16. P. Stutz, A. Bernstein, and W. Cohen. Signal/collect: graph algorithms for the (semantic) web. In *Proceedings of the 9th international semantic web conference on The semantic web - Volume Part I*, ISWC'10, pages 764–780, Berlin, Heidelberg, 2010. Springer-Verlag.

A Experimental Data

Below is the data we collected running the triangle query on Jena, Sesame, and MTGL. We divide the data into three tables, one for each edge factor. Times are in seconds.

A.1 Edge Factor = 16

$\|V\|$	$\|E\|$	Num Triangles	Jena	Sesame	MTGL
2^5	512	7,645	0.66	0.39	0.0011
2^6	1,024	16,159	0.96	0.44	0.0024
2^7	2,048	33,263	1.23	0.57	0.0081
2^8	4,096	72,357	1.96	0.88	0.0132
2^9	8,192	156,716	2.96	1.78	0.0365
2^{10}	16,384	333,174	7.79	3.86	0.0721
2^{11}	32,768	739,951	26.38	10.67	0.1798
2^{12}	65,536	1,648,301	89.72	31.12	0.4696
2^{13}	131,072	3,450,520	291.65	92.78	1.1767
2^{14}	262,144	7,573,624	1263.91	317.21	3.3691
2^{15}	524,288	16,864,063	5550.93	1022.38	8.7734
2^{16}	1,048,576	35,286,039		3062.77	21.6454
2^{17}	2,097,152	74,837,468			57.1703
2^{18}	4,194,304	168,767,188			153.9390
2^{19}	8,388,608	357,383,850			409.6070

A.2 Edge Factor = 32

$\|V\|$	$\|E\|$	Num Triangles	Jena	Sesame	MTGL
2^5	1,024	37,561	1.151	0.496	0.0048
2^6	2,048	66,132	1.288	0.643	0.0115
2^7	4,096	137,725	1.845	1.083	0.0225
2^8	8,192	292,062	3.878	2.299	0.0514
2^9	16,384	624,619	10.916	5.064	0.1197
2^{10}	32,768	1,388,020	35.841	13.330	0.2796
2^{11}	65,536	3,033,157	122.567	42.596	0.7512
2^{12}	131,072	6,537,422	448.030	135.857	1.8940
2^{13}	262,144	14,955,653	1829.724	459.467	5.4287
2^{14}	524,288	32,521,939	8067.696	1486.286	14.8550
2^{15}	1,048,576	73,026,129		4916.445	38.6545
2^{16}	2,097,152	155,196,692			100.6970
2^{17}	4,194,304	342,379,527			275.3078
2^{18}	8,388,608	746,302,590			720.8690

A.3 Edge Factor = 64

$\|V\|$	$\|E\|$	Num Triangles	Jena	Sesame	MTGL
2^6	4096	303,016	2.510	1.464	0.0432
2^7	8,192	554,405	5.053	2.937	0.0843
2^8	16,384	1,151,110	14.524	7.235	0.1872
2^9	32,768	2,481,009	48.234	19.189	0.4539
2^{10}	65,536	5,452,648	168.198	91.224	1.1790
2^{11}	131,072	12,051,583	612.075	189.095	3.0500
2^{12}	262,144	26,614,815	2674.014	657.880	8.4796
2^{13}	524,288	57,835,285	9738.723	2103.770	22.5987
2^{14}	1,048,576	131,190,442			63.0300
2^{15}	2,097,152	290,104,980			166.7070
2^{16}	4,194,304	634,855,745			451.3670
2^{17}	8,388,608	1,420,402,577			1250.280

Cascading Map-Side Joins over HBase for Scalable Join Processing

Alexander Schätzle, Martin Przyjaciel-Zablocki,
Christopher Dorner, Thomas Hornung, and Georg Lausen

Department of Computer Science, University of Freiburg, Germany
{schaetzle,zablocki,dornerc,hornungt,lausen}
@informatik.uni-freiburg.de

Abstract. One of the major challenges in large-scale data processing with MapReduce is the smart computation of joins. Since Semantic Web datasets published in RDF have increased rapidly over the last few years, scalable join techniques become an important issue for SPARQL query processing as well. In this paper, we introduce the *Map-Side Index Nested Loop Join* (MAPSIN join) which combines scalable indexing capabilities of NoSQL data stores like HBase, that suffer from an insufficient distributed processing layer, with MapReduce, which in turn does not provide appropriate storage structures for efficient large-scale join processing. While retaining the flexibility of commonly used reduce-side joins, we leverage the effectiveness of map-side joins without any changes to the underlying framework. We demonstrate the significant benefits of MAPSIN joins for the processing of SPARQL basic graph patterns on large RDF datasets by an evaluation with the LUBM and SP^{2}Bench benchmarks. For selective queries, MAPSIN join based query execution outperforms reduce-side join based execution by an order of magnitude.

1 Introduction

Most of the information in the classical *"Web of Documents"* is designed for human readers, whereas the idea behind the Semantic Web is to build a *"Web of Data"* that enables computers to understand and use the information in the web. The advent of this Web of Data gives rise to new challenges with regard to query evaluation on the Semantic Web. The core technologies of the Semantic Web are RDF (Resource Description Framework) [1] for representing data in a machine-readable format and SPARQL [2] for querying RDF data. However, querying RDF datasets at web-scale is challenging, especially because the computation of SPARQL queries usually requires several joins between subsets of the data. On the other side, classical single-place machine approaches have reached a point where they cannot scale with respect to the ever increasing amount of available RDF data (cf. [16]). Renowned for its excellent scaling properties, the MapReduce paradigm [8] is an attractive candidate for distributed SPARQL processing. The *Apache Hadoop* platform is the most prominent and widely used open-source MapReduce implementation. In the last few years many companies

have built-up their own Hadoop infrastructure but there are also ready-to-use cloud services like Amazon's Elastic Compute Cloud (EC2) offering the Hadoop platform as a service (PaaS). Thus, in contrast to specialized distributed RDF systems like YARS2 [15] or 4store [14], the use of existing Hadoop MapReduce infrastructures enables scalable, distributed and fault-tolerant SPARQL processing out-of-the-box without any additional installation or management overhead. Following this avenue, we introduced the *PigSPARQL* project in [26] that offers full support for SPARQL 1.0 and is implemented on top of Hadoop. However, while the performance and scaling properties of PigSPARQL for complex analytical queries are competitive, the performance for selective queries is not satisfying due to the lack of built-in index structures and unnecessary data shuffling as join computation is done in the reduce phase.

In this paper we present a new MapReduce join technique, the *Map-Side Index Nested Loop Join* (MAPSIN join), that uses the indexing capabilities of a distributed NoSQL data store to improve query performance of selective queries. MAPSIN joins are completely processed in the map phase to avoid costly data shuffling by using HBase as underlying storage layer. Our evaluation shows an improvement of up to one order of magnitude over the common reduce-side join for selective queries. Overall, the major contributions of this paper are as follows:

- We describe a space-efficient storage schema for large RDF graphs in HBase while retaining favourable access characteristics. By using HBase instead of HDFS, we can avoid shuffling join partitions across the network and instead only access the relevant join partners in each iteration.
- We present the MAPSIN join algorithm, which can be evaluated cascadingly in subsequent MapReduce iterations. In contrast to other approaches, we do not require an additional shuffle and reduce phase in order to preprocess the data for consecutive joins. Moreover, we do not require any changes to the underlying frameworks.
- We demonstrate an optimization of the basic MAPSIN join algorithm for the efficient processing of multiway joins. This way, we can save n MapReduce iterations for star join queries with $n + 2$ triple patterns.

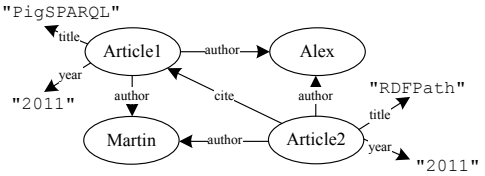
The paper is structured as follows: Section 2 provides a brief introduction to the technical foundations for this paper. Section 3 describes our RDF storage schema for HBase, while Section 4 presents the MAPSIN join algorithm. We continue with a presentation of the evaluation of our approach in Section 5, followed by a discussion of related work in Section 6. We conclude in Section 7 and give an outlook on future work.

2 Background

2.1 RDF & SPARQL

RDF [1] is the W3C recommended standard model for representing knowledge about arbitrary resources, e.g. articles and authors. An RDF dataset consists of a

set of RDF triples in the form (*subject, predicate, object*) that can be interpreted as "*subject* has property *predicate* with value *object*". For clarity of presentation, we use a simplified RDF notation in the following. It is possible to visualize an RDF dataset as directed, labeled graph where every triple corresponds to an edge (predicate) from subject to object. Figure 1 shows an RDF graph with information about articles and corresponding authors.



SPARQL BGP query

```
SELECT *
WHERE {
  ?article title  ?title .
  ?article author ?author .
  ?article year   ?year
}
```

Fig. 1. RDF graph and SPARQL query

SPARQL is the W3C recommended declarative query language for RDF. A SPARQL query defines a graph pattern P that is matched against an RDF graph G. This is done by replacing the variables in P with elements of G such that the resulting graph is contained in G (pattern matching). The most basic constructs in a SPARQL query are *triple patterns*, i.e. RDF triples where subject, predicate and object can be variables, e.g. ($?s$, p, $?o$). A set of triple patterns concatenated by AND (.) is called a *basic graph pattern* (BGP) as illustrated in Figure 1. The query asks for all articles with known title, author and year of publication. The result of a BGP is computed by joining the variable mappings of all triple patterns on their shared variables, in this case $?article$. For a detailed definition of the SPARQL syntax we refer the interested reader to the official W3C Recommendation [2]. A formal definition of the SPARQL semantics can also be found in [23]. In this paper we focus on efficient join processing with MapReduce and NoSQL (i.e. HBase) and therefore only consider SPARQL BGPs.

2.2 MapReduce

The MapReduce programming model [8] enables scalable, fault tolerant and massively parallel computations using a cluster of machines. The basis of Google's MapReduce is the distributed file system GFS [12] where large files are split into equal sized blocks, spread across the cluster and fault tolerance is achieved by replication. The workflow of a MapReduce program is a sequence of MapReduce iterations each consisting of a *Map* and a *Reduce* phase separated by a so-called *Shuffle & Sort* phase. A user has to implement *map* and *reduce* functions which are automatically executed in parallel on a portion of the data. The map function gets invoked for every input record represented as a key-value pair. It outputs a list of new intermediate key-value pairs which are then sorted and grouped by their key. The reduce function gets invoked for every distinct intermediate key

together with the list of all according values and outputs a list of values which can be used as input for the next MapReduce iteration.

We use *Apache Hadoop* as it is the most popular open-source implementation of Google's GFS and MapReduce framework that is used by many companies like Yahoo!, IBM or Facebook.

Map-Side vs. Reduce-Side Join. Processing joins with MapReduce is a challenging task as datasets are typically very large [5,20]. If we want to join two datasets with MapReduce, $L \bowtie R$, we have to ensure that the subsets of L and R with the same join key values can be processed on the same machine. For joining arbitrary datasets on arbitrary keys we generally have to shuffle data over the network or choose appropriate pre-partitioning and replication strategies.

The most prominent and flexible join technique in MapReduce is called *Reduce-Side Join* [5,20]. Some literature also refer to it as *Repartition Join* [5] as the idea is based on reading both datasets (map phase) and repartition them according to the join key (shuffle phase). The actual join computation is done in the reduce phase. The main drawback of this approach is that both datasets are completely transferred over the network regardless of the join output. This is especially inefficient for selective joins and consumes a lot of network bandwidth. Another group of joins is based on getting rid of the shuffle and reduce phase to avoid transferring both datasets over the network. This kind of join technique is called *Map-Side Join* since the actual join processing is done in the map phase. The most common one is the *Map-Side Merge Join* [20]. However, this join cannot be applied on arbitrary datasets. A preprocessing step is necessary to fulfill several requirements: datasets have to be sorted and equally partitioned according to the join key. If the preconditions are fulfilled, the map phase can process an efficient parallel merge join between pre-sorted partitions and data shuffling is not necessary. However, if we want to compute a sequence of such joins, the shuffle and reduce phases are needed to guarantee that the preconditions for the next join iteration are fulfilled. Therefore, map-side joins are generally hard to cascade and the advantage of avoiding a shuffle and reduce phase is lost. Our MAPSIN join approach is designed to overcome this drawback by using the distributed index of a NoSQL system like HBase.

2.3 HBase

HBase is a distributed, scalable and strictly consistent column-oriented NoSQL data store, inspired by Google's Bigtable [7] and well integrated into Hadoop. Hadoop's distributed file system, HDFS, is designed for sequential reads and writes of very large files in a batch processing manner but lacks the ability to access data randomly in close to real-time. HBase can be seen as an additional storage layer on top of HDFS that supports efficient random access. The data model of HBase corresponds to a sparse multi-dimensional sorted map with the following access pattern:

$$(Table, RowKey, Family, Column, Timestamp) \rightarrow Value$$

The rows of a table are sorted and indexed according to their *row key* and every row can have an arbitrary number of *columns*. Columns are grouped into *column families* and column values (denoted as cell) are timestamped and thus support multiple versions. HBase tables are dynamically split into *regions* of contiguous row ranges with a configured maximum size. When a region becomes too large, it is automatically split into two regions at the middle key (auto-sharding).

