10th International Workshop on Scalable Semantic Web Knowledge Base Systems (SSWS 2014)

At the 13th International Semantic Web Conference (ISWC2014), Riva del Garda, Italy October, 2014

SSWS 2014 PC Co-chairs' Message

SSWS 2014 is the tenth edition of the successful Scalable Semantic Web Knowledge Base Systems workshop series. The workshop series is focussed on addressing scalability issues with respect to the development and deployment of knowledge base systems on the Semantic Web. Typically, such systems deal with information described in Semantic Web languages such as OWL and RDF(S), and provide services such as storing, reasoning, querying and debugging. There are two basic requirements for these systems. First, they have to satisfy the applications semantic requirements by providing sufficient reasoning support. Second, they must scale well in order to be of practical use. Given the sheer size and distributed nature of the Semantic Web, these requirements impose additional challenges beyond those addressed by earlier knowledge base systems. This workshop brought together researchers and practitioners to share their ideas regarding building and evaluating scalable knowledge base systems for the Semantic Web.

This year we received 9 submissions. Each paper was carefully evaluated by three workshop Program Committee members. Based on these reviews, we accepted 5 papers for presentation. We sincerely thank the authors for all the submissions and are grateful for the excellent work by the Program Committee members.

October 2014

Thorsten Liebig
Achille Fokoue

Program Committee

Mihaela Bornea
IBM Watson Research Center, USA

Oscar Corcho
Univ. Politecnica de Madrid, Spain

Mike Dean
Raytheon BBN Technologies, USA

Achille Fokoue
IBM Watson Research Center, USA

Jhonatan Garcia
University of Aberdeen, UK

Raúl García-Castro
Univ. Politecnica de Madrid, Spain

Volker Haarslev
Condordia University, Canada

Anastasios Kementsietsidis
Google Research, Mountain View, USA

Pavel Klinov
Ulm University, Germany

Adila A. Krisnadhi
Wright State University, Ohio, USA

Thorsten Liebig
derivo GmbH, Germany

Ralf Möller
Hamburg Univ. of Techn., Germany

Raghava Mutharaju
Wright State University, Ohio, USA

Jeff Z. Pan
University of Aberdeen, UK

Padmashree Ravindra
North Carolina State University, USA

Mariano Rodriguez-Muro
IBM Watson Research Center, USA

Pierpaolo Tommasi
IBM Research, Dublin, Ireland

Takahira Yamaguchi
Keio University, Japan

Additional Reviewers

Yuan Ren
University of Aberdeen, UK

Jelena Vlasenko
Free University of Bozen-Bolzano, Italy

Table of Contents

Invited Talk: RDFox A Modern Materialisation-Based RDF System 1
Boris Motik

The NPD Benchmark for OBDA Systems 3
Davide Lanti, Martin Rezk, Mindaugas Slusnys, Guohui Xiao and Diego Calvanese

Scheduling for SPARQL Endpoints 19
Fadi Maali, Islam A. Hassan and Stefan Decker

Querying Distributed RDF Graphs: The Effects of Partitioning 29
Anthony Potter, Boris Motik and Ian Horrocks

A Distributed Query Execution Method for RDF Storage Managers 45
Kiyoshi Nitta and Iztok Savnik

Distributed OWL EL Reasoning: The Story So Far 61
Raghava Mutharaju, Pascal Hitzler, Prabhaker Mateti

Invited Talk: RDFox A Modern Materialisation-Based RDF System

Boris Motik

University of Oxford

Abstract. RDFox is a new materialisation-based RDF system currently being developed at Oxford University. The system is currently RAM-based, and its algorithms have been designed to take full advantage of modern multi-core/processor systems. In my talk I will present an overview of some of the techniques we developed in the context of the RDFox project. In particular, I will discuss our algorithm that parallelises computation with very little overhead, I will present an overview of our lock-free indexes for RDF data, and I will discuss a novel incremental update algorithm. I will also briefly talk about some issues that we are currently working on, such as improving query planning and distributing data in a cluster of servers.

The NPD Benchmark for OBDA Systems

Davide Lanti, Martin Rezk, Mindaugas Slusnys, Guohui Xiao, and Diego Calvanese

Faculty of Computer Science, Free University of Bozen-Bolzano, Italy

Abstract. In Ontology-Based Data Access (OBDA), queries are posed over a high-level conceptual view, and then translated into queries over a potentially very large (usually relational) data source. The ontology is connected to the data sources through a declarative specification given in terms of mappings. Although prototype OBDA systems providing the ability to answer SPARQL queries over the ontology are available, a significant challenge remains: performance. To properly evaluate OBDA systems, benchmarks tailored towards the requirements in this setting are needed. OWL benchmarks, which have been developed to test the performance of generic SPARQL query engines, however, fail to evaluate OBDA specific features. In this work, we propose a novel benchmark for OBDA systems based on the Norwegian Petroleum Directorate (NPD). Our benchmark comes with novel techniques to generate, from available data, datasets of increasing size, taking into account the requirements dictated by the OBDA setting. We validate our benchmark on significant OBDA systems, showing that it is more adequate than previous benchmarks not tailored for OBDA.

1 Introduction

In Ontology-Based Data Access (OBDA), queries are posed over a high-level conceptual view, and then translated into queries over a potentially very large (usually relational) data source. The conceptual layer is given in the form of an ontology that defines a shared vocabulary. The ontology is connected to the data sources through a declarative specification given in terms of mappings that relate each (class and property) symbol in the ontology to a (SQL) view over the data. The W3C standard R2RML [10], was created with the goal of providing a standardized language for the specification of mappings in the OBDA setting. The ontology and expose a virtual instance (RDF graph) that can be queried using the de facto query language in the Semantic Web community: SPARQL. To properly evaluate the performance of OBDA systems, benchmarks tailored towards the requirements in this setting are needed. OWL benchmarks, which have been developed to test the performance of generic SPARQL query engines, however, fail at 1) exhibiting a complex real-world ontology, 2) providing challenging real world queries, 3) providing large amounts of real world data, and the possibility to test a system over data of increasing size, and 4) capturing important OBDA-specific measures related to the rewriting-based query answering approach in OBDA.

In this work, we propose a novel benchmark for OBDA systems based on the Norwegian Petroleum Directorate (NPD), which is a real world use-case adopted in the EU project Optique[1]. In the benchmark, which is available online[2], we adopt the NPD

[1] http://www.optique-project.eu/

[2] https://github.com/ontop/npd-benchmark

Fact Pages as dataset, the NPD Ontology, which has been mapped to the NPD Fact Pages stored in a relational database, and queries over such an ontology developed by domain experts. One of the main challenges we address here has been to develop a data generator for generating datasets of increasing size, starting from the available data. This problem has been studied before in the context of databases [2], where increasing the data size is achieved by encoding domain-specific information into the data generator [9, 4, 5]. One drawback of this approach is that each benchmark requires its ad-hoc generator, and also that it disregards OBDA specific aspects. In the context of triple stores, [25, 14] present an interesting approach based on machine learning. Unfortunately, the approach proposed in these papers is specifically tailored for triple stores, and thus it is not directly applicable to the OBDA settings.

In Section 2, we present the necessary requirements for an OBDA Benchmark. In Section 3, we discuss the requirements for an OBDA instance generator. In Section 4, we present the NPD benchmark and an associated relational database generator that gives rise to a virtual instance through the mapping; we call our generator *Virtual Instance Generator* (VIG). In Section 5, we perform a qualitative analysis of the virtual instances obtained using VIG. In Section 6, we carry out a validation of our benchmark, showing that it is more adequate than previous benchmarks not tailored for OBDA. We conclude in Section 7.

2 Requirements for Benchmarking OBDA

In this section we study the requirements that are necessary for a benchmark to evaluate OBDA systems. In order to define these requirements, we first recall that the three fundamental components of such systems are: *(i)* the *conceptual layer* constituted by the ontology; *(ii)* the *data layer* provided by the data sources; and *(iii)* the *mapping layer* containing the declarative specification connecting the ontological terms to the data sources. It is this mapping layer that decouples the virtual instance being queried, from the physical data stored in the data sources. Observe that triple stores cannot be considered as full-fledged OBDA systems, since they do not make a distinction between physical and virtual layer. However, given that both, OBDA systems and triple stores, are considered as (usually SPARQL) query answering systems, we consider it important that a benchmark for OBDA can also be used to evaluate triple stores. Also, since one of the components of an OBDA system is an ontology, the requirements we identify include those to evaluate general knowledge based systems [18, 14, 25]. However, due to the additional components, there are also notable differences.

Typically OBDA systems follow the workflow below for query answering:

1. *Starting phase*. The system loads the ontology, the mappings, and performs some auxiliary tasks needed to process/answer queries in a later stage. Depending on the system, this step step might be critical, since it might include some reasoning tasks, for example *inference materialization* or the embedding of the inferences into the mappings (*T-mappings* [20]).
2. *Query rewriting phase*. The input query is *rewritten* to a (maybe more complex) query that takes into account the inferences induced by the intensional level of the ontology (we forward the interested reader to [16]).
3. *Query translation (unfolding) phase*. The rewritten query is transformed into a query over the data sources. This is the phase where the mapping layer comes into play.

Table 1: Measures for OBDA

Performance Metrics		
name	triple store	related to phase
Loading Time	(**T**)	1
Rewriting Time	(**T***)	2
Unfolding Time	—	3
Query execution time	(**T**)	4
Overall response time	(**T**)	2, 3, 4
Quality Metrics		
Simplicity R Query	(**T***)	2
Simplicity U Query	—	3
Weight of R+U	(**T***)	2, 3, 4

4. *Query execution phase*. The data query is executed over the original data source, answers are produced according to the data source schema, and are translated into answers in terms of the ontology vocabulary and RDF data types, thus obtaining an answer for the original input query.

Note that a variation of the above workflow has actually been proposed in [18], but without identifying a distinct starting phase, and singling out a result translation phase from query execution. There are several approaches to deal with Phase 2 [16, 24]. The most challenging task in this phase is to deal with existentials in the right-hand side of ontology axioms. These axioms infer unnamed individuals in the virtual instance that cannot be retrieved as part of the answer, but can affect the evaluation of the query. An approach that has proved to produce good results in practice is the *tree-witness rewriting* technique, for which we refer to [16]. For us, it is only important to observe that *tree-witnesses* lead to an extension of the original query to account for matching in the existentially implied part of the virtual instance. Below, we take the number of tree-witnesses identified in Phase 2 as one of the parameters to measure the complexity of the combination ontology/query. Since existentials do not occur very often in practice [16], and can produce an exponential blow-up in the query size, some systems allow to turn off the part of Phase 2 that deals with *reasoning with respect to existentials*.

Ideally, an OBDA benchmark should provide *meaningful* measures for each of these phases. Unfortunately, such a fine-grained analysis is not always possible, for instance because the system comes into the form of a black-box with proprietary code with no APIs providing the necessary information, e.g., the access to the rewritten query; or because a system combines one or more phases, e.g., query rewriting and query translation. Based on the above phases, we identify the measures important for *evaluating* OBDA systems in Table 1. The meaning of the *Performance Measures* is clear from the name, but we will give a brief explanation of the meaning of the *Quality Metrics*:

– *Simplicity R Query*. Simplicity of the rewritten query in terms of language dependent measures, like the *number of rules* in case the rewritten query is a datalog program. In addition, one can include system-dependent features, e.g., # of tree-witnesses in *Ontop*.

– *Simplicity U Query*. This measures the simplicity of the query over the data source, including relevant SQL-specific metrics like the number of joins/left-join, the number of inner queries, etc.
– *Weight of R+U*. It is the cost of the construction of the SQL query divided by the overall cost.

We label with **(T)** those measures that are also valid for triple stores, and with (**T***) those that are valid only if the triple store is based on query rewriting (e.g., Stardog). Notice that the two *Simplicity* measures, even when retrievable, are not always suitable for *comparing* different OBDA systems. For example, it might not be possible to compare the simplicity of queries in the various phases, when such queries are expressed in different languages.

With these measures in mind, the different components of the benchmark should be designed so as to reveal strengths and weaknesses of a system in each phase. The conclusions drawn from the benchmark are more significant if the benchmark resembles a typical real-world scenario in terms of the complexity of the ontology and queries and size of the data set. Therefore, we consider the requirements in Table 2.

Table 2: Benchmark Requirements

O1	**Q1**	**M1**
The **ontology** should include rich hierarchies of classes and properties.	The **query set** should be based on actual user queries.	The **mappings** should be defined for elements of most hierarchies.
O2	**Q2**	**M2**
The **ontology** should contain a rich set of axioms that infer new objects and could lead to inconsistency, in order to test the reasoner capabilities.	The **query set** should be complex enough to challenge the query rewriter.	The **mappings** should contain redundancies, and suboptimal SQL queries to test optimizations.
D1	**D2**	**S1**
The **virtual instance** should be based on real world data.	The size of the **virtual instance** should be tunable.	The **languages** of the ontology, mapping, and query should be *standard*, i.e., based on R2RML, SPARQL, and OWL respectively.

The current benchmarks available for OBDA do not meet several of the requirements above. Next we list some of the best known benchmarks and their shortcomings when it comes to evaluate OBDA systems. We show general statistics in Table 3.