However, HBase has neither a declarative query language nor built-in support for native join processing, leaving higher-level data transformations to the overlying application layer. In our approach we propose a map-side join strategy that leverages the implicit index capabilities of HBase to overcome the usual restrictions of map-side joins as outlined in Section 2.2.

3 RDF Storage Schema for HBase

In contrast to relational databases, NoSQL data stores do neither have a common data model nor a common query language like SQL. Hence, the implementation of our join approach strongly relies on the actual NoSQL store used as backend. In our initial experiments we considered HBase and Cassandra, two popular NoSQL stores with support for MapReduce. We decided to use HBase for our implementation as it proved to be more stable and also easier to handle in our cluster since HBase was developed to work with Hadoop from the beginning.

In [28] the authors adopted the idea of Hexastore [30] to index all possible orderings of an RDF triple for storing RDF data in HBase. This results in six tables in HBase allowing to retrieve results for any possible SPARQL triple pattern with a single lookup on one of the tables (except for a triple pattern with three variables). However, as HDFS has a default replication factor of three and data in HBase is stored in files on HDFS, an RDF dataset is actually stored 18 times using this schema. But it's not only about storage space, also loading a web-scale RDF dataset into HBase becomes very costly and consumes many resources. Our storage schema for RDF data in HBase is inspired by [10] and uses only two tables, T_{s_po} and T_{o_ps}. We extend the schema with a triple pattern mapping that leverages the power of predicate push-down filters in HBase to overcome possible performance shortcomings of a two table schema. Furthermore, we improve the scalibility of the schema by introducing a modified row key design for class assignments in RDF which would otherwise lead to overloaded regions constraining both scalability and performance.

In T_{s_po} table an RDF triple is stored using the subject as row key, the predicate as column name and the object as column value. If a subject has more than one object for a given predicate (e.g. an article having more than one author), these objects are stored as different versions in the same column. The notation T_{s_po} indicates that the table is indexed by subject. Table T_{o_ps} follows the same design. In both tables there is only one single column family that contains all columns. Table 1 illustrates the corresponding T_{s_po} table for the RDF graph in Section 2.1.

Table 1. T_{s_po} table for RDF graph in Section 2.1

rowkey	family:column→value
Article1	p:title→{"PigSPARQL"}, p:year→{"2011"}, p:author→{Alex, Martin}
Article2	p:title→{"RDFPath"}, p:year→{"2011"}, p:author→{Martin, Alex}, p:cite→{Article1}

At first glance, this storage schema seems to have performance drawbacks when compared to the six table schema in [28] since there are only indexes for subjects and objects. However, we can use the HBase Filter API to specify additional column filters for table index lookups. These filters are applied directly on server side such that no unnecessary data must be transferred over the network (*predicate push-down*). As already mentioned in [10], a table with predicates as row keys causes scalability problems since the number of predicates in an ontology is usually fixed and relatively small which results in a table with just a few very fat rows. Considering that all data in a row is stored on the same machine, the resources of a single machine in the cluster become a bottleneck. Indeed, if only the predicate in a triple pattern is given, we can use the HBase Filter API to answer this request with a table scan on T_{s_po} or T_{o_ps} using the predicate as column filter. Table 2 shows the mapping of every possible triple pattern to the corresponding HBase table. Overall, experiments on our cluster showed that the two table schema with server side filters has similar performance characteristics compared to the six table schema but uses only one third of storage space.

Table 2. SPARQL triple pattern mapping using HBase predicate push-down filters

pattern	table	filter
(s, p, o)	T_{s_po} or T_{o_ps}	column & value
(?s, p, o)	T_{o_ps}	column
(s, ?p, o)	T_{s_po} or T_{o_ps}	value
(s, p, ?o)	T_{s_po}	column
(?s, ?p, o)	T_{o_ps}	
(?s, p, ?o)	T_{s_po} or T_{o_ps} (table scan)	column
(s, ?p, ?o)	T_{s_po}	
(?s, ?p, ?o)	T_{s_po} or T_{o_ps} (table scan)	

Our experiments also revealed some fundamental scaling limitations of the storage schema caused by the T_{o_ps} table. In general, an RDF dataset uses a relatively small number of classes but contains many triples that link resources to classes, e.g. (Alex, rdf:type, foaf:Person). Thus, using the object of a triple as row key means that all resources of the same class will be stored in the same row. With increasing dataset size these rows become very large and exceed the configured maximum region size resulting in overloaded regions that contain only a single row. Since HBase cannot split these regions the resources of a single machine become a bottleneck for scalability. To circumvent this problem we use a modified T_{o_ps} row key design for triples with predicate rdf:type. Instead of using the object as row key we use a compound row key of object and subject,

e.g. (foaf:Person|Alex). As a result, we can not access all resources of a class with a single table lookup but as the corresponding rows will be consecutive in T_{o_ps} we can use an efficient range scan starting at the first entry of the class.

4 MAPSIN Join

The major task in SPARQL query evaluation is the computation of joins between triple patterns, i.e. basic graph patterns. However, join processing on large RDF datasets, especially if it involves more than two triple patterns, is challenging [20]. Our approach combines the scalable storage capabilities of NoSQL data stores (i.e. HBase), that suffer from a suitable distributed processing layer, with MapReduce, a highly scalable and distributed computation framework, which in turn does not support appropriate storage structures for large scale join processing. This allows us to catch up with the flexibility of reduce-side joins while utilizing the effectiveness of a map-side join without any changes to the underlying frameworks.

First, we introduce the needed SPARQL terminology analogous to [23]: Let V be the infinite set of query variables and T be the set of valid RDF terms.

Definition 1. *A (solution) mapping μ is a partial function $\mu : V \rightarrow T$. We call $\mu(?v)$ the variable binding of μ for $?v$. Abusing notation, for a triple pattern p we call $\mu(p)$ the triple pattern that is obtained by substituting the variables in p according to μ. The domain of μ, $dom(\mu)$, is the subset of V where μ is defined and the domain of p, $dom(p)$, is the subset of V used in p. The result of a SPARQL query is a multiset of solution mappings Ω.*

Definition 2. *Two mappings μ_1, μ_2 are compatible if, for every variable $?v \in dom(\mu_1) \cap dom(\mu_2)$, it holds that $\mu_1(?v) = \mu_2(?v)$. It follows that mappings with disjoint domains are always compatible and the set-union (merge) of μ_1 and μ_2, $\mu_1 \cup \mu_2$, is also a mapping.*

4.1 Base Case

To compute the join between two triple patterns, $p_1 \bowtie p_2$, we have to merge the compatible mappings for p_1 and p_2. Therefore, it is necessary that subsets of both multisets of mappings are brought together such that all compatible mappings can be processed on the same machine.

Our MAPSIN join technique computes the join between p_1 and p_2 in a single map phase. At the beginning, the map phase is initialized with a parallel distributed HBase table scan for the first triple pattern p_1 where each machine retrieves only those mappings that are locally available. This is achieved by utilizing a mechanism for allocating local records to map functions, which is supported by the MapReduce input format for HBase. The map function is invoked for each retrieved mapping μ_1 for p_1. To compute the partial join between p_1 and p_2 for the given mapping μ_1, the map function needs to retrieve those

mappings for p_2 that are compatible to μ_1 based on the shared variables between p_1 and p_2. At this point, the map function utilizes the input mapping μ_1 to substitute the shared variables in p_2, i.e. the join variables. The substituted triple pattern p_2^{sub} is then used to retrieve the compatible mappings with a table lookup in HBase following the triple pattern mapping outlined in Table 2. Since there is no guarantee that the corresponding HBase entries reside on the same machine, the results of the request have to be transferred over the network in general. However, in contrast to a reduce-side join approach where a lot of data is transferred over the network, we only transfer the data that is really needed. Finally, the computed multiset of mappings is stored in HDFS.

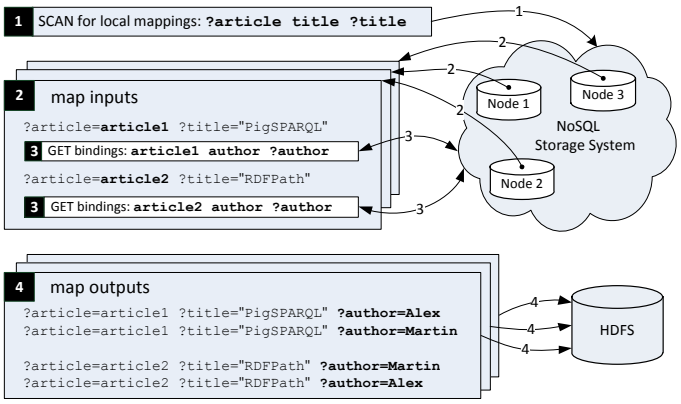


Fig. 2. MAPSIN join base case for the first two triple patterns of query in Figure 1

Figure 2 is an example for the base case of our MAPSIN join that illustrates the join between the first two triple patterns of the SPARQL query in Figure 1. While the mappings for the first triple pattern (?*article*, title, ?*title*) are retrieved locally using a distributed table scan (step 1+2), the compatible mappings for (?*article*, author, ?*author*) are requested within the map function (step 3) and the resulting set of mappings is stored in HDFS (step 4).

4.2 Cascading Joins

Chains of concatenated triple patterns require some slight modifications to the previously described base case. To compute a query of at least three triple patterns we have to process several joins successively, e.g. $p_1 \bowtie p_2 \bowtie p_3$. The processing of the first two patterns $p_1 \bowtie p_2$ correspond to the base case and the results are stored in HDFS. The additional triple pattern p_3 is then joined with the mappings for $p_1 \bowtie p_2$. To this end, an additional map-phase (without any intermediate shuffle or reduce phase) is initialized with the previously computed mappings as input. Since these mappings reside in HDFS, they are retrieved

locally in parallel such that the map function gets invoked for each mapping μ_2 for $p_1 \bowtie p_2$. The compatible mappings for p_3 are retrieved using the same strategy as for the base case, i.e. μ_2 is used to substitute the shared variables in p_3 and compatible mappings are retrieved following the triple pattern mapping outlined in Table 2. Algorithm 1 outlines one iteration of the MAPSIN join. The input for the map function contains either a mapping for the first triple pattern (via distributed table scan) or a mapping for previously joined triple patterns (loaded from HDFS).

Algorithm 1: MAPSIN join: **map**(inKey, inValue)

```
   input : inKey, inValue: value contains input mapping, key can be ignored
   output: multiset of mappings
 1 p_{n+1} ← Config.getNextPattern()
 2 μ_n ← inValue.getInputMapping()
 3 Ω_{n+1} ← ∅
 4 if dom(μ_n) ∩ dom(p_{n+1}) ≠ ∅ then
 5 |   // substitute shared vars in p_{n+1}
 6 |   p^{sub}_{n+1} ← μ_n(p_{n+1})
 7 |   results ← HBase.GET(p^{sub}_{n+1}) // table index lookup using substituted pattern
 8 else
 9 |   results ← HBase.GET(p_{n+1}) // table index lookup using unsubstituted pattern
10 end
11 if results ≠ ∅ then
12 |   // merge μ_n with compatible mappings for p_{n+1}
13 |   foreach mapping μ in results do
14 |   |   μ_{n+1} ← μ_n ∪ μ
15 |   |   Ω_{n+1} ← Ω_{n+1} ∪ μ_{n+1}
16 |   end
17 |   emit(null, Ω_{n+1}) // key is not used since there is no reduce phase
18 end
```

4.3 Multiway Join Optimization

Instead of processing concatenated triple patterns successively as a sequence of two-way joins, some basic graph patterns allow to apply a multiway join approach to process joins between several concatenated triple patterns at once in a single map phase. This is typically the case for star pattern queries where triple patterns share the same join variable. The SPARQL query introduced in Section 2.1 is an example for such a query as all triple patterns share the same join variable ?*article*. This query can be processed by a three-way join in a single map-phase instead of two consecutive two-way joins.

We extended our approach to support this multiway join optimization. Again, the first triple pattern p_1 is processed using a distributed table scan as input for the map phase. But instead of using a sequence of n map phases to compute $p_1 \bowtie p_2 \bowtie ... \bowtie p_{n+1}$ we use a single map phase thus saving $n-1$ MapReduce iterations. Hence, the map function needs to retrieve all mappings for $p_2, p_3, ..., p_{n+1}$ that are compatible to the input mapping μ_1 for p_1. Therefore, the join variable $?v_s$ in $p_2, p_3, ..., p_{n+1}$ (e.g. ?*article*) is substituted with the corresponding variable

binding $\mu_1(?v_s)$. The substituted triple patterns $p_2^{sub}, p_3^{sub}, ..., p_{n+1}^{sub}$ are then used to retrieve the compatible mappings using HBase table lookups. This general case of the MAPSIN multiway join is outlined in Algorithm 2.

Algorithm 2: MAPSIN multiway join: **map**(inKey, inValue)

```
   input : inKey, inValue: value contains input mapping, key can be ignored
   output: multiset of mappings
 1 #p ← Config.getNumberOfMultiwayPatterns()
 2 μ_n ← inValue.getInputMapping()
 3 Ω_n ← {μ_n}
 4 // iterate over all subsequent multiway patterns
 5 for i ← 1 to #p do
 6 |   Ω_{n+i} ← ∅
 7 |   p_{n+i} ← Config.getNextPattern()
 8 |   // substitute shared vars in p_{n+i}
 9 |   p_{n+i}^{sub} ← μ_n(p_{n+i})
10 |   results ← HBase.GET(p_{n+i}^{sub}) // table index lookup using substituted pattern
11 |   if results ≠ ∅ then
12 |   |   // merge previous mappings with compatible mappings for p_{n+i}
13 |   |   foreach mapping μ in results do
14 |   |   |   foreach mapping μ' in Ω_{n+i-1} do
15 |   |   |   |   Ω_{n+i} ← Ω_{n+i} ∪ (μ ∪ μ')
16 |   |   |   end
17 |   |   end
18 |   else
19 |   |   // no compatible mappings for p_{n+i} hence join result for μ_n is empty
20 |   |   return
21 |   end
22 end
23 emit(null, Ω_{n+#p}) // key is not used since there is no reduce phase
```

The performance of MAPSIN joins strongly correlates with the number of index lookups in HBase. Hence, minimizing the number of lookups is a crucial point for optimization. In many situations, it is possible to reduce the number of requests by leveraging the RDF schema design for HBase outlined in Section 3. If the join variable for all triple patterns is always on subject or always on object position, then all mappings for $p_2, p_3, ..., p_{n+1}$ that are compatible to the input mapping μ_1 for p_1 are stored in the same HBase table row of T_{s_po} or T_{o_ps}, respectively, making it possible to use a single instead of n subsequent table lookups. Hence, all compatible mappings can be retrieved at once thus saving $n-1$ lookups for each invocation of the map function. Due to space limitations the corresponding algorithm for this optimized case can be found in the technical report version of this paper [24].

5 Evaluation

The evaluation was performed on a cluster of 10 Dell PowerEdge R200 servers equipped with a Dual Core 3.16 GHz CPU, 8 GB RAM, 3 TB disk space and connected via gigabit network. The software installation includes Hadoop 0.20.2, HBase 0.90.4 and Java 1.6.0 update 26.

Table 3. SP^{2}Bench & LUBM loading times for tables T_{s_po} and T_{o_ps} (hh:mm:ss)

SP^{2}Bench	**200M**	**400M**	**600M**	**800M**	**1000M**
# RDF triples	~ 200 million	~ 400 million	~ 600 million	~ 800 million	~ 1000 million
T_{s_po}	00:28:39	00:45:33	01:01:19	01:16:09	01:33:47
T_{o_ps}	00:27:24	01:04:30	01:28:23	01:43:36	02:19:05
total	00:56:03	01:50:03	02:29:42	02:59:45	03:52:52
LUBM	**1000**	**1500**	**2000**	**2500**	**3000**
# RDF triples	~ 210 million	~ 315 million	~ 420 million	~ 525 million	~ 630 million
T_{s_po}	00:28:50	00:42:10	00:52:03	00:56:00	01:05:25
T_{o_ps}	00:48:57	01:14:59	01:21:53	01:38:52	01:34:22
total	01:17:47	01:57:09	02:13:56	02:34:52	02:39:47

We used the well-known Lehigh University Benchmark (LUBM) [13] as the queries can easily be formulated as SPARQL basic graph patterns. Furthermore, we also considered the SPARQL-specific SP^{2}Bench Performance Benchmark [27]. However, because most of the SP^{2}Bench queries are rather complex queries that use all different kinds of SPARQL 1.0 operators, we only evaluated some of the queries as the focus of our work is the efficient computation of joins, i.e. basic graph patterns. Both benchmarks offer synthetic data generators that can be used to generate arbitrary large datasets. For SP^{2}Bench we generated datasets from 200 million up to 1000 million triples. For LUBM we generated datasets from 1000 up to 3000 universities and used the WebPIE inference engine for Hadoop [29] to pre-compute the transitive closure. The loading times for both tables T_{s_po} and T_{o_ps} as well as all datasets are listed in Table 3.

The goal of our approach was to optimize MapReduce based join computation for selective queries. Therefore, we compared our MAPSIN join approach with the reduce-side join based query execution in PigSPARQL [26], a SPARQL 1.0 engine built on top of *Pig*. Pig is an Apache top-level project developed by Yahoo! Research that offers a high-level language for the analysis of very large datasets with Hadoop MapReduce. The crucial point for this choice was the sophisticated and efficient reduce-side join implementation of Pig [11] that incorporates sampling and hash join techniques which makes it a challenging candidate for comparison. We illustrate the performance comparison of PigSPARQL and MAPSIN for some selected LUBM queries that represent the different query types in Figure 3. Our proof-of-concept implementation is currently limited to a maximum number of two join variables as the goal was to demonstrate the feasibility of the approach for selective queries rather than supporting all possible BGP constellations. For detailed comparison, the runtimes of all executed queries are listed in Table 4.

LUBM queries Q1, Q3, Q5, Q11, Q13 as well as SP^{2}Bench query Q3a demonstrate the base case with a single join between two triple patterns (cf. Figure 3a). For the LUBM queries, MAPSIN joins performed 8 to 13 times faster compared to the reduce-side joins of PigSPARQL. Even for the less selective SP^{2}Bench query, our MAPSIN join required only one third of the PigSPARQL execution time. Furthermore, the performance gain increases with the size of the dataset for both LUBM and SP^{2}Bench.

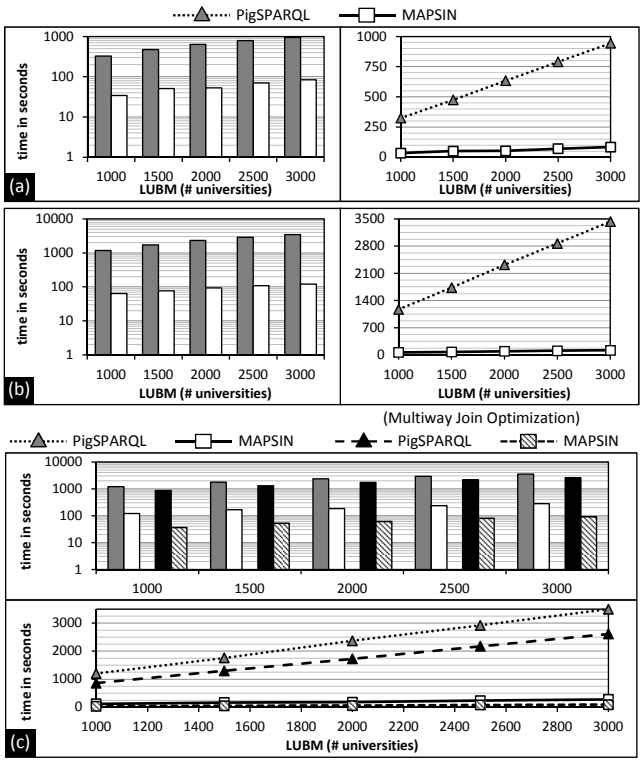


Fig. 3. Performance comparison for LUBM Q1 (a), Q8 (b), Q4 (c)

LUBM queries Q4 (5 triple patterns), Q7 (4 triple patterns), Q8 (5 triple patterns) and SP^{2}Bench queries Q1 (3 triple patterns), Q2 (9 triple patterns) demonstrate the more general case with a sequence of cascaded joins (cf. Figure 3b). In these cases, MAPSIN joins perform even up to 28 times faster than PigSPARQL for LUBM queries and up to 12 times faster for SP^{2}Bench queries.

Of particular interest are queries Q4 of LUBM and Q1, Q2 of SP^{2}Bench since these queries support the multiway join optimization outlined in Section 4.3 as all triple patterns share the same join variable. This kind of optimization is also supported by PigSPARQL such that both approaches can compute the query results with a single multiway join (cf. Figure 3c). The MAPSIN multiway join optimization improves the basic MAPSIN join execution time by a factor of 1.4 (SP^{2}Bench Q1) to 3.3 (LUBM Q4), independently of the data size. For the LUBM queries, the MAPSIN multiway join optimization performs 19 to 28 times faster than the reduce-side based multiway join implementation of PigSPARQL. For the more complex SP^{2}Bench queries, the performance improvements degrade to a factor of approximately 8.5.

The remaining queries (LUBM Q6, Q14 and SP^{2}Bench Q10) consist of only one single triple pattern. Consequently they do not contain a join processing

Table 4. Query execution times for PigSPARQL (P) and MAPSIN (M) in seconds

LUBM	**1000**		**1500**		**2000**		**2500**		**3000**	
	P	M	P	M	P	M	P	M	P	M
Q1	324	34	475	51	634	53	790	70	944	84
Q3	324	33	480	42	642	49	805	59	961	72
Q4	1202	121	1758	167	2368	182	2919	235	3496	279
Q4 MJ	861	37	1297	53	1728	62	2173	81	2613	92
Q5	329	33	484	44	640	53	800	66	955	80
Q6	149	48	214	60	284	69	355	84	424	104
Q7	1013	62	1480	68	1985	93	2472	114	2928	123
Q8	1172	64	1731	77	2318	33	2870	108	3431	121
Q11	319	33	469	46	620	53	780	69	931	79
Q13	325	44	482	72	645	84	800	108	957	128
Q14	149	43	214	70	288	79	364	89	434	107
SP^2Bench	**200M**		**400M**		**600M**		**800M**		**1000M**	
	P	M	P	M	P	M	P	M	P	M
Q1	545	58	1026	118	1527	153	2018	177	2519	214
Q1 MJ	310	42	600	87	896	118	1187	154	1476	174
Q2 MJ	1168	241	2341	444	3514	671	4745	834	6005	999
Q3a	227	70	435	139	641	178	845	235	1050	274
Q10	99	40	174	84	254	111	340	151	414	167

step and illustrate primarily the advantages of the distributed HBase table scan compared to the HDFS storage access of PigSPARQL. Improvements are still present but less significant, resulting in an up to 5 times faster query execution.

An open issue of the evaluation remains the actual data flow between HBase and MapReduce as HBase is like a black box where data distribution and partitioning is handled by the system automatically. Since data locality is an important aspect of distributed systems, it is crucial to examine additional measures for future optimizations.

Overall, the MAPSIN join approach clearly outperforms the reduce-side join based query execution for selective queries. Both approaches reveal a linear scaling behavior with the input size but the slope of the MAPSIN join is much smaller. Especially for LUBM queries, MAPSIN joins outperform reduce-side joins by an order of magnitude as these queries are generally rather selective. Moreover, the application of the multiway join optimization results in a further significant improvement of the total query execution times.

6 Related Work

Single machine RDF systems like *Sesame* [6] and *Jena* [31] are widely-used since they are user-friendly and perform well for small and medium sized RDF datasets. *RDF-3X* [21] is considered one of the fastest single machine RDF systems in terms of query performance that vastly outperforms previous single machine systems but performance degrades for queries with unbound objects and low selectivity factor [17]. Furthermore, as the amount of RDF data continues to grow, it will become more and more difficult to store entire datasets on a single machine due to the limited scaling capabilities [16]. One possible approach are specialized clustered RDF systems like OWLIM [19], YARS2 [15] or 4store [14].

However, these systems require a dedicated infrastructure and pose additional installation and management overhead. In contrast, our approach builds upon the idea to use existing infrastructures that are well-known and widely used. As we do not require any changes to Hadoop and HBase at all, it is possible to use any existing Hadoop cluster or cloud service (e.g. Amazon EC2) out of the box.

There is a large body of work dealing with join processing in MapReduce considering various aspects and application fields [4,5,18,20,22,25,32]. In Section 2.2 we briefly outlined the advantages and drawbacks of the general-purpose reduce-side and map-side (merge) join approaches in MapReduce. In addition to these general-purpose approaches there are several proposals focusing on certain join types or optimizations of existing join techniques for particular application fields. In [22] the authors discussed how to process arbitrary joins (theta joins) using MapReduce, whereas [4] focuses on optimizing multiway joins. However, in contrast to our MAPSIN join, both approaches process the join in the reduce phase including a costly data shuffle phase. *Map-Reduce-Merge* [32] describes a modified MapReduce workflow by adding a merge phase after the reduce phase, whereas *Map-Join-Reduce* [18] proposes a join phase in between the map and reduce phase. Both techniques attempt to improve the support for joins in MapReduce but require profound modifications to the MapReduce framework. In [9] the authors present non-invasive index and join techniques for SQL processing in MapReduce that also reduce the amount of shuffled data at the cost of an additional co-partitioning and indexing phase at load time. However, the schema and workload is assumed to be known in advance which is typically feasible for relational data but does not hold for RDF in general.

HadoopDB [3] is a hybrid of MapReduce and DBMS where MapReduce is the communication layer above multiple single node DBMS. The authors in [16] adopt this hybrid approach for the semantic web using RDF-3X. However, the initial graph partitioning is done on a single machine and has to be repeated if the dataset is updated or the number of machines in the cluster change. As we use HBase as underlying storage layer, additional machines can be plugged in seamlessly and updates are possible without having to reload the entire dataset.

HadoopRDF [17] is a MapReduce based RDF system that stores data directly in HDFS and does also not require any changes to the Hadoop framework. It is able to rebalance automatically when cluster size changes but join processing is also done in the reduce phase. Our MAPSIN join does not use any shuffle or reduce phase at all even in consecutive iterations.

7 Conclusion

In this paper we introduced the Map-Side Index Nested Loop join (MAPSIN join) which combines the advantages of NoSQL data stores like HBase with the well-known and approved distributed processing facilities of MapReduce. In general, map-side joins are more efficient than reduce-side joins in MapReduce as there is no expensive data shuffle phase involved. However, current map-side join approaches suffer from strict preconditions what makes them hard to ap-

ply in general, especially in a sequence of joins. The combination of HBase and MapReduce allows us to cascade a sequence of MAPSIN joins without having to sort and repartition the intermediate output for the next iteration. Furthermore, with the multiway join optimization we can reduce the number of MapReduce iterations and HBase requests. Using an index to selectively request only those data that is really needed also saves network bandwidth, making parallel query execution more efficient. The evaluation with the LUBM and SP^{2}Bench benchmarks demonstrate the advantages of our approach compared to the commonly used reduce-side join approach in MapReduce. For selective queries, MAPSIN join based SPARQL query execution outperforms reduce-side join based execution by an order of magnitude while scaling very smoothly with the input size. Lastly, our approach does not require any changes to Hadoop and HBase at all. Consequently, MAPSIN joins can be run on any existing Hadoop infrastructure and also on an instance of Amazon's Elastic Compute Cloud (EC2) without additional installation or management overhead.