Adolena: Designed in order to extend the South African National Accessibility Portal [15] with OBDA capabilities. It provides a rich class hierarchy, but a quite poor structure for properties. This means that queries over this ontology will usually be devoid of tree-witnesses. No data-generator is included, nor mappings.
Requirements Missing: O1, **Q2**, **D2**, **S1**

LUBM: The Lehigh University Benchmark (LUBM) [13] consists of a university domain ontology, data, and queries. For data generation, the UBA (Univ-Bench

Table 3: Popular Benchmark Ontologies: Statistics

name	Ontology Stats. (Total) #classes	#obj/data_prop	#i-axioms	Queries Stats. (Max) #joins	#opt	#tw
adolena	141	16	189	5	0	0
lubm	43	32	91	7	0	0
dbpedia	530	2148	3836	7	8	0
bsbm	8	40	0	14	4	0
fishmark	11	94	174	24	12	0

Artificial) data generator is available. However, the ontology is rather small, and the benchmark is not tailored towards OBDA, since no mappings to a (relational) data source are provided.
Requirements Missing: O1, Q2, M1, M2, D1

DBpedia: The DBpedia benchmark consists of a relatively large—yet, simple[3]—ontology, a set of user queries chosen among the most popular queries posed against the DBpedia[4] SPARQL endpoint, and a synthetic RDF data generator able to generate data having similar properties to the real-world data. This benchmark is specifically tailored to triple stores, and as such it does not provide any OBDA specific components like R2RML mappings, or a data set in the form of a relational database.
Requirements Missing: O1, O2, Q2, M1, M2

BSBM: The Berlin SPARQL Benchmark [3] is built around an e-commerce use case. It has a data generator that allows one to configure the data size (in triples), but there is no ontology to measure reasoning tasks, and the queries are rather simple. Moreover, the data is fully artificial.
Requirements Missing: O1, O2, Q2, M1, M2, D1,

FishMark: FishMark [1] collects comprehensive information about finned fish species. This benchmark is based in the FishBase real world dataset, and the queries are extracted from popular user SQL queries over FishBase; they are more complex than those from BSBM. However, the benchmark comes neither with mappings nor with a data generator. The data size is rather small (≈20M triples).
Requirements Missing: O1, D2, S1

A specific challenge comes from requirements **D1** and **D2**, i.e., given an initial real-world dataset, together with a rich ontology and mappings, expand the dataset in such a way that it populates the virtual instance in a sensible way (i.e., coherently with the ontology constraints and relevant statistical properties of the initial dataset). We address this problem in the next section.

3 Requirements for Generating Virtual Instances

In this section, we present the requirements for an OBDA data generator, under the assumption that we have an initial database that can be used as a seed to understand the

[3] In particular, it is not suitable for reasoning w.r.t. existentials.
[4] `http://dbpedia.org/sparql`

distribution of the data that needs to be increased. To ease the presentation, we illustrate the main issues that arise in this context with an example.

Example 1. Suppose we have the following database tables, where the primary keys are in **bold font** and the foreign keys are *emphasized*. We assume that every employee sells the majority of the products, hence the table TSellsProduct contains roughly the cross product of the tables TEmployee and TProduct. Next we present only a fragment of the data.

TEmployee

id	name	branch
1	John	B1
2	Lisa	B1

TAssignment

branch	task
B1	task1
B1	task2
B2	task1
B2	task2

TSellsProduct

id	*product*
1	p1
2	p2
1	p2
2	p3

TProduct

product	size
p1	big
p2	big
p3	small
p4	big

Consider the following mappings populating the ontology concepts Employee, Branch, and ProductSize, and the object properties SellsProduct and AssignedTo.

```
M1 :{id} rdf:type :Employee       ← SELECT id from TEmployee
M2 :{branch} rdf:type :Branch     ← SELECT branch FROM TAssignments
M3 :{branch} rdf:type :Branch     ← SELECT branch FROM TEmployee
M4 :{id} :SellsProduct :{product} ← SELECT id, product FROM TSellsProduct
M5 :{size} rdf:type :ProductSize  ← SELECT size FROM TProduct
M6 :{id} :AssignedTo :{task}      ← SELECT id, task
                                    FROM TEmployee
                                    NATURAL JOIN
                                    TAssignments
```

The virtual instance corresponding to the above database and mappings includes the following RDF triples:

```
:1  rdf:type  :Employee.        :1  :SellsProduct  :p1.
:2  rdf:type  :Employee.        :1  :SellsProduct  :p2.
                                :2  :AssignedTo    :t1.
```

Suppose now we want to increase the *virtual* RDF graph by a *growth-factor* of 2. Observe that this is not as simple as doubling the number of triples in every concept and property, or the number of tuples in every database relation. Let us first analyze the behavior of some of the ontology elements w.r.t. this aspect, and then how the mappings to the database come into play.

- ProductSize: This concept will contain two individuals, namely small and big, independently of the growth-factor. Therefore, the virtual instances of the concept should not be increased when the RDF graph is extended.
- Employee and Branch: Since these classes do not depend on other properties, and since they are not intrinsically constant, we expect their size to grow linearly with the growth-factor.
- AssignedTo: Since this property represents an n-to-n relationship, we expect its size to grow roughly quadratically with the growth-factor.
- SellsProduct: The size of this property grows roughly with the product of the numbers of Employees and Products. Hence, when we double these numbers, the size of SellsProduct will roughly quadruplicate.

In fact, the above considerations show that we do not have *one* uniform growth-factor for the ontology elements. Our choice is to characterize the growth in terms of the increase in size of those concepts in the ontology that are not intrinsically constant (e.g., `ProductSize`), and that do not "depend" on any other concept, considering the semantics of the domain of interest (e.g., `Employee`). We take this as measure for the growth-factor.

The problem of understanding how to generate from a given RDF graph new additional triples coherently with the domain semantics is addressed in [25, 19]. The algorithm in [25] starts from an initial RDF graph and produces a new RDF graph, considering key features of the original graph (e.g., the distribution of connections among individuals). However, this approach, and all approaches producing RDF graphs in general, cannot be directly applied to the context of OBDA, where the RDF graph is virtual and generated from a relational database. Trying to apply these approaches indirectly, by first producing a "realistic" virtual RDF graph and then trying to reflect the virtual data into the physical (relational) data-source, is far from trivial due to the correlations in the underlying data. This problem, indeed, is closely related to the *view update problem* [8], where each class (resp., role or data property) can be seen as a *view* on the underlying physical data. The view update problem is known to be challenging and actually decidable only for a very restricted class of queries used in the mappings [12]. Note, however, that our setting does not necessarily require to fully solve the view update problem, since we are interested in obtaining a physical instance that gives rise to a virtual instance with certain statistics, but not necessarily to a specific given virtual instance. The problem we are facing nevertheless remains challenging, and requires further research. We illustrate the difficulties that one encounters again on our example.

- The property `SellsProduct` grows linearly w.r.t. the size of the table `TSellsProduct`, hence also this table has to grow quadratically with the growth-factor. Since `TSellsProduct` has foreign keys from the tables `TEmployee` and `TProduct`, to preserve the inter-table correlation (according to which roughly every employee is connected to every product), the two tables `TEmployee` and `TProduct` have both to grow linearly. It is worth noting that, to produce one `SellsProduct` triple in the virtual instance, we have to insert three tuples in the database.
- Since also the `Branches` concept should grow linearly with the growth-factor, while preserving the intra- and inter-table correlations, also the `TAssignment` table should grow linearly, and there should always be less branches than employees in `TEmployee`.
- Since `ProductSize` does not grow, the attribute `Size` must contain only two values, despite the linear growth of `TProduct`.