In our future work, we will investigate alternatives and improvements of the RDF storage schema for HBase and incorporate MAPSIN joins into PigSPARQL in a hybrid fashion such that the actual join method is dynamically selected based on pattern selectivity and statistics gathered at data loading time.

References

1. RDF Primer. W3C Recom. (2004), `http://www.w3.org/TR/rdf-primer/`
2. SPARQL Query Language for RDF. W3C Recom. (2008), `http://www.w3.org/TR/rdf-sparql-query/`
3. Abouzeid, A., Bajda-Pawlikowski, K., Abadi, D.J., Rasin, A., Silberschatz, A.: HadoopDB: An Architectural Hybrid of MapReduce and DBMS Technologies for Analytical Workloads. PVLDB 2(1), 922–933 (2009)
4. Afrati, F.N., Ullman, J.D.: Optimizing Multiway Joins in a Map-Reduce Environment. IEEE Trans. Knowl. Data Eng. 23(9), 1282–1298 (2011)
5. Blanas, S., Patel, J.M., Ercegovac, V., Rao, J., Shekita, E.J., Tian, Y.: A Comparison of Join Algorithms for Log Processing in MapReduce. In: SIGMOD (2010)
6. Broekstra, J., Kampman, A., van Harmelen, F.: Sesame: A Generic Architecture for Storing and Querying RDF and RDF Schema. In: ISWC (2002)
7. Chang, F., Dean, J., Ghemawat, S., Hsieh, W., Wallach, D., Burrows, M., Chandra, T., Fikes, A., Gruber, R.: Bigtable: A Distributed Storage System for Structured Data. ACM Transactions on Computer Systems (TOCS) 26(2), 4 (2008)
8. Dean, J., Ghemawat, S.: MapReduce: Simplified Data Processing on Large Clusters. Communications of the ACM 51(1), 107–113 (2008)
9. Dittrich, J., Quiané-Ruiz, J.A., Jindal, A., Kargin, Y., Setty, V., Schad, J.: Hadoop++: Making a Yellow Elephant Run Like a Cheetah (Without It Even Noticing). PVLDB 3(1), 518–529 (2010)
10. Franke, C., Morin, S., Chebotko, A., Abraham, J., Brazier, P.: Distributed Semantic Web Data Management in HBase and MySQL Cluster. In: IEEE International Conference on Cloud Computing (CLOUD). pp. 105 –112 (2011)
11. Gates, A.F., Natkovich, O., Chopra, S., Kamath, P., Narayanamurthy, S.M., Olston, C., Reed, B., Srinivasan, S., Srivastava, U.: Building a High-Level Dataflow System on top of Map-Reduce: The Pig Experience. PVLDB 2(2) (2009)

12. Ghemawat, S., Gobioff, H., Leung, S.: The Google File System. In: ACM SIGOPS Operating Systems Review. vol. 37, pp. 29–43. ACM (2003)
13. Guo, Y., Pan, Z., Heflin, J.: LUBM: A Benchmark for OWL Knowledge Base Systems. Web Semantics 3(2) (2005)
14. Harris, S., Lamb, N., Shadbolt, N.: 4store: The Design and Implementation of a Clustered RDF Store. In: SSWS. pp. 94–109 (2009)
15. Harth, A., Umbrich, J., Hogan, A., Decker, S.: YARS2: A Federated Repository for Querying Graph Structured Data from the Web. The Semantic Web (2007)
16. Huang, J., Abadi, D.J., Ren, K.: Scalable SPARQL Querying of Large RDF Graphs. PVLDB 4(11), 1123–1134 (2011)
17. Husain, M.F., McGlothlin, J.P., Masud, M.M., Khan, L.R., Thuraisingham, B.M.: Heuristics-Based Query Processing for Large RDF Graphs Using Cloud Computing. IEEE TKDE 23(9) (2011)
18. Jiang, D., Tung, A.K.H., Chen, G.: Map-Join-Reduce: Toward Scalable and Efficient Data Analysis on Large Clusters. IEEE TKDE 23(9), 1299–1311 (2011)
19. Kiryakov, A., Ognyanov, D., Manov, D.: OWLIM - A Pragmatic Semantic Repository for OWL. In: WISE Workshops. pp. 182–192 (2005)
20. Lee, K.H., Lee, Y.J., Choi, H., Chung, Y.D., Moon, B.: Parallel Data Processing with MapReduce: A Survey. SIGMOD Record 40(4), 11–20 (2011)
21. Neumann, T., Weikum, G.: RDF-3X: a RISC-style engine for RDF. PVLDB 1(1), 647–659 (2008)
22. Okcan, A., Riedewald, M.: Processing Theta-Joins using MapReduce. In: SIGMOD Conference. pp. 949–960 (2011)
23. Pérez, J., Arenas, M., Gutierrez, C.: Semantics and Complexity of SPARQL. ACM Transactions on Database Systems (TODS) 34(3), 16 (2009)
24. Przyjaciel-Zablocki, M., Schätzle, A., Hornung, T., Dorner, C., Lausen, G.: Cascading Map-Side Joins over HBase for Scalable Join Processing. Technical Report. CoRR abs/1206.6293 (2012)
25. Przyjaciel-Zablocki, M., Schätzle, A., Hornung, T., Lausen, G.: RDFPath: Path Query Processing on Large RDF Graphs with MapReduce. In: ESWC Workshops. pp. 50–64 (2011)
26. Schätzle, A., Przyjaciel-Zablocki, M., Lausen, G.: PigSPARQL: Mapping SPARQL to Pig Latin. In: Proceedings of the International Workshop on Semantic Web Information Management (SWIM). pp. 4:1–4:8 (2011)
27. Schmidt, M., Hornung, T., Lausen, G., Pinkel, C.: SP2Bench: A SPARQL Performance Benchmark. In: ICDE. pp. 222–233 (2009)
28. Sun, J., Jin, Q.: Scalable RDF Store Based on HBase and MapReduce. In: ICACTE. vol. 1, pp. 633–636 (2010)
29. Urbani, J., Kotoulas, S., Maassen, J., van Harmelen, F., Bal, H.: OWL Reasoning with WebPIE: Calculating the Closure of 100 Billion Triples. In: ESWC. pp. 213–227 (2010)
30. Weiss, C., Karras, P., Bernstein, A.: Hexastore: Sextuple Indexing for Semantic Web Data Management. PVLDB 1(1), 1008–1019 (2008)
31. Wilkinson, K., Sayers, C., Kuno, H.A., Reynolds, D.: Efficient RDF Storage and Retrieval in Jena2. In: SWDB. pp. 131–150 (2003)
32. Yang, H.C., Dasdan, A., Hsiao, R.L., Jr., D.S.P.: Map-Reduce-Merge: Simplified Relational Data Processing on Large Clusters. In: SIGMOD (2007)

Scalable Nonmonotonic Reasoning over RDF data using MapReduce

Ilias Tachmazidis[1,2], Grigoris Antoniou[1,3], Giorgos Flouris[1], and Spyros Kotoulas[4]

[1] Institute of Computer Science, FORTH
[2] Department of Computer Science, University of Crete
[3] University of Huddersfield, UK
[4] IBM Research, IBM Ireland

Abstract. In this paper, we are presenting a scalable method for nonmonotonic rule-based reasoning over Semantic Web Data, using MapReduce. Our work is motivated by the recent unparalleled explosion of available data coming from the Web, sensor readings, databases, ontologies and more. Such datasets could benefit from the introduction of rule sets encoding commonly accepted rules or facts, application- or domain-specific rules, commonsense knowledge etc. This raises the question of whether, how, and to what extent knowledge representation methods are capable of handling huge amounts of data for these applications. We present a scalable MapReduce-based method for reasoning using defeasible stratified logics. Our results indicate that our method shows good scalability properties and is able to handle a benchmark dataset of 1 billion triples, bringing it on par with state-of-the-art methods for monotonic logics.

1 Introduction

Recently, we experience a significant growth of the amount of available data published on the Semantic Web. Billions of RDF triples from Wikipedia, U.S. Census, CIA World Factbook, open government sites in the US and the UK, memory organizations like the British Museum and Europeanna, as well as news and entertainment sources such as BBC, are published, along with numerous vocabularies and conceptual schemas from e-science aiming to facilitate annotation and interlinking of scientific and scholarly data [24]. The recent rising of the Linked Open Data initiative[5] [6] is an answer to the need for such large and interconnected data. RDF(S) [18, 7] has become the de facto standard for representing such knowledge in the Semantic Web, due to its flexible and extensible representation of information, which is independent of the existence or absence of a schema, under the form of triples.

The amount, diversity and interlinkage of data published in this manner enables a new generation of decision making and business intelligence applications across domains. To fully exploit the immense value of such datasets and their interconnections, one should be able to reason over them using rule sets that allow the aggregation, visualization, understanding and exploitation of the raw data. Such reasoning is based on rules which capture the RDFS or OWL inference semantics, but also rules which encode

[5] `http://linkeddata.org/`

commonsense, domain-specific, or other practical knowledge that humans possess and would allow the system to automatically reach useful conclusions based on the provided data, i.e., infer new and useful knowledge based on the data and their interconnections.

The knowledge representation field has provided a rich set of semantics and techniques to use for reasoning using such rule sets, although the focus has been on complex knowledge structures and reasoning methods. On the other hand, RDF datasets are much simpler, but their size raises scalability challenges that cannot be addressed by standard approaches. For example, as described in [29], for 78,8 million statements crawled from the Web (a small percentage of the available knowledge), the number of inferred conclusions using the relatively simple RDFS ruleset consists of 1,5 billion triples; it is evident that coping with such amounts of data is impossible in standard, single-machine approaches due to both memory and performance issues.

To address this problem, the use of massive parallelism has been recently proposed [28, 22, 29, 14, 10, 30], where reasoning is handled by a set of machines, assigning each of them a part of the parallel computation. In some cases, this approach has allowed scaling reasoning up to 100 billion triples [28]. However, such approaches have focused on monotonic reasoning, or have not been evaluated in terms of scalability [19].

In this paper, we concentrate on nonmonotonic rule sets [2, 17]. Such rule sets provide additional benefits because they are more suitable for encoding commonsense knowledge and reasoning. In addition, in the case of poor quality data, monotonic logics such as RDFS cause an explosion of trivial (and often useless derivations), as also identified in [12]. The occurrence of low quality data is very common in the context of the Semantic Web [24], as data are fetched from different sources, which are not controlled by the data engineer; thus, nonmonotonic reasoning is more suitable for this context.

Our previous works [26, 27] described how defeasible logic reasoning [16], a commonly used nonmonotonic logic, can be implemented using massively parallel techniques. In [26, 27] we adopted the MapReduce framework [8], which is widely used for parallel processing of huge datasets. In particular, we used Hadoop, an open-source implementation of the MapReduce framework, with an extensive user list including companies like IBM, Yahoo!, Facebook and Twitter[6].

The approach of [27] addressed reasoning for stratified rule sets. Stratification is a well-known concept employed in many areas of knowledge representation for efficiency reasons, e.g., in tractable RDF query answering [25], Description Logics [4, 11, 20] and nonmonotonic formalisms [5], as it has been shown to reduce the computational complexity of various reasoning problems.

This paper is the first attempt evaluating the feasibility of applying nonmonotonic reasoning over RDF data using mass parallelization techniques. We present a technique for materialization using stratified defeasible logics, based on MapReduce and focussing on performance. A defeasible rule set for the LUBM[7] benchmark is presented, which is used to evaluate our approach. We present scalability results indicating that our approach scales superlinearly with the data size. In addition, since load-balancing is a significant performance inhibitor in reasoning systems [22], we show that our ap-

[6] http://wiki.apache.org/hadoop/PoweredBy

[7] http://swat.cse.lehigh.edu/projects/lubm/

Algorithm 1 Wordcount example

```
map(Long key, String value) :
   // key: position in document
   // value: document line
   for each word w in value
      EmitIntermediate(w, "1");

reduce(String key, Iterator values) :
   // key: a word
   // values : list of counts
   int count = 0;
   for each v in values
      count += ParseInt(v);
   Emit(key , count);
```

proach performs very well in this respect for the considered dataset, distributing data fairly uniformly across MapReduce tasks. Compared to our previous work with similar content, we extend [26] by considering multi-arity predicates, and improve [27] by experimenting over a standard RDF data benchmark (LUBM).

The rest of the paper is organized as follows. Section 2 introduces briefly the MapReduce Framework and Defeasible Logic. The algorithm for defeasible reasoning using MapReduce is described in Section 3, while Section 4 presents our experimental results. We conclude in Section 5.

2 Preliminaries

2.1 MapReduce

MapReduce is a framework for parallel processing over huge datasets [8]. Processing is carried out in two phases, a map and a reduce phase. For each phase, a set of user-defined map and reduce functions are run in a parallel fashion. The former performs a user-defined operation over an arbitrary part of the input and partitions the data, while the latter performs a user-defined operation on each partition.