The previous example illustrated several challenges that need to be addressed by the generator regarding the *analysis* of the virtual and physical data, and the *insertion* of values in the database. Our goal is to generate a synthetic virtual graph where the cost of the queries is as similar as possible to the cost that the same query would have in a real-world virtual graph of comparable size. Observe that the same virtual graph can correspond to different database instances, that could behave very differently w.r.t. the

cost of SQL query evaluation. Therefore, in order to keep the cost of the SPARQL query "realistic", we need to keep the cost of the translated SQL "realistic" as well.

We are interested in data generators that perform an analysis phase on real-world data, and that use the statistical information learned in the analysis phase for their task. We present first in Table 5 the measures that are relevant in the analysis phase. We then derive the requirements for the data generator by organizing them in two categories: one for the analysis phase, and one for the generation phase.

Measures for the Analysis Phase. Table 5 is divided in three parts. The top part refers to measures relevant at virtual instance level, i.e., those capturing the shape of the virtual instance. *Virtual correlation* measures the correlation between individuals connected through a property, i.e., the number of individuals/values to which every individual is connected via an object/data property. *Virtual growth* is the expected growth for each ontology term w.r.t. the growth-factor. Observe that these two measures are strongly related to each other. The middle part refers to measures at the physical level that strongly affect the shape of the virtual instance through the form of the mappings. They are based on the sets of attributes of a table used to define individuals and values in the ontology through the mapping. We call such a set of attributes an *IGA* (individual-generating attributes set). Establishing the relevant statistics requires to identify pairs of IGAs through mapping analysis. Specifically, *intra-table IGA correlation* is defined for two IGAs of the same table, both mapped to individuals/values at the virtual level. It is measured for tuples over the IGAs as the virtual correlation of the individuals that are generated via the mapping from the tuples. *Inter-table IGA correlation* is measured for IGAs belonging to two different tables, by taking the intra-table IGA correlation over the join of the two tables. The bottom part refers to measures at the physical level that do not affect correlation at the virtual instance level, but that influence growth at the virtual level and the overall performance of the system. Specifically, *IGA duplication* measures the number of identical copies of tuples over an IGA, while (intra-table and inter-table) *IGA-pair duplication* is measured as the number of identical copies of a tuple over two correlated IGAs. Notice that, for benchmarking purposes, both IGA correlation and IGA duplication are important.

Now we are ready to list the requirements for a data generator for OBDA systems.

Requirements for the Analysis Phase. The generator should be able to analyze the physical instance and the mappings, in order to acquire statistics to assess the measures identified in Table 5.

Requirements for the Generation Phase. We list now important requirements for the generation of physical data that gives rise through the mappings to the desired virtual data instance.

Tunable. The user must be able to specify a growth factor according to which the virtual instance should be populated.

Virtually Sound. The virtual instance corresponding to the generated physical data must meet the statistics discovered during the analysis phase and that are relevant at the virtual instance level.

Physically Sound. The generated physical instance must meet the statistics discovered during the analysis phase and that are relevant at the physical instance level.

Table 5: Relevant measures at the virtual and physical instance level

Measures affecting the virtual instance level	
Virtual Correlation (VC)	**Virtual Growth (VG)**
Correlations between the various elements in the virtual instance.	Function describing how fast concepts (resp., role/data properties) grow w.r.t. the growth-factor.
Measures affecting virtual correlation and virtual growth	
Intra-table IGA Correlation (Intra-C)	**Inter-table IGA Correlation (Inter-C)**
Correlation (obtained through repetition analysis) between IGAs belonging to the same table and generating objects connected through a mapped property.	Correlation (obtained through analysis of the repetitions of *tuples used to join IGAs* and of the joined IGAs) between IGAs belonging to different tables.
Measures affecting RDBMS performance and virtual growth	
IGA Duplication (D)	
Repeated IGAs	
Intra-table IGA-pair Duplication (Intra-D)	**Inter-table IGA-pair Duplication (Inter-D)**
Repeated pairs of intra-table correlated IGAs.	Repeated pairs of inter-table correlated IGAs.

Database Compliant. The generator must generate data that does not violate the constraints of the RDBMS engine—e.g., primary keys, foreign keys, constraints on datatypes, etc.

Fast. The generator must be able to produce a vast amount of data in a reasonable amount of time (e.g., 1 day for generating an amount of data sufficient to push the limits of the considered RDBMS system). This requirement is important because OBDA systems are expected to operate in the context of "big-data" [6].

4 NPD Benchmark

The Norwegian Petroleum Directorate[5] (NPD) is a governmental organisation whose main objective is to contribute to maximize the value that society can obtain from the oil and gas activities. The initial dataset that we use are the *NPD Fact Pages*[6], containing information regarding the petroleum activities on the Norwegian continental shelf. The ontology, the query set, and the mappings to the dataset have all been developed at the University of Oslo [22], and are freely available online[7]. Next we provide more details on each of these items.

The Ontology. The ontology contains OWL axioms specifying comprehensive information about the underlying concepts in the dataset; specifically rich hierarchies of classes and properties, axioms that infer new objects, and disjointness assertions. We took the OWL QL fragment of this ontology, and we obtained 343 classes, 142 object properties, 238 data properties, 1451 axioms, and maximum hierarchy depth of 10. Since we are interested in benchmarking OBDA systems that are able to rewrite queries over the ontology into SQL-queries that can be evaluated by a relational DBMS, we concentrate

[5] http://www.npd.no/en/
[6] http://factpages.npd.no/factpages/
[7] http://sws.ifi.uio.no/project/npd-v2/

Table 6: Statistics for the queries considered in the benchmark

query	#operators	#join	#depth	#tw	max(#subclasses)	# opt
Q1	12	4	8	0	0	0
Q2	14	5	9	0	0	0
Q3	10	3	7	0	0	0
Q4	14	5	9	0	0	0
Q5	14	5	8	0	0	0
Q6	18	6	12	2	23	0
Q7	17	7	10	0	0	0
Q8	10	3	7	0	0	0
Q9	10	3	7	0	38	0
Q10	9	2	7	0	0	0
Q11	20	7	12	2	23	0
Q12	26	8	12	4	23	0
Q13	8	2	7	0	0	2
Q14	12	2	7	0	0	2

here on the OWL 2 QL profile[8] of OWL, which guarantees rewritability of unions of conjunctive queries (see, e.g., [7]). This ontology is suitable for benchmarking reasoning tasks, given that *(i)* it is a representative [17] and complex real-world ontology in terms of number of classes and maximum depth of the class hierarchy (hence, it allows for reasoning w.r.t. class hierarchies); *(ii)* it is complex w.r.t. properties, therefore it allows for reasoning with respect to existentials.
From the previous facts, it follows that the ontology satisfies requirements **O1**, **O2**, **S1**.