MapReduce is designed to operate over key/value pairs. Specifically, each Map function receives a key/value pair and emits a set of key/value pairs. All key/value pairs produced during the map phase are grouped by their key and passed to the reduce phase. During the reduce phase, a $Reduce$ function is called for each unique key, processing the corresponding set of values.

Probably the most well-known MapReduce example is the *wordcount* example. In this example, we take as input a large number of documents and the final result is the calculation of the number of occurrences of each word. The pseudo-code for the Map and $Reduce$ functions is depicted in Algorithm 1.

During the map phase, each map operation gets as input a line of a document. The Map function extracts words from each line and emits that word *w* occurred once ("1").

Here we do not use the position of each line in the document, thus the *key* in Map is ignored. However, a word can be found more than once in a line. In this case we emit a <w, 1> pair for each occurrence. Consider the line "Hello world. Hello MapReduce.". Instead of emitting a pair <Hello, 2>, our simple example emits <Hello, 1> twice (pairs for words *world* and *MapReduce* are emitted as well). As mentioned above, the MapReduce framework will group and sort pairs by their key. Specifically for the word *Hello*, a pair <Hello, <1,1>> will be passed to the $Reduce$ function. The $Reduce$ function has to sum up all occurrence values for each word emitting a pair containing the word and the final number of occurrences. The final result for the word *Hello* will be <Hello, 2>.

2.2 Defeasible Logic - Syntax

A defeasible theory [21], [3] (a knowledge base in defeasible logic) consists of five different kinds of knowledge: facts, strict rules, defeasible rules, defeaters, and a superiority relation.

Facts are literals that are treated as known knowledge (given or observed facts).

Strict rules are rules in the classical sense: whenever the premises are indisputable (e.g., facts) then so is the conclusion. An example of a strict rule is "Emus are birds", which can be written formally as: "emu(X) $\rightarrow$ bird(X).".

Defeasible rules are rules that can be defeated by contrary evidence. An example of such a rule is "Birds typically fly"; written formally: "bird(X) $\Rightarrow$ flies(X).".

Defeaters are rules that cannot be used to draw any conclusions. Their only use is to prevent some conclusions. An example is "If an animal is heavy then it might not be able to fly". Formally: "heavy(X) $\rightsquigarrow$ $\neg$flies(X).".

The *superiority relation* among rules is used to define priorities among rules, that is, where one rule may override the conclusion of another rule. For example, given the defeasible rules "r : bird(X) $\Rightarrow$ flies(X)" and "r$'$: brokenWing(X) $\Rightarrow$ $\neg$flies(X)" which contradict one another, no conclusive decision can be made about whether a bird with broken wings can fly. But if we introduce a superiority relation $>$ with r$'$ $>$ r, with the intended meaning that r$'$ is strictly stronger than r, then we can indeed conclude that the bird cannot fly.

Note that in this paper, the aforementioned term *literal* is defined strictly by the defeasible logic semantics. An RDF triple can be represented as a literal. However, considering the term *literal* as an RDF literal would be a common misunderstanding.

2.3 Defeasible Logic - Formal Definition

A rule r consists (a) of its antecedent (or body) A(r) which is a finite set of literals, (b) an arrow, and, (c) its consequent (or head) C(r) which is a literal. Given a set R of rules, we denote the set of all strict rules in R by R_s, and the set of strict and defeasible rules in R by R_{sd}. R[q] denotes the set of rules in R with consequent q. If q is a literal, $\sim$q denotes the complementary literal (if q is a positive literal p then $\sim$q is $\neg$p; and if q is $\neg$p, then $\sim$q is p)

A defeasible theory D is a triple (F,R,$>$) where F is a finite set of facts, R a finite set of rules, and $>$ a superiority relation upon R.

2.4 Defeasible Logic - Proof Theory

A conclusion of D is a tagged literal and can have one of the following four forms:

- $+\Delta$q, meaning that q is definitely provable in D.
- $-\Delta$q, meaning that q is not definitely provable in D (this does not necessarily mean that $\sim$q is definitely provable).
- $+\partial$q, meaning that q is defeasibly provable in D.
- $-\partial$q, meaning that q is not defeasibly provable in D (this does not necessarily mean that $\sim$q is defeasibly provable).

Provability is defined below. It is based on the concept of a derivation (or proof) in D = (F, R, $>$). A derivation is a finite sequence P = P(1), ..., P(n) of tagged literals satisfying the following conditions. The conditions are essentially inference rules phrased as conditions on proofs. P(1..$\imath$) denotes the initial part of the sequence P of length i. For more details on provability and an explanation of the intuition behind the conditions below, see [16].

$+\Delta$: We may append P($\imath$ + 1) = $+\Delta$q if either
q $\in$ F or
$\exists$r $\in$ R_s[q] $\forall\alpha \in$ A(r): $+\Delta\alpha \in$ P(1..$\imath$)

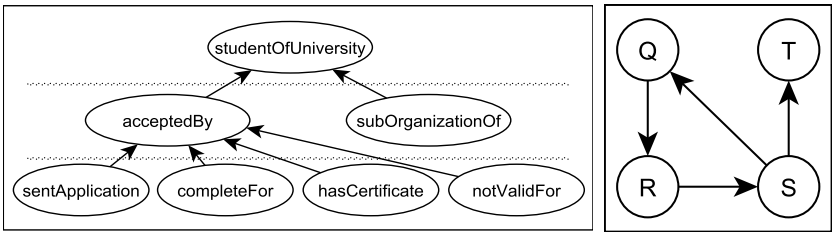
$-\Delta$: We may append P($\imath$ + 1) = $-\Delta$q if
q $\notin$ F and
$\forall$r $\in$ R_s[q] $\exists\alpha \in$ A(r): $-\Delta\alpha \in$ P(1..$\imath$)

$+\partial$: We may append P ($\imath$ + 1) = $+\partial$q if either
(1) $+\Delta$q $\in$ P(1..$\imath$) or
(2) (2.1) $\exists$r $\in$ R_{sd}[q] $\forall\alpha \in$ A(r): $+\partial\alpha \in$ P(1..$\imath$) and
(2.2) $-\Delta \sim$q $\in$ P(1..$\imath$) and
(2.3) $\forall$s $\in$ R[$\sim$q] either
(2.3.1) $\exists\alpha \in$ A(s): $-\partial\alpha \in$ P(1..$\imath$) or
(2.3.2) $\exists$t $\in$ R_{sd}[q] such that
$\forall\alpha \in$ A(t): $+\partial\alpha \in$ P(1..$\imath$) and t $>$ s

$-\partial$: We may append P($\imath$ + 1) = $-\partial$q if
(1) $-\Delta$q $\in$ P(1..$\imath$) and
(2) (2.1) $\forall$r $\in$ R_{sd}[q] $\exists\alpha \in$ A(r): $-\partial\alpha \in$ P(1..$\imath$) or
(2.2) $+\Delta \sim$q $\in$ P(1..$\imath$) or
(2.3) $\exists$s $\in$ R[$\sim$q] such that
(2.3.1) $\forall\alpha \in$ A(s): $+\partial\alpha \in$ P(1..$\imath$) and
(2.3.2) $\forall$t $\in$ R_{sd}[q] either
$\exists\alpha \in$ A(t): $-\partial\alpha \in$ P(1..$\imath$) or t $\not>$ s

3 Algorithm description

The algorithm that is described in this section, shows how parallel reasoning can be performed using the MapReduce framework. Parallel reasoning can be based either on



(a) Stratified (b) Non-stratified

Fig. 1: Predicate dependency graph

rule partitioning or on data partitioning [13]. Rule partitioning assigns the computation of each rule to a computer in the cluster. However, balanced work distribution in this case is difficult to achieve, as the computational burden per rule (and node) depends on the structure of the rule set. On the other hand, data partitioning assigns a subset of data to each computer. Data partitioning is more flexible, providing more fine-grained partitioning and allowing easier distribution among nodes in a balanced manner. Our approach is based on data partitioning.

For reasons that will be explained later, defeasible reasoning over rule sets with multi-argument predicates is based on the dependencies between predicates which is encoded using the *predicate dependency graph*. Thus, rule sets can be divided into two categories: *stratified* and *non-stratified*. Intuitively, a *stratified* rule set can be represented as an acyclic hierarchy of dependencies between predicates, while a *non-stratified* cannot. We address the problem for stratified rule sets by providing a well-defined reasoning sequence, and explain at the end of the section the challenges for non-stratified rule sets.

The dependencies between predicates can be represented using a *predicate dependency graph*. For a given rule set, the *predicate dependency graph* is a directed graph whose:

- vertices correspond to predicates. For each literal p, both p and $\neg p$ are represented by the positive predicate.
- edges are directed from a predicate that belongs to the body of a rule, to a predicate that belongs to the head of the same rule. Edges are used for all three rule types (strict rules, defeasible rules, defeaters).

Stratified rule sets (correspondingly, *non-stratified rule sets*) are rule sets whose predicate dependency graph is acyclic (correspondingly, contains a cycle). *Stratified theories* are theories based on stratified rule sets. Figure 1a depicts the predicate dependency graph of a stratified rule set, while Figure 1b depicts the predicate dependency graph of a non-stratified rule set. The superiority relation is not part of the graph.

As an example of a stratified rule set, consider the following:

r1: X *sentApplication* A, A *completeFor* D $\Rightarrow$ X *acceptedBy* D.
r2: X *hasCertificate* C, C *notValidFor* D $\Rightarrow$ X $\neg$*acceptedBy* D.
r3: X *acceptedBy* D, D *subOrganizationOf* U $\Rightarrow$ X *studentOfUniversity* U.
r1 $>$ r2.

The predicate dependency graph for the above rule set is depicted in Figure 1a. The predicate graph can be used to determine strata for the different predicates. In particular, predicates (nodes) with no outgoing edges are assigned the maximum stratum, which is equal to the maximum depth of the directed acyclic graph (i.e., the size of the maximum path that can be defined through its edges), say k. Then, all predicates that are connected with a predicate of stratum k are assigned stratum $k-1$, and the process continues recursively until all predicates have been assigned some stratum. Note that predicates are reassigned to a lower stratum in case of multiple dependencies. The dashed horizontal lines in Figure 1a are used to separate the various strata, which, in our example, are as follows:

Stratum 2: *studentOfUniversity*
Stratum 1: *acceptedBy*, *subOrganizationOf*
Stratum 0: *sentApplication*, *completeFor*, *hasCertificate*, *notValidFor*

Stratified theories are often called decisive in the literature [5].

Proposition 1. *[5] If D is stratified, then for each literal p:*
(a) either $D \vdash +\Delta p$ *or* $D \vdash -\Delta p$
(b) either $D \vdash +\partial p$ *or* $D \vdash -\partial p$

Thus, there are three possible states for each literal p in a stratified theory: (a) $+\Delta p$ *and* $+\partial p$, (b) $-\Delta p$ *and* $+\partial p$ and (c) $-\Delta p$ *and* $-\partial p$.

Reasoning is based on facts. According to defeasible logic algorithm, facts are $+\Delta$ and every literal that is $+\Delta$, is $+\partial$ too. Having $+\Delta$ and $+\partial$ in our initial knowledge base, it is convenient to store and perform reasoning only for $+\Delta$ and $+\partial$ predicates.

This representation of knowledge allows us to reason and store provability information regarding various facts more efficiently. In particular, if a literal is not found as a $+\Delta$ (correspondingly, $+\partial$) then it is $-\Delta$ (correspondingly, $-\partial$). In addition, stratified defeasible theories have the property that if we have computed all the $+\Delta$ and $+\partial$ conclusions up to a certain stratum, and a rule whose body contains facts of said stratum does not currently fire, then this rule will also be inapplicable in subsequent passes; this provides a well-defined reasoning sequence, namely considering rules from lower to higher strata.

3.1 Reasoning overview

During reasoning we will use the representation (<fact, ($+\Delta$,$+\partial$)>) to store our inferred facts. We begin by transforming the given facts, in a single MapReduce pass, into (<fact, ($+\Delta$,$+\partial$)>).

Now let us consider for example the facts "John *sentApplication* App", "App *completeFor* Dep", "John *hasCertificate* Cert", "Cert *notValidFor* Dep" and "Dep *subOrganizationOf* Univ" . The *initial pass* on these facts using the aforementioned rule set will create the following output:

<John *sentApplication* App, ($+\Delta$,$+\partial$)> <App *completeFor* Dep, ($+\Delta$,$+\partial$)>
<John *hasCertificate* Cert, ($+\Delta$,$+\partial$)> <Cert *notValidFor* Dep, ($+\Delta$,$+\partial$)>
<Dep *subOrganizationOf* Univ, ($+\Delta$,$+\partial$)>

No reasoning needs to be performed for the lowest stratum (stratum 0) since these predicates (*sentApplication*, *completeFor*, *hasCertificate*, *notValidFor*) do not belong to the head of any rule. As is obvious by the definition of $+\partial$, $-\partial$, defeasible logic introduces uncertainty regarding inference, because certain facts/rules may "block" the firing of other rules. This can be prevented if we reason for each stratum separately, starting from the lowest stratum and continuing to higher strata. This is the reason why for a hierarchy of N strata we have to perform $N - 1$ times the procedure described below. In order to perform defeasible reasoning we have to run two passes for each stratum. The first pass computes which rules can fire. The second pass performs the actual reasoning and computes for each literal if it is definitely or defeasibly provable. The reasons for both decisions (reasoning sequence and two passes per stratum) are explained in the end of the next subsection.