The Query Set. The original NPD SPARQL query set contains 25 queries obtained by interviewing users of the NPD dataset. Starting from the original NPD query set, we devised 14 queries having different degrees of complexity (see Table 6). In particular, observe that most complex queries involve both classes with a rich hierarchy and tree witnesses, which means that they are particularly suitable for testing the reasoner capabilities. We also fixed some minor issues, e.g., the absence in the ontology of certain concepts present in the queries, removing aggregates (to be tackled in future work), and flattening of nested sub-queries.
From the previous facts, it follows that the queries satisfy requirements **Q1**, **Q2**, **S1**.

The Mappings. The R2RML mapping consists of 1190 assertions mapping a total of 464 among classes, objects properties, and data properties. The SQL queries in the mappings count an average of 2.6 unions of select-project-join queries (SPJ), with 1.7 joins per SPJ. We observe that the mappings have not been optimized to take full advantage of an OBDA framework, e.g., by trying to minimize the number of mappings that refer to the same ontology class or property, so as to reduce the size of the SQL query generated by unfolding the mapping. This gives the opportunity to the OBDA system to apply different optimization on the mappings at loading time.
From the previous facts, it follows that the mappings satisfies requirements **M1**, **M2**, **S1**.

4.1 VIG: The Data Generator.

Next we present the **V**irtual **I**nstances **G**enerator (VIG) that we implemented in the NPD Benchmark. VIG produces a virtual instance by inserting data into the original database. The generator is general in the sense that, although it currently works with the NPD database, it can produce data also starting from instances different than NPD. The algorithm can be divided into two main phases, namely (i) an *analysis phase*, where

[8] `http://www.w3.org/TR/owl2-profiles/#OWL_2_QL`

statistics for relevant measures on the real-world data are identified; (ii) a *generation phase*, where data is produced according to the statistics identified in the analysis phase.

VIG starts from a non-empty database D. Given a growth factor g, VIG generates a new database D' such that $|T'| = |T| \cdot (1+g)$, for each table T of D (where $|T|$ denotes the number of tuples of T). This first approach assumes that each table in the database should grow linearly with respect to the growth factor, which is not true in general, but it holds for NPD. In addition, VIG approximates the measures described Table 5 as shown below.

Measures (D), (Intra-D). We compute (an approximation for) these measures by *Duplicate Values Discovery.* For each column $T.C$ of a table $T \in D$, VIG discovers the *duplicate ratio* for values contained in that column. The *duplicate ratio* is the ratio $(\|T.C\| - |T.C|)/\|T.C\|$, where $\|T.C\|$ denotes the number of values in the column $T.C$, and $|T.C|$ denotes the number distinct of values in $T.C$. A duplicate ratio "close to 1" indicates that the content of the column is essentially *independent* from the size of the database, and it should not be increased by the data generator.

Measures (Intra-C), (Inter-C), (Inter-D). Instead of computing (an approximation for) these measures, VIG identifies the domain of each attribute. That is, for each non-fixed domain column $T.C$ in a table T, VIG analyzes the content of $T.C$ in order to decide the range of values from which *fresh* non-duplicate values can be chosen. More specifically, if the domain of $T.C$ is a string or unordered (e.g., polygons), then simply a random value is generated. Instead, if the domain is a total order, then fresh values can be chosen from the non-duplicate values in the interval $[\min(T.C), \max(T.C)]$ or in the range of values adjacent to it. Observe that this helps in maintaining the domain of a column similar to the original one, and this in turn helps in maintaining intra- and inter-table correlations. VIG also preserves standard database constraints, like primary keys, foreign keys, and datatypes, that during the generation phase will help in preserving the IGA correlations. For instance, VIG analyses the loops in foreign key dependencies in the database. Let $T_1 \rightarrow T_2$ denote the presence of a foreign key from table T_1 to table T_2. In case of a cycle $T_1 \rightarrow T_2 \rightarrow \cdots \rightarrow T_k \rightarrow T_1$, inserting a tuple in T_1 could potentially trigger an infinite number of insertions. VIG performs an analysis on the values contained in the columns involved by the dependencies and discovers the maximum number of insertions that can be performed in the generation phase.

Next we describe the generation phase, and how it meets some of the requirements given in *Section* 5.

Duplicate Values Generation. VIG inserts duplicates in each column according to the duplicate ratio discovered in the analysis phase. Each duplicate is chosen with a uniform probability distribution. This ensures, for those concepts that are not dependent from other concepts and whose individual are "constructed" from a single database column, a growth that is equal to the growth factor. In addition, it prevents intrinsically constant concepts from being increased (by never picking a fresh value in those columns where the duplicates ratio is close to 1). Finally, it helps keeping the sizes for join result sets "realistic" [23]. This is true in particular for the NPD database, where almost every join is realized by a single equality on two columns.
Requirement: Physically/Virtually Sound.

Fresh Values Generation. For each column, VIG picks fresh non-duplicate values from the interval discovered during the analysis phase. If the number of values to insert exceeds the number of different fresh values that can be chosen from the interval $\mathcal{I}$, then values outside the interval are allowed. The choices for the generation of new value guarantee that columns always contain values "close" to the ones already present in the column. This ensures that the number of individual for concepts based on comparisons grows accordingly to the growth factor.
Requirement: Physically/Virtually Sound.

Metadata Constraints VIG generates values that do not violate the constraints of the underlying database, like primary keys, foreign keys, or type constraints. The NPD database makes use of geometric datatypes available in MYSQL. Some of them come with constraints, e.g., a polygon is a closed non-intersecting line composed of a finite number of straight lines. For each geometric column in the database, VIG first identifies the minimal rectangular region of space enclosing all the values in the column, and then it generates values in that region. This ensures that artificially generated geometric values will fall in the result sets of selection queries.
Requirement: Database Compliant/Virtually Sound.

Length of Chase Cycles. In case a cycle of foreign key dependencies was identified during the analysis phase, then VIG stops the chain of insertions according to the boundaries identified in the analysis phase, while ensuring that no foreign key constraint is violated. This is done by inserting either a duplicate or a null in those columns that have a foreign key dependency.
Requirement: Database Compliant.

Furthermore, VIG allows the user to tune the growth factor, and the generation process is considerably fast, for instance, it takes ≈10hrs to generate 130 Gb of data.

5 Validation of the Data Generator

In this section we perform a qualitative analysis of the virtual instances obtained using VIG. We focus our analysis on those concepts and properties that either are supposed to grow linearly w.r.t. the growth factor or are supposed not to grow. These are 138 concepts, 28 object properties, and 226 data properties.

We report in Table 7 the growth of the ontology elements w.r.t. the growth of databases produced by VIG and by a purely random generator. The first column indicates the type of ontology elements being analyzed, and the growth factor g (e.g., "class_npd2" refers to the population of classes for the database incremented with a growth factor $g = 2$). The columns "avg dev" show the average deviation of the actual growth from the expected growth, in terms of percentage of the expected growth. The remaining columns report the number and percentage of concepts (resp., object/data properties) for which the deviation was greater than 50%.

Concerning concepts, VIG behaves close to optimally. For properties, the difference between the expected virtual growth and the actual virtual growth is more evident. However, it is orders of magnitude better than the random generator. We shall see how this difference strongly affects the results of the benchmark (*Section* 6).

Table 7: Comparison between VIG and a random data generator

type_db	avg dev heuristic	avg dev random	err ≥50% (absolute) heuristic	err ≥50% (absolute) random	err ≥50% (relative) heuristic	err ≥50% (relative) random
class_npd2	3.24%	370.08%	2	67	1.45%	48.55%
class_npd10	6.19%	505.02%	3	67	2.17%	48.55%
obj_npd2	87.48%	648.22%	8	12	28.57%	42.86%
obj_npd10	90.19%	883.92%	8	12	28.57%	42.86%
data_npd2	39.38%	96.30%	20	46	8.85%	20.35%
data_npd10	53.49%	131.17%	28	50	12.39%	22.12%

Table 8: Tractable queries (time in ms)

db	avg(ex_time)	avg(out_time)	avg(res_size)	qmpH	#(triples)
NPD	62	113	16766	1507.85	≈2M
NPD2	116	232	35991	896.55	≈6M
NPD5	246	410	70411	554.35	≈12M
NPD10	408	736	124253	305.46	≈25M
NPD50	2138	3208	539461	66.82	≈116M
NPD100	5292	6727	1160972	29.12	≈220M
NPD500	37382	48512	7511516	4.22	≈1.3B
NPD1500	132155	148495	23655243	1.27	≈4B

Virtual Correlation. From our experiments we witnessed that the virtual correlation is preserved for the 28 object properties that are generated from a single table. That is, the correlation remains constant and it grows only in the case of cartesian products on columns with high duplicate ratio and that together form a primary key. More results can be found in the benchmark page.

6 Benchmark Results

We ran the benchmark on the *Ontop* system[9] [21], which, to the best of our knowledge, is the only fully implemented OBDA system that is freely available. In addition, we compared *Ontop* with *Stardog 2.1.3*. Stardog[10] is a commercial RDF database developed by Clark&Parsia that supports SPARQL 1.1 queries and OWL 2 for reasoning. Since Stardog is a triple store, we needed to materialize the virtual RDF graph exposed by the mappings and the database using *Ontop*.

MYSQL was used as underlying relational database system. The hardware consisted of an HP Proliant server with 24 Intel Xeon X5690 CPUs (144 cores @3.47GHz), 160GB of RAM and a 1TB 15K RPM HD. The OS is Ubuntu 12.04 LTS. Due to space

[9] `http://ontop.inf.unibz.it/`
[10] `http://stardog.com/`

Table 9: Hard Queries Rewriting And Unfolding

query	Ext. Reasoning OFF #rw	Ext. Reasoning OFF #un	Ext. Reasoning OFF rw time sec.	Ext. Reasoning OFF un time sec.	Ext. Reasoning ON #rw	Ext. Reasoning ON #un	Ext. Reasoning ON rw time sec.	Ext. Reasoning ON un time sec.
q6	—	48	—	0.1	73	1740	1.8	1.3
q9	—	570	—	0.1	1	150	0	0.03
q10	—	24	—	0.9	1	24	0	0.01
q11	—	1	—	0.1	73	870	0.03	0.7
q12	—	1	—	0.2	10658	5220	525	139

constraints, we present the results for only one running client. We obtained results with the *existential reasoning* on and off.

In order to test the scalability of the systems w.r.t. the growth of the database, we used the data generator described in Section 4.1 and produced several databases, the largest being approximately 1500 times bigger than the original one ("NPD1500" in Table 8, ≈ 117 GB of size on disk (≈ 6 Bn triples)).

Table 8 shows the 9 easiest queries from the initial query set, for which the unfolding produces a single select-project-join SQL query. These results were obtained in *Ontop*. The query mix of 9 queries was executed 10 times, each time with different filter conditions so that the effect of caching is minimized, and statistics were collected in each execution. We measure the sum of the *query execution time* (avg(ex_time)), the time spent by the system to display the results to the user (avg(out_time)), the number of results (avg(res_size)), and the query mixes per hour (qmpH), that is, the number of times that the 9 queries can be answered in one hour.

Table 9 contains results showing the number of unions of SPJ queries generated after rewriting (#rw) and after unfolding (#un) for the 5 hardest queries. In addition, it shows the time spent by *Ontop* on rewriting and unfolding. Here we can observe how existential reasoning can produce a noticeable performance overhead, by producing queries consisting of unions of more than 5000 sub-queries (c.f., q12). This blow-up is due to the combination of rich hierarchies, existentials, and mappings. These queries are meant to be used in future research on query optimization in OBDA.

Table 10 contains results for the 5 hardest queries in *Ontop*. Each query was run once, since qmpH is not so informative in this case. Observe that the response time tends to grow faster than the growth of the underlying database. This follows from the complexity of the queries produced by the unfolding step, which usually contain several joins (remember that the worst case cardinality of a result set produced by a join is quadratic in the size of the original tables). Column NPD10 RAND witnesses how using a purely random data generator gives rise to datasets for which the queries are much simpler to evaluate. This is mainly due to the fact that a random generation of values tends to decrease the ratio of duplicates inside columns, resulting in smaller join results over the tables [23]. Hence, purely randomly generated datasets are not appropriate for benchmarking.

Table 11 shows the 6 queries where the performance difference is bigger. As expected, the 3 queries with worst performance in OBDA are those that were affected by the blow-up shown in Table 9. On the other hand, the 3 queries that perform the best are those where the different optimization led to a simple SPJ SQL query. Note how in these 3 queries, in *Ontop*, the overhead caused by the increment of dataset size is minimum.

7 Conclusions and Future Work

The benchmark proposed in this work is the first one that thoroughly analyzes a complete OBDA system in all significant components. So far, little or no work has been done in this direction, as pointed out in [18], since the research community has mostly focused on *rewriting* engines. Thanks to our work, we have gained a better understanding of the current state of the art for OBDA systems: first, we confirm [11] that the unfolding phase is the real bottleneck of modern OBDA systems; second, more research work is needed in order to understand how to improve the design of mappings,

Table 10: Hard queries

query	NPD rp_t/weight R+U (sec./ratio)	NPD2 rp_t/weight R+U (sec./ratio)	NPD5 rp_t/weight R+U (sec./ratio)	NPD10 rp_t/weight R+U (sec./ratio)	NPD10 RAND rp_t/weight R+U (sec./ratio)
No Existentials Reasoning					
q6	1.5/0.07	8.2/0.02	23/<0.01	51/<0.01	54/<0.01
q9	0.6/0.17	2.3/0.03	4/0.03	50/<0.01	51/<0.01
q10	0.07/0.14	0.1/0.1	0.16/0.06	0.2/0.05	0.3/0.03
q11	0.9/0.1	36/<0.01	198/<0.01	1670/<0.01	70/<0.01
q12	0.8/0.16	41/<0.01	275/<0.01	1998/<0.01	598/<0.01
Existentials Reasoning					
q6	8.5/0.35	18/0.19	36/0.09	85/0.04	88/0.03
q9	0.2/0.2	0.2/0.2	0.2/0.2	0.2/0.2	0.2/0.2
q10	0.1/0.2	0.1/0.2	0.3/0.07	0.7/0.03	1.8/0.01
q11	3/0.2	25/0.03	980/<0.01	980/<0.01	41/0.02
q12	686/0.97	733/0.91	868/0.74	2650/0.24	880/0.74

Table 11: Query executions in Stardog and Ontop (Time in sec.)