3.2 Pass #1: Fired rules calculation

During the first pass, we calculate the inference of fired rules, which is performed by joining predicates on common argument values. Such techniques for basic and multi-way join have been described in [9] and optimized in [1]. In order to achieve an efficient implementation, optimizations in [1] should be taken into consideration. Here we elaborate on our approach for basic joins and explain at the end of the subsection how it can be generalized for multi-way joins.

Basic join is performed on common argument values. Consider the following rule:

r1: X *sentApplication* A, A *completeFor* D $\Rightarrow$ X *acceptedBy* D.

The key observation is that "X *sentApplication* A" and "A *completeFor* D" can be joined on their common argument *A*. Based on this observation, during the Map operation, we emit pairs of the form <A, (X, *sentApplication*)> for predicate *sentApplication* and <A, (D, *completeFor*)> for predicate *completeFor*. The idea is to join *sentApplication* and *completeFor* only for literals that have the same value on argument *A*. During the $Reduce$ operation we combine *sentApplication* and *completeFor* producing *acceptedBy*.

In our example, the facts "John *sentApplication* App" and "App *completeFor* Dep" will cause Map to emit <App, (John, *sentApplication*)> and <App, (Dep, *completeFor*)>. The MapReduce framework groups and sorts intermediate pairs passing <App, <(John, *sentApplication*), (Dep, *completeFor*)>> to the $Reduce$ operation. Finally, at $Reduce$ we combine given values and infer "John *acceptedBy* Dep".

To support defeasible logic rules which have blocking rules, this approach must be extended. We must record all fired rules prior to any conclusion inference, whereas for monotonic logics this is not necessary, and conclusion derivation can be performed immediately. The reason why this is so is explained at the end of the subsection. Pseudo-code for Map and $Reduce$ functions, for a basic join, is depicted in Algorithm 2. Map function reads input of the form <literal, ($+\Delta$, $+\partial$)> or <literal, ($+\partial$)> and emits pairs of the form <matchingArgumentValue, (nonMatchingArgumentValue, *Predicate*, $+\Delta$, $+\partial$)> or <matchingArgumentValue, (nonMatchingArgumentValue, *Predicate*, $+\partial$)> respectively.

Algorithm 2 Fired rules calculation

```
map(Long key, String value):
  // key: position in document (irrelevant)
  // value: document line (derived conclusion)
  For every common argumentValue in value
    EmitIntermediate(argumentValue, value);

reduce(String key, Iterator values):
  // key: matching argument
  // value: literals for matching
  For every argument value match in values
    If strict rule fired with all premises being +Δ then
      Emit(firedLiteral, "[¬,] +Δ, +∂, ruleID");
    else
      Emit(firedLiteral, "[¬,] +∂, ruleID");
```

Now consider again the stratified rule set described in the beginning of the section, for which the *initial pass* will produce the following output:

<John *sentApplication* App, ($+\Delta$,$+\partial$)> <App *completeFor* Dep, ($+\Delta$,$+\partial$)>
<John *hasCertificate* Cert, ($+\Delta$,$+\partial$)> <Cert *notValidFor* Dep, ($+\Delta$,$+\partial$)>
<Dep *subOrganizationOf* Univ, ($+\Delta$,$+\partial$)>

We perform reasoning for stratum 1, so we will use as premises all the available information for predicates of stratum 0. The Map function will emit the following pairs:

<App, (John,*sentApplication*,$+\Delta$,$+\partial$)> <App, (Dep,*completeFor*,$+\Delta$,$+\partial$)>
<Cert, (John,*hasCertificate*,$+\Delta$,$+\partial$)> <Cert, (Dep,*notValidFor*,$+\Delta$,$+\partial$)>

The MapReduce framework will perform grouping/sorting resulting in the following intermediate pairs:

<App, <(John,*sentApplication*,$+\Delta$,$+\partial$), (Dep,*completeFor*,$+\Delta$,$+\partial$)>>
<Cert, <(John,*hasCertificate*,$+\Delta$,$+\partial$), (Dep,*notValidFor*,$+\Delta$,$+\partial$)>>

During reduce we combine premises in order to emit the *firedLiteral* which consists of the fired rule head predicate and the *nonMatchingArgumentValue* of the premises. However, inference depends on the type of the rule. In general, for all three rule types (strict rules, defeasible rules and defeaters) if a rule fires then we emit as output <firedLiteral, ([¬,] $+\partial$, ruleID)> ([¬,] denotes that "¬" is optional and appended only if the *firedLiteral* is negative). However, there is a special case for strict rules. This special case covers the required information for $+\Delta$ conclusions inference. If all premises are $+\Delta$ then we emit as output <firedLiteral, ([¬,]$+\Delta$,$+\partial$,ruleID)> instead of <firedLiteral, ([¬,]$+\partial$, ruleID)>.

For example, during the reduce phase the reducer with key:

App will emit <John *acceptedBy* Dep, ($+\partial$, r1)>
Cert will emit <John *acceptedBy* Dep, ($\neg$,$+\partial$, r2)>

As we see here, "John *acceptedBy* Dep" and "John $\neg$*acceptedBy* Dep" are computed by different reducers (with key *App* and *Cert* respectively) which do not communicate with each other. Thus, none of the two reducers has all the available information in order to perform defeasible reasoning. Therefore, we need a second pass for the reasoning.

Let us illustrate why reasoning has to be performed for each stratum separately, requiring stratified rule sets. Consider again our running example. We will attempt to perform reasoning for all the strata simultaneously. On the one hand, we cannot join "John *acceptedBy* Dep" with "Dep *subOrganizationOf* Univ" prior to the second pass because we do not have a final conclusion on "John *acceptedBy* Dep". Thus, we will not perform reasoning for "John *studentOfUniversity* Univ" during the second pass, which leads to data loss. On the other hand, if another rule (say r4) supporting "John $\neg$*studentOfUniversity* Univ" had also fired, then during the second pass, we would have mistakenly inferred "John $\neg$*studentOfUniversity* Univ", leading our knowledge base to inconsistency.

In case of multi-way joins we compute the head of the rule (*firedLiteral*) by performing joins, on common argument values, in one or more MapReduce passes as explained in [9] and [1]. As above, for each fired rule, we must take into consideration the type of the rule and whether all the premises are $+\Delta$ or not. Finally, the format of the output remains the same (<firedLiteral, ([$\neg$,] $+\Delta$, $+\partial$, ruleID)> or <firedLiteral, ([$\neg$,] $+\partial$, ruleID)>).

3.3 Pass #2: Defeasible reasoning

We proceed with the second pass. Once fired rules are calculated, a second MapReduce pass performs reasoning for each literal separately. We should take into consideration that each literal being processed could already exist in our knowledge base (due to the *initial pass*). In this case, we perform duplicate elimination by not emitting pairs for existing conclusions. The pseudo-code for Map and $Reduce$ functions, for stratified rule sets, is depicted in Algorithm 3.

After both *initial pass* and fired rules calculation (first pass), our knowledge base will consist of:

<John *sentApplication* App, ($+\Delta$,$+\partial$)> <App *completeFor* Dep, ($+\Delta$,$+\partial$)>
<John *hasCertificate* Cert, ($+\Delta$,$+\partial$)> <Cert *notValidFor* Dep, ($+\Delta$,$+\partial$)>
<Dep *subOrganizationOf* Univ, ($+\Delta$,$+\partial$)> <John *acceptedBy* Dep, ($+\partial$, r1)>
<John *acceptedBy* Dep, ($\neg$,$+\partial$, r2)>

During the Map operation we must first extract from $value$ the literal and the inferred knowledge or the fired rule using *extractLiteral()* and *extractKnowledge()* respectively. For each literal p, both p and $\neg p$ are sent to the same reducer. The "$\neg$" in *knowledge* distinguishes p from $\neg p$. The Map function will emit the following pairs:

<Dep *subOrganizationOf* Univ, ($+\Delta$,$+\partial$)> <John *acceptedBy* Dep, ($+\partial$, r1)>
<John *acceptedBy* Dep, ($\neg$,$+\partial$, r2)>

Algorithm 3 Defeasible reasoning

```
map(Long key, String value) :
  // key: position in document (irrelevant)
  // value: inferred knowledge/fired rules
  String p = extractLiteral(value);
  If p does not belong to current stratum then
    return;
  String knowledge = extractKnowledge(value);
  EmitIntermediate(p, knowledge);

reduce(String p, Iterator values) :
  // p: a literal
  // values : inferred knowledge/fired rules
  For each value in values
    markKnowledge(value);
  For literal in {p, ¬p} check
    If literal is already +Δ then
      return;
    Else If strict rule fired with all premises being +Δ then
      Emit(literal, "+Δ, +∂");
    Else If literal is +∂ after defeasible reasoning then
      Emit(literal, "+∂");
```

MapReduce framework will perform grouping/sorting resulting in the following intermediate pairs:

<Dep *subOrganizationOf* Univ, ($+\Delta$,$+\partial$)>
<John *acceptedBy* Dep, <($+\partial$, r1), ($\neg$,$+\partial$, r2)>>

For the Reduce, the key contains the literal and the values contain all the available information for that literal (known knowledge, fired rules). We traverse over *values* marking known knowledge and fired rules using the *markKnowledge()* function. Subsequently, we use this information in order to perform reasoning for each literal.

During the reduce phase the reducer with key:

"Dep *subOrganizationOf* Univ" will not emit anything
"John *acceptedBy* Dep" will emit <John *acceptedBy* Dep, ($+\partial$)>

The literal "Dep *subOrganizationOf* Univ" is known knowledge. For known knowledge a potential duplicate elimination must be performed. We reason simultaneously both for "John *acceptedBy* Dep" and "John $\neg$*acceptedBy* Dep". As "John $\neg$*acceptedBy* Dep" is $-\partial$, it does not need to be recorded.

3.4 Final remarks

The total number of MapReduce passes is independent of the size of the given data, and is determined by the form of the rules, in particular by the number of strata that the

rules are stratified into. As mentioned in subsection 3.2, performing reasoning for each stratum separately eliminates data loss and inconsistency, thus our approach is sound and complete since we fully comply with the defeasible logic provability. Eventually, our knowledge base consists of $+\Delta$ and $+\partial$ literals.

The situation for non-stratified rule sets is more complex. Reasoning can be based on the algorithm described in [15], performing reasoning until no new conclusion is derived. However, the total number of required passes is generally unpredictable, depending both on the given rule set and the data distribution. Additionally, an efficient mechanism for "$\forall$r $\in$ R_s[q] $\exists\alpha \in$ A(r): $-\Delta\alpha \in$ P(1..i)" (in $-\Delta$ provability) and "$\forall$r $\in$ R_{sd}[q] $\exists\alpha \in$ A(r): $-\partial\alpha \in$ P(1..i)" (in 2.1 of $-\partial$ provability) computation is yet to be defined because all the available information for the literal must be processed by a single node (since nodes do not communicate with each other), causing either main memory insufficiency or skewed load balancing decreasing the parallelization. Finally, we have to reason for and store every possible conclusion ($+\Delta$,$-\Delta$,$+\partial$,$-\partial$), producing a significantly larger stored knowledge base.

4 Evaluation

In this Section, we are presenting the methodology, dataset and experimental results for an implementation of our approach using Hadoop.

Methodology. Our evaluation is centered around scalability and the capacity of our system to handle large datasets. In line with standard practice in the field of high-performance systems, we have defined scalability as the ability to process datasets of increasing size in a proportional amount of time and the ability of our system to perform well as the computational resources increase. With regard to the former, we have performed experiments using datasets of various sizes (yet similar characteristics).

With regard to scaling computational resources, it has been empirically observed that the main inhibitor of parallel reasoning systems has been load-balancing between compute nodes [14]. Thus, we have also focused our scalability evaluation on this aspect.

The communication model of Hadoop is not sensitive to the physical location of each data partition. In our experiments, Map tasks only use local data (implying very low communication costs) and Reduce operates using hash-partitioning to distribute data across the cluster (resulting in very high communication costs regardless of the distribution of data and cluster size). In this light, scalability problems do not arise by the number of compute nodes, but by the unequal distribution of the workload in each reduce task. As the number of compute nodes increases, this unequal distribution becomes visible and hampers performance.

Dataset. We have used the most popular benchmark for reasoning systems, LUBM. LUBM allows us to scale the size of the data to an arbitrary size while keeping the reasoning complexity constant. For our experiments, we generated up to 8000 universities resulting in approximately 1 billion triples.

Rule set. The logic of LUBM can be partially expressed using RDFS and OWL2-RL. Nevertheless, neither of these logics are defeasible. Thus, to evaluate our system, we have created the following ruleset:

r1: X rdf:type FullProfessor → X rdf:type Professor.
r2: X rdf:type AssociateProfessor → X rdf:type Professor.
r3: X rdf:type AssistantProfessor → X rdf:type Professor.
r4: P publicationAuthor X, P publicationAuthor Y → X commonPublication Y.
r5: X teacherOf C, Y takesCourse C → X teaches Y.
r6: X teachingAssistantOf C, Y takesCourse C → X teaches Y.
r7: X commonPublication Y → X commonResearchInterests Y.
r8: X hasAdvisor Z, Y hasAdvisor Z → X commonResearchInterests Y.
r9: X hasResearchInterest Z, Y hasResearchInterest Z → X commonResearchInterests Y.
r10: X hasAdvisor Y ⇒ X canRequestRecommendationLetter Y.
r11: Y teaches X ⇒ X canRequestRecommendationLetter Y.
r12: Y teaches X, Y rdf:type PostgraduateStudent ⇒ X ¬canRequestRecommendationLetter Y.
r13: X rdf:type Professor, X worksFor D, D subOrganizationOf U ⇒ X canBecomeDean U.
r14: X rdf:type Professor, X headOf D, D subOrganizationOf U ⇒ X ¬canBecomeDean U.
r15: X worksFor D ⇒ X canBecomeHeadOf D.
r16: X worksFor D, Z headOf D, X commonResearchInterests Z ⇒ X ¬canBecomeHeadOf D.
r17: Y teaches X ⇒ X suggestAdvisor Y.
r18: Y teaches X, X hasAdvisor Z ⇝ X ¬suggestAdvisor Y.
r12 > r11, r14 > r13, r16 > r15, r18 > r17.