Query	NPD1		NPD2		NPD5		NPD10	
	Stardog	Ontop	Stardog	Ontop	Stardog	Ontop	Stardog	Ontop
q1	1.56	0.597	1.616	0.463	1.852	0.683	6.184	0.571
q7	1.585	0.021	2.223	0.041	3.714	0.108	5.199	0.204
q8	0.865	0.08	1.561	0.192	2.504	0.526	6.492	0.669
q6	0.486	1.593	1.592	21.217	2.678	23.272	4.285	88.979
q11	0.394	0.974	1.412	25.693	2.138	197.786	2.584	978.465
q12	0.425	0.934	1.649	275.548	2.65	1396.185	3.464	3292.464
Mater.	54.75s		2m58.014s		9m58.509s		41m23s	
Load	60.935s		4m42.849s		12m8.565s		57m18.297s	

avoiding the use of mappings that give rise to huge queries after unfolding. We conclude by observing that for a better analysis it is crucial to refine the generator in such a way that domain-specific information is taken into account, and a better approximation of real-world data is produced.

References

1. Bail, S., Alkiviadous, S., Parsia, B., Workman, D., van Harmelen, M., Goncalves, R.S., Garilao, C.: FishMark: A linked data application benchmark. In: Proc. of the Joint Workshop on Scalable and High-Performance Semantic Web Systems (SSWS+HPCSW 2012), vol. 943, pp. 1–15. CEUR, `ceur-ws.org` (2012)
2. Bitton, D., Dewitt, D.J., Turbyfill, C.: Benchmarking database systems: A systematic approach. In: Proc. of VLDB. pp. 8–19 (1983)
3. Bizer, C., Schultz, A.: The Berlin SPARQL benchmark. Int. J. on Semantic Web and Information Systems 5(2), 1–24 (2009)
4. Bruno, N., Chaudhuri, S.: Flexible database generators. In: Proc. of VLDB. pp. 1097–1107 (2005)
5. Bsche, K., Sellam, T., Pirk, H., Beier, R., Mieth, P., Manegold, S.: Scalable generation of synthetic GPS traces with real-life data characteristics. In: Selected Topics in Performance Evaluation and Benchmarking, LNCS, vol. 7755, pp. 140–155. Springer (2013)
6. Calvanese, D., Giese, M., Haase, P., Horrocks, I., Hubauer, T., Ioannidis, Y., Jiménez-Ruiz, E., Kharlamov, E., Kllapi, H., Klüwer, J., Koubarakis, M., Lamparter, S., Möller, R., Neuenstadt, C., Nordtveit, T., Özcep, Ö., Rodriguez-Muro, M., Roshchin, M., Ruzzi, M., Savo, F., Schmidt, M., Soylu, A., Waaler, A., Zheleznyakov, D.: The Optique project: Towards OBDA systems for industry (Short paper). In: Proc. of OWLED. CEUR, `ceur-ws.org`, vol. 1080 (2013)

7. Calvanese, D., De Giacomo, G., Lembo, D., Lenzerini, M., Poggi, A., Rodríguez-Muro, M., Rosati, R.: Ontologies and databases: The *DL-Lite* approach. In: Tessaris, S., Franconi, E. (eds.) RW Tutorial Lectures, LNCS, vol. 5689, pp. 255–356. Springer (2009)
8. Cosmadakis, S.S., Papadimitriou, C.H.: Updates of relational views. JACM 31(4), 742–760 (1984)
9. Crolotte, A., Ghazal, A.: Introducing skew into the TPC-H Benchmark. In: Topics in Performance Evaluation, Measurement and Characterization, LNCS, vol. 7144, pp. 137–145. Springer (2012)
10. Das, S., Sundara, S., Cyganiak, R.: R2RML: RDB to RDF mapping language. W3C Recommendation, W3C (Sep 2012), available at `http://www.w3.org/TR/r2rml/`
11. Di Pinto, F., Lembo, D., Lenzerini, M., Mancini, R., Poggi, A., Rosati, R., Ruzzi, M., Savo, D.F.: Optimizing query rewriting in ontology-based data access. In: Proc. of EDBT. pp. 561–572. ACM Press (2013)
12. Franconi, E., Guagliardo, P.: The view update problem revisited. CoRR Technical Report arXiv:1211.3016, arXiv.org e-Print archive (2012), available at `http://arxiv.org/abs/1211.3016`
13. Guo, Y., Pan, Z., Heflin, J.: LUBM: A benchmark for OWL knowledge base systems. J. of Web Semantics 3(2–3), 158–182 (2005)
14. Guo, Y., Qasem, A., Pan, Z., Heflin, J.: A requirements driven framework for benchmarking semantic web knowledge base systems. IEEE TKDE 19(2), 297–309 (2007)
15. Keet, C.M., Alberts, R., Gerber, A., Chimamiwa, G.: Enhancing web portals with Ontology-Based Data Access: the case study of South Africa's Accessibility Portal for people with disabilities. In: Proc. of OWLED. CEUR, `ceur-ws.org`, vol. 432 (2008)
16. Kontchakov, R., Lutz, C., Toman, D., Wolter, F., Zakharyaschev, M.: The combined approach to query answering in *DL-Lite*. In: Proc. of KR. pp. 247–257 (2010)
17. LePendu, P., Noy, N.F., Jonquet, C., Alexander, P.R., Shah, N.H., Musen, M.A.: Optimize first, buy later: Analyzing metrics to ramp-up very large knowledge bases. In: Proc. of ISWC. LNCS, vol. 6496, pp. 486–501. Springer (2010)
18. Mora, J., Corcho, O.: Towards a systematic benchmarking of ontology-based query rewriting systems. In: Proc. of ISWC. LNCS, vol. 8218, pp. 369–384. Springer (2013)
19. Morsey, M., Lehmann, J., Auer, S., Ngonga Ngomo, A.C.: DBpedia SPARQL Benchmark – Performance assessment with real queries on real data. In: Proc. of ISWC, Volume 1. LNCS, vol. 7031, pp. 454–469. Springer (2011)
20. Rodríguez-Muro, M., Calvanese, D.: Dependencies: Making ontology based data access work in practice. In: Proc. of AMW. CEUR, `ceur-ws.org`, vol. 749 (2011)
21. Rodriguez-Muro, M., Kontchakov, R., Zakharyaschev, M.: Ontology-based data access: Ontop of databases. In: Proc. of ISWC. LNCS, vol. 8218, pp. 558–573. Springer (2013)
22. Skjæveland, M.G., Lian, E.H.: Benefits of publishing the Norwegian Petroleum Directorate's FactPages as Linked Open Data. In: Proc. of Norsk informatikkonferanse (NIK 2013). Tapir (2013)
23. Swami, A., Schiefer, K.B.: On the estimation of join result sizes. In: Proc. of EDBT. LNCS, vol. 779, pp. 287–300. Springer (1994)
24. Venetis, T., Stoilos, G., Stamou, G.B.: Query extensions and incremental query rewriting for OWL 2 QL ontologies. J. on Data Semantics 3(1), 1–23 (2014)
25. Wang, S.Y., Guo, Y., Qasem, A., Heflin, J.: Rapid benchmarking for semantic web knowledge base systems. In: Proc. of ISWC. LNCS, vol. 3729, pp. 758–772. Springer (2005)