MapReduce jobs description. We need 8 jobs in order to perform reasoning on the above rule set. The *first* job is the *initial pass* described in Section 3 (which we also use to compute rules r1-r3). For the rest of the jobs, we first compute fired rules and then perform reasoning for each stratum separately. The *second* job computes rules r4-r6. During the *third* job we perform duplicate elimination, since r4-r6 are strict rules. We compute rules r7-r14 during the *fourth* job while reasoning on them, is performed during the *fifth* job. Jobs *six* and *seven* compute rules r15-r18. Finally, during the *eighth* job we perform reasoning on r15-r18, finishing the whole procedure.

Platform. Our experiments were performed on a IBM x3850 server with 40 cores and 750GB of RAM, connected to a XIV Storage Area Network (SAN), using a 10Gbps storage switch. We have used IBM Hadoop Cluster v1.3, which is compatible with Hadoop v0.20.2, along with an optimization to reduce Map task overhead, in line with [23]. Although our experiments were run on a single machine, there was no direct communication between processes and all data was transferred through persistent storage. We have used a number of Mappers and Reducers equal to the number of cores in the system (i.e. 40).

Results. Figure 2 shows the runtimes of our system for varying input sizes. We make the following observations: (a) even for a single node, our system is able to handle very large datasets, easily scaling to 1 billion triples. (b) The scaling properties with regard to dataset size are excellent: in fact, as the size of the input increases, the throughput of our system increases. For example, while our system can process a dataset of 125 million triples at a throughput of 27Ktps, for 1 billion triples, the throughput becomes 63Ktps. This is attributed to the fact that job startup costs are amortized over the longer runtime of the bigger datasets.

The above show that our system is indeed capable of achieving high performance and scales very well with the size of the input. Nevertheless, to further investigate how

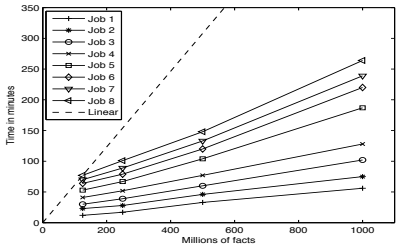
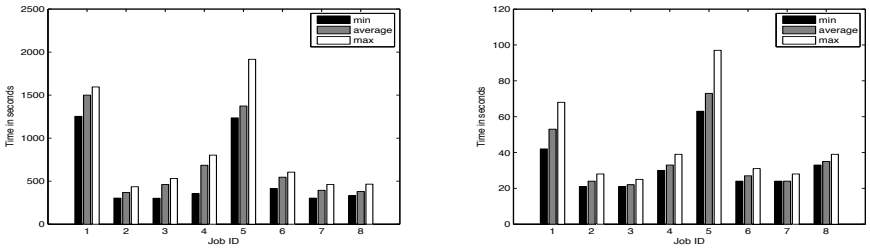


Fig. 2: Runtime in minutes for various datasets, and projected linear scalability. Job runtimes are stacked (i.e. runtime for Job 8 includes the runtimes for Jobs 1-7).



(a) 40 reduce tasks

(b) 400 reduce tasks

Fig. 3: Minimum, average and maximum reduce task runtime for each job.

our system would perform when the data size precludes the use of a single machine, it is critical to examine the load-balancing properties of our algorithm.

As previously described, in typical MapReduce applications, load-balancing problems arise during the reduce phase. Namely, it is possible that the partitions of the data processed in a single reduce task vary widely in terms of compute time required. This is a potential scalability bottleneck. To test our system for such issues, we have launched an experiment where we have increased the number of reduce tasks to 400. We can expect that, if the load balance for 400 reduce tasks is relatively uniform, our system is able to scale at least to that size.

Figure 3 shows the load balance between different reduce tasks, for 1 billion triples and 40 (Figure 3a) or 400 (Figure 3b) reduce tasks. In principle, an application performs badly when a single task dominates the runtime, since all other tasks would need to wait for it to finish. In our experiments, it is evident that no such task exists. In addition, one may note that the system is actually faster with 400 reduce tasks. This is attributed both to the fact that each core in our platform can process two threads in parallel, and to implementation aspects of Hadoop that result in tasks, processing approximately 1GB, demonstrating higher throughput than larger tasks.

Although a direct comparison is not meaningful, the throughput of our system is in line with results obtained when doing monotonic reasoning using state of the art RDF stores and inference engine. For example, OWLIM claims a 14.4-hour loading time

for the same dataset when doing OWL horst inference [8]. WebPIE [28], which is also based on MapReduce, presents an OWL-horst inference time of 35 minutes, albeit on 64 lower-spec nodes and requiring an additional dictionary encoding step.

Given the significant overhead of nonmonotonic reasoning, and in particular, the fact that inferences can not be drawn directly, this result is counter-intuitive. The key to the favorable performance of our approach is that the "depth" of the reasoning is fixed, on a per rule set basis. The immediate consequence is that the number of MapReduce jobs, which bear significant startup costs, is also fixed. In other words, the "predictable" nature of stratified logics allows us to have less complicated relationships between facts in the system.

Finally, we should take into consideration the fact that LUBM produces fairly uniform data. Although there is significant skew in LUBM (e.g. in the frequency of terms such as rdf:type), the rule set that we have used in the evaluation does not perform joins on such highly skewed terms. However, our previous work [27] shows that our approach can cope with highly skewed data, which follow a zipf distribution.

5 Conclusion and Future Work

In this paper we studied the feasibility of nonmonotonic rule systems over large volumes of semantic data. In particular, we considered defeasible reasoning over RDF, and ran experiments over RDF data. Our results demonstrate that such reasoning scales very well. In particular, we have shown that nonmonotonic reasoning is not only possible, but can compete with state-of-the-art monotonic logics. To the best of our knowledge, this is the first study demonstrating the feasibility of inconsistency-tolerant reasoning over RDF data using mass parallelization techniques.

In future work, we intend to perform an extensive experimental evaluation in order to verify our results for different input dataset morphologies. In addition, we plan to apply the MapReduce framework to ontology dynamics (including evolution, diagnosis, and repair) approaches based on validity rules (integrity constraints). These problems are closely related to inconsistency-tolerant reasoning, as violation of constraints may be viewed as a logical inconsistency.

6 Acknowledgments

This work was partially supported by the PlanetData NoE (FP7:ICT-2009.3.4, #257641).

References

1. Afrati, F.N., Ullman, J.D.: Optimizing joins in a mapreduce environment. In: EDBT (2010)
2. Antoniou, G., van Harmelen, F.: A Semantic Web Primer, 2nd Edition. The MIT Press, 2 edn. (March 2008)
3. Antoniou, G., Williams, M.A.: Nonmonotonic reasoning. MIT Press (1997)

[8] http://www.ontotext.com/owlim/benchmark-results/lubm

4. Baader, F., Kosters, R.: Nonstandard Inferences in Description Logics: The Story So Far. Mathematical Problems from Applied Logic I, volume 4 of International Mathematical Series (2006)
5. Billington, D.: Defeasible Logic is Stable. J. Log. Comput. 3(4), 379–400 (1993)
6. Bizer, C., Heath, T., Berners-Lee, T.: Linked Data - The Story So Far. Int. J. Semantic Web Inf. Syst. 5(3), 1–22 (2009)
7. Brickley, D., Guha, R.: RDF Vocabulary Description Language 1.0: RDF Schema. www.w3.org/TR/2004/REC-rdf-schema-20040210 (2004)
8. Dean, J., Ghemawat, S.: MapReduce: simplified data processing on large clusters
9. Fische, F.: Investigation & Design for Rule-based Reasoning. Tech. rep., LarKC (2010)
10. Goodman, E.L., Jimenez, E., Mizell, D., Al-Saffar, S., Adolf, B., Haglin, D.J.: High-Performance Computing Applied to Semantic Databases. In: ESWC (2). pp. 31–45 (2011)
11. Haase, C., Lutz, C.: Complexity of Subsumption in the EL Family of Description Logics: Acyclic and Cyclic TBoxes. In: ECAI-08. pp. 25–29 (2008)
12. Hogan, A., Harth, A., Polleres, A.: Scalable Authoritative OWL Reasoning for the Web. Int. J. Semantic Web Inf. Syst. 5(2), 49–90 (2009)
13. Kotoulas, S., van Harmelen, F., Weaver, J.: KR and Reasoning on the Semantic Web: Web-Scale Reasoning (2011)
14. Kotoulas, S., Oren, E., van Harmelen, F.: Mind the data skew: distributed inferencing by speeddating in elastic regions. In: WWW. pp. 531–540 (2010)
15. Maher, M.J., Rock, A., Antoniou, G., Billington, D., Miller, T.: Efficient Defeasible Reasoning Systems. IJAIT 10, 2001 (2001)
16. Maher, M.J.: Propositional Defeasible Logic has Linear Complexity. CoRR cs.AI/0405090 (2004)
17. Maluszynski, J., Szalas, A.: Living with Inconsistency and Taming Nonmonotonicity. In: Datalog. pp. 384–398 (2010)
18. Manola, F., Miller, E., McBride, B.: RDF Primer. www.w3.org/TR/rdf-primer (2004)
19. Mutharaju, R., Maier, F., Hitzler, P.: A MapReduce Algorithm for EL+. In: Description Logics (2010)
20. Nebel, B.: Terminological Reasoning is Inherently Intractable. Artificial Intelligence 43, 235–249 (1990)
21. Nute, D.: Defeasible Logic. In: Handbook of Logic in Artificial Intelligence and Logic Programming-Nonmonotonic Reasoning and Uncertain Reasoning(Volume 3) (1994)
22. Oren, E., Kotoulas, S., Anadiotis, G., Siebes, R., ten Teije, A., van Harmelen, F.: Marvin: Distributed reasoning over large-scale Semantic Web data. J. Web Sem. 7(4), 305–316 (2009)
23. R. Vernica, A. Balmin, K.B., Ercegovac, V.: Adaptive Mapreduce using Situation-Aware Mappers. In: EDBT
24. Roussakis, Y., Flouris, G., Christophides, V.: Declarative Repairing Policies for Curated KBs. In: HDMS (2011)
25. Serfiotis, G., Koffina, I., Christophides, V., Tannen, V.: Containment and Minimization of RDF(S) Query Patterns. In: ISWC-05 (2005)
26. Tachmazidis, I., Antoniou, G., Flouris, G., Kotoulas, S.: Towards Parallel Nonmonotonic Reasoning with Billions of Facts. In: KR-12 (2012)
27. Tachmazidis, I., Antoniou, G., Flouris, G., Kotoulas, S., McCluskey, L.: Large-scale Parallel Stratified Defeasible Reasoning. In: ECAI-12 (2012)
28. Urbani, J., Kotoulas, S., Maassen, J., van Harmelen, F., Bal, H.E.: OWL reasoning with webPIE: Calculating the Closure of 100 Billion Triples. In: ESWC (1). pp. 213–227 (2010)
29. Urbani, J., Kotoulas, S., Oren, E., van Harmelen, F.: Scalable Distributed Reasoning Using MapReduce. In: ISWC. pp. 634–649 (2009)
30. Weaver, J., Hendler, J.A.: Parallel materialization of the finite rdfs closure for hundreds of millions of triples. In: International Semantic Web Conference. pp. 682–697 (2009)

A Scalability Metric for Parallel Computations on Large, Growing Datasets (like the Web)

Jesse Weaver

Tetherless World Constellation, Rensselaer Polytechnic Institute, Troy, NY, USA

Abstract. One of the greatest challenges facing computations on data crawled from the Web is the (in)ability to scale to such large quantities of data. While some computations are less challenged by this than others, inference on the Semantic Web is certainly limited in this regard. Parallelism has been employed to scale inference to larger datasets, but evaluations of recent works have fallen back on common parallel computing metrics that do not apply to this specific scalability challenge. In this position paper, the name *data scaling* is given to this scalability challenge, and the metric *growth efficiency* is defined.

1 Introduction

One of the greatest challenges facing computations on data crawled from the Web is the (in)ability to scale to such large quantities of data. Parallelism is often employed to scale computation to larger datasets while keeping execution time reasonable. However, traditional parallel computing metrics focus on one of two forms of scaling: *strong scaling* or *weak scaling*. The goal of strong scaling is to reduce execution time for a fixed problem by adding processors. The goal of weak scaling is to keep execution time constant by adding processors to accommodate additional workload (not necessarily proportional to amount of data). In contrast to these notions of scaling, a new notion of scalability – *data scaling* – is introduced herein, and it is concerned with keeping execution time constant by adding processors to accommodate more *data*.

The ideas presented in this position paper are preliminary in nature, still requiring formal development. The specific intent of this paper, though, is to instigate a paradigm shift in the way scalability evaluations of parallel inference (and possibly other problems) on large, RDF datasets are performed.

In section 2, the need for data scaling is motivated. In section 3, common scalability metrics are shown to be unfit for measuring data scalability, and a new data scaling metric – *growth efficiency* – is defined. Discussion about, and a retroactive example of, evaluating a system with growth efficiency is given in section 4, and conclusion is given in section 5.

2 Motivation: data scaling

This work subscribes to the following statement by Hitzler and van Harmelen: "Concerning scalability, reasoning systems have made major leaps in the recent

past .... However, it remains an open question when (and if[1]) these approaches will scale to the size of the web, ..." [5]. From this statement, two assumptions are inferred which are used to motivate the work presented herein.

1. It is important for reasoning (including inference) systems to scale toward the size of the Web.
2. The Web is continuously growing.

Perhaps these assumptions are debatable, but for the intents and purposes of this paper, they are considered axiomatic. Additionally, although inference motivates this work, the definitions and their application are not specific to any particular class of problems.

To support definitions throughout this paper, a simplistic, intentionally non-specific computational model is assumed. A (terminating) parallel computation is performed on some dataset, in some amount of time, utilizing some number of processors. This is sufficient for the following discussion.

Indeed, progress has been made for specific forms of reasoning on large datasets. At the very least, varying degrees of RDFS and OWL inference have been achieved on real-world datasets containing around a billion RDF triples [6, 8, 11, 14]. However, in 2010, linked RDF data on the Web consisted of over 24.7 billion RDF triples, and the amount of such data appears to be rapidly growing [1]. Therefore, even if effective inference could be demonstrated on the entire body of RDF data on the (current) Web, it would likely need to scale to even larger datasets in the future.

Evaluations of recently studied, parallel inference approaches [2, 6–13] give no direct indication of how well the approaches will scale to dataset sizes beyond those used in the evaluations. This seems to be due to reliance on scalability metrics traditionally used in parallel computing that do not apply to the challenge of large and growing datasets. Therefore, there is a need to explicitly name and define this scalability issue, and to provide relevant metrics for it.

Data scaling *is concerned with the change in execution time of a parallel computation as processors are added to accommodate larger datasets.* Data scaling is arguably the central scalability issue for parallel computations on the Web, and it is distinct from strong scaling and weak scaling as illustrated through metrics in the following section.

3 Common scalability metrics and growth efficiency

This section contains a brief review of fundamental metrics often used for measuring scalability (in some sense) of parallel systems. These are defined herein in an atypical way in order to relate the metrics to dataset size, thus highlighting their insufficiency for measuring data scalability. Then a new metric for data scaling is introduced. To aide this discussion, the following definition is needed.

[1] "Since the web keeps growing, they may never scale, even if they become much more efficient." Footnote in quote from [5]. Footnote number appearing in the quote herein differs from the number used in [5].

Definition 1. *A **growing dataset** is effectively a function D that maps positive integers to datasets such that for any positive integer n, $|D(n)| = n$ and $D(n) \subset D(n+1)$.*

Relative speedup and metrics based on it are the scalability metrics reported by nearly every recent work [2, 8–12]. Others report only (variants of) execution time [6, 7, 13].

Definition 2. *Let D be a growing dataset, and fix k to some positive integer. Let T_1 be the execution time for one processor with dataset $D(k)$, and let T_P be the execution time for P processors with dataset $D(k)$. Then the **relative speedup** is defined as $S_P = T_1/T_P$.*

Clearly, relative speedup (the most common strong scaling metric) gives no direct indication of data scaling since the dataset is fixed. As an alternative, one may resort to (empirical) scaled speedup [4] (the most common weak scaling metric).

Definition 3. *Let D be a growing dataset, and fix k to some positive integer. Let t_1 be the execution time for one processor with dataset $D(k)$. Let $i > k$ be a positive integer such that the execution time for P processors with dataset $D(i)$ is t_1. Then let t_P be the execution time for one processor with dataset $D(i)$. Then the **scaled speedup** is defined as $\mathcal{S}_P = t_P/t_1$.*

Unlike with relative speedup, in scaled speedup, the dataset size changes with the number of processors. However, processors are added not to accommodate more data but rather to keep execution time constant.

Therefore, these metrics (and those metrics derived from them) are unsuited for measuring data scalability, and a new metric is needed.

Definition 4. *Let D be a growing dataset, and fix k to some positive integer. Let $\mathcal{T}_1$ be the execution time for one processor with dataset $D(k)$, and let $\mathcal{T}_P$ be the execution time for P processors with dataset $D(P \cdot k)$. Then **growth efficiency** is defined as $\mathcal{G}_P = \mathcal{T}_1/\mathcal{T}_P$.*

Growth efficiency directly aligns with the notion of data scaling. The idea is that the size of the input dataset should grow linearly with the number of processors, as captured in the definition. Thus, processors are added to accommodate more data. Growth efficiency is more comparable to efficiency $E_P = S_P/P$ or scaled efficiency $\mathcal{E}_P = \mathcal{S}_P/P$ in that it is a value between zero and one[2] (inclusively).

4 Evaluations using growth efficiency

Performance evaluations using growth efficiency are fairly intuitive, but there are some important details, particularly when the evaluation is meant to compare systems.

[2] That is, in theory. Although undetermined at present, there may exist some conditions in practice that allow for growth efficiency to be greater than one. This would be akin to superlinear speedup in which efficiency can be greater than one.

Points $\langle x, y \rangle$ in scatter plots should be such that x is the dataset size and y is the growth efficiency. Using notation from definition 4, the points are $\langle P \cdot k, \mathcal{G}_P \rangle$ for some k and various P. This brings up the issue of what k should be. k is the amount of data for a single processor, and as such, it is referred to herein as the *processor capacity*. Processor capacity can be defined in numerous ways. One possibility is availability of space (e.g., RAM, disk); another possibility is the maximum amount of data a single processor can handle without exceeding a specific upper bound on execution time. Regardless, an evaluation should include discussion and justification of how processor capacity is determined.

It is often the case that evaluations of different systems are performed independently of each other, and some meaningful comparison is retroactively sought. Thus, consideration must be given to potential differences in choice of growing dataset and notion of processor capacity. That is, evaluations are comparable only in as much as the parameters of the evaluations are comparable.

Growing datasets should be similar, if not the same. Meeting this requirement is straightforward with synthetic datasets, but more difficult with real-world datasets. Unless it is obvious, an evaluation should make clear the method by which the dataset was linearly "grown" with number of processors. It is conceivable that the order of adding data can vary the change of execution time, for example, in the context of inference with negation as failure.

Notions of processor capacity should be similar, although it is not necessary that they be exactly the same. For example, two evaluations using different notions of space-bounded, processor capacity are likely to still be comparable strictly in the context of data scaling.

Although more thorough discussion is warranted, an example, retroactive evaluation illustrating the differences of strong scaling and data scaling would likely be a better use of the remaining space, given the limitation on paper length. Some of the results from parallel, RDFS inference in [13] are reorganized in this section to address data scaling. The growing dataset is LUBM [3] generated using a seed of zero. A notion of space-bounded capacity is used, specifically RAM-bounded capacity. In this case, the processor capacity is 2,699,360 triples. This is not necessarily the maximum processor capacity, but since this evaluation is retroactive, it is sufficient for the purposes of demonstration.

This example is intended to illustrate how growth efficiency differs from efficiency $E_P = S_P/P$ in the strong scaling sense. Therefore, the two are plotted below over number of processors. (Recall that it was stated that the x-axis should be dataset size for growth efficiency, but such an x-axis is nonsensical for efficiency.)

Table 1a shows metrics for the overall computation, which includes I/O from/to a parallel file system. Efficiency and growth efficiency are plotted for number of processors in figure 1a. Growth efficiency for 256 processors is 0.66. That is, 256 times as much data was processed in about $1/0.66 \approx 1.5$ times as much time as with a single processor. Efficiency, which is 0.36 for 256 processors, does not make this evident at all.

Table 1b and figure 1b show the same metrics for only the inference portion of the computation. The inference portion of the computation is embarrassingly parallel, so there is no interprocess communication or contention. Clearly, the inference portion of the computation is very scalable – at least for LUBM data – in both the strong and data scaling senses. This indicates that the inference portion of the computation will likely scale to very large datasets without significantly impacting execution time, although the same cannot be said for the overall computation.

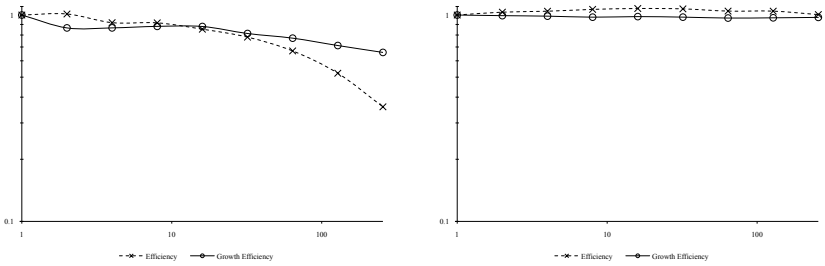
P	T_P	E_P	$\mathcal{T}_P$	$\mathcal{G}_P$
1	360	1.00	360	1.00
2	178	1.01	415	0.87
4	98.0	0.92	415	0.87
8	49.1	0.92	408	0.88
16	26.3	0.85	409	0.88
32	14.4	0.78	442	0.81
64	8.39	0.67	466	0.77
128	3.91	0.52	506	0.71
256	3.91	0.36	546	0.66

(a) Overall

P	T_P	E_P	$\mathcal{T}_P$	$\mathcal{G}_P$
1	283	1.00	283	1.00
2	137	1.03	285	0.99
4	67.7	1.04	286	0.99
8	33.2	1.06	289	0.98
16	16.4	1.08	288	0.98
32	8.24	1.07	289	0.98
64	4.23	1.04	292	0.97
128	2.11	1.04	291	0.97
256	1.10	1.00	290	0.97

(b) Inference Only

Table 1: Execution times, efficiency, and growth efficiency up to 256 processors



(a) Overall (b) Inference Only

Fig. 1: Efficiency and growth efficiency (log/log) up to 256 processors

5 Conclusion

Traditional scalability metrics from parallel computing fail to address the specific scalability challenge faced by parallel computations on data crawled from the Web, that is, the ability to handle large, growing datasets. A notion of data

scaling has been defined that is concerned with how execution time is affected as data grows, increasing the number of processors linearly with dataset size. A new metric, growth efficiency, has been introduced for evaluating data scalability of parallel computations. Focus has been on inference over RDF data crawled from the Semantic Web.

Acknowledgements. Much thanks to Jacopo Urbani and David Mizell for their constructive feedback in revising this paper.

References

1. Bizer, C.: Pay-as-you-go Data Integration on the public Web of Linked Data. Keynote Presentation at the 3rd Future Internet Symposium (September 2010)
2. Goodman, E.L., Mizell, D.: Scalable In-memory RDFS Closure on Billions of Triples. In: Proceedings of the 6th International Workshop on Scalable Semantic Web Knowledge Base Systems (2010)
3. Guo, Y., Pan, Z., Heflin, J.: LUBM: A benchmark for OWL knowledge base systems. Web Semantics: Science, Services and Agents on the World Wide Web 3(2-3) (2005)
4. Gustafson, J.L.: Reevaluating Amdahl's Law. Communications of the ACM 31(5), 532–533 (May 1988)
5. Hitzler, P., van Harmelen, F.: A Reasonable Semantic Web. Semantic Web Journal 1(1), 39–44 (2010)
6. Hogan, A., Pan, J.Z., Polleres, A., Decker, S.: SAOR: Template Rule Optimisations for Distributed Reasoning over 1 Billion Linked Data Triples. In: Proceedings of the 9th International Semantic Web Conference. (2010)
7. Kaoudi, Z., Miliaraki, I., Koubarakis, M.: RDFS Reasoning and Query Answering on Top of DHTs. In: Proceedings of the 8th International Semantic Web Conference. pp. 499–516 (2008)
8. Kotoulas, S., Oren, E., van Harmelen, F.: Mind the Data Skew: Distributed Inferencing by Speeddating in Elastic Regions. In: Proceedings of the 19th International World Wide Web Conference (2010)
9. Oren, E., Kotoulas, S., Anadiotis, G., Siebes, R., ten Teije, A., van Harmelen, F.: Marvin: Distributed reasoning over large-scale Semantic Web data. Web Semantics: Science, Services and Agents on the World Wide Web 7(4), 305–316 (2009)
10. Soma, R., Prasanna, V.K.: Parallel Inferencing for OWL Knowledge Bases. In: Proceedings of the 37th International Conference on Parallel Processing. pp. 75–82 (2008)
11. Urbani, J., Kotoulas, S., Maassen, J., van Harmelen, F., Bal, H.: OWL reasoning with WebPIE: calculating the closure of 100 billion triples. In: Proceedings of the 7th Extended Semantic Web Conference (2010)
12. Urbani, J., Kotoulas, S., Oren, E., van Harmelen, F.: Scalable Distributed Reasoning using MapReduce. In: Proceedings of the 8th International Semantic Web Conference (2009)
13. Weaver, J., Hendler, J.A.: Parallel Materialization of the Finite RDFS Closure for Hundreds of Millions of Triples. In: Proceedings of the 8th International Semantic Web Conference. pp. 682–697 (2009)
14. Williams, G.T., Weaver, J., Atre, M., Hendler, J.A.: Scalable Reduction of Large Datasets to Interesting Subsets. Web Semantics: Science, Services and Agents on the World Wide Web 8 (2010)