Scheduling for SPARQL Endpoints

Fadi Maali, Islam A. Hassan, and Stefan Decker

Insight Centre for Data Analytics, National University of Ireland Galway
{firstname.lastname}@insight-centre.org

Abstract. When providing public access to data on the Semantic Web, publishers have various options that include downloadable dumps, Web APIs, and SPARQL endpoints. Each of these methods is most suitable for particular scenarios. SPARQL provides the richest access capabilities and is the most suitable option when granular access to the data is needed. However, SPARQL expressivity comes at the expense of high evaluation cost. The potentially large variance in the cost of different SPARQL queries makes guaranteeing consistently good quality of service a very difficult task. Current practices to enhance the reliability of SPARQL endpoints, such as query timeouts and limiting the number of results returned, are far from ideal. They can result in under utilisation of resources by rejecting some queries even when the available resources are sitting idle and they do not isolate "well-behaved" users from "ill-behaved" ones and do not ensure fair sharing among different users. In similar scenarios, where unpredictable contention for resources exists, scheduling algorithms have proven to be effective and to significantly enhance the allocation of resources. To the best of our knowledge, using scheduling algorithms to organise query execution at SPARQL endpoints has not been studied. In this paper, we study, and evaluate through simulation, the applicability of a few algorithms to scheduling queries received at a SPARQL endpoint.

1 Introduction

When providing public access to data on the Semantic Web, publishers have various options that include downloadable dumps, Web APIs, and SPARQL endpoints. Each of these methods is most suitable for particular scenarios, however none of them provides an ideal global solution [7, 13]. SPARQL, the recommended W3C query language[1], is an attractive option to provide expressive access to RDF data. SPARQL is basically a graph pattern matching language that provides rich capabilities for slicing and dicing RDF data. The latest version, SPARQL 1.1, added support for aggregation, nested and distributed queries, and other features.

However, supporting public SPARQL access to data is expensive. It has been shown that evaluating SPARQL is PSPACE-complete in general and coNP-complete for well-defined queries [10]. Therefore, the cost of different SPARQL

[1] http://www.w3.org/TR/sparql11-query/

queries can vary a lot; making guaranteeing consistently good quality of service a very difficult task. Evidence of this can be seen on the SPARQL Endpoint Status web page[2], in literature [2] and across the Web[3] and the blogosphere[4].

Existing SPARQL endpoints employ different measures to enhance their reliability and to ensure consistent quality of service. Such measures include query timeouts, refusing expensive SPARQL queries, limiting the number of triples returned or returning partial results. For example, 4Store supports a soft limit for execution time[5] and Virtuoso allows setting a maximum threshhold on the expected query cost[6]. There are still a number of problems with these approaches: (i) they provide an inconsistent user experience and limit the expressiveness of allowed queries (ii) there is no clear way to communicate these non-standard shortcomings to the user (iii) they can result in under utilisation of resources by rejecting some queries even when the available resources are sitting idle (iv) they do not isolate "well-behaved" users from "ill-behaved" ones and do not ensure fair sharing among different users.

In similar scenarios, where unpredictable contention for resources exists, scheduling algorithms have proven to be effective and to significantly enhance the allocation of resources. Scheduling has been utilised for data networks [3, 12], for processes assignment in operating systems [8], for cloud and grid computing [9, 4], and recently to schedule jobs sent to a Hadoop cluster [15, 14]. Nevertheless, to the best of our knowledge, it has not been studied in the context of SPARQL endpoints.

In this paper we argue for employing scheduling algorithms to organise query execution at a, possibly public, SPARQL endpoint. Giving the wide applicability of scheduling, there exists a large number of scheduling algorithms. In this paper, we study, and evaluate through a simulation, the applicability of a few algorithms to schedule queries received at some SPARQL endpoint. A number of scheduling algorithms, mainly from the data networks domain, are reviewd (Section 3.1). We then describe their applicability to scheduling SPARQL queries (Section 3.2) and study through a simulation their effect on two popular SPARQL engines (Section 4).

We do not claim that scheduling solves the problem of providing a reliable publicly-accessible SPARQL endpoint. Nevertheless, our results show that scheduling enhances throughput and reduces the effect of complex queries on simpler ones. We also note that scheduling has the extra advantages of rewarding socially-aware behaviour and achieving better utilisation and fairer allocation of the available resources.

---

[2] http://sparqles.okfn.org/

[3] See for example http://answers.semanticweb.com/questions/14440/can-the-economic-problem-of-shared-sparql-endpoints-be-solved

[4] E.g. http://daverog.wordpress.com/2013/06/04/the-enduring-myth-of-the-sparql-endpoint/ and http://ruben.verborgh.org/blog/2013/09/30/can-i-sparql-your-endpoint/

[5] http://4store.org/trac/wiki/SparqlServer

[6] http://virtuoso.openlinksw.com/dataspace/doc/dav/wiki/Main/VirtSPARQLEndpointProtection

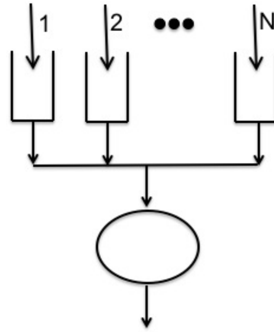


Fig. 1: General model of scheduling

2 Related Work

Most current practices to enhance the reliability of SPARQL endpoints are based on introducing ad-hoc measures and limits such as query timeouts, refusing expensive SPARQL queries, limiting the number of triples returned or returning partial results. These measures have a number of problems as discussed in the introduction. Linked Data Fragments [13] is a recent proposal to enhance the reliability of SPARQL endpoints via shifting part of the computation needed to answer SPARQL queries towards the client side. Furthermore, [1] proposed a vision for a workload-aware and adaptive system to deal with the diversity and dynamism that are inherent in SPARQL workloads. To the best of our knowledge, scheduling has not been studied in the context of SPARQL endpoints. Nevertheless, scheduling has been used and studied in many domains.

Scheduling has been extensively studied in data networks where the capacity of a switch is shared by multiple senders [3, 6, 12]. We describe the main algorithms used in data networks in the next section.

Scheduling has also been used to organise jobs of shared Hadoop clusters. First In First Out (FIFO) Scheduler, Fair Scheduler[7] and Capacity Scheduler[8] are the most widely used schedulers in practice. Similar to our work, these schedulers re-use algorithms defined for the data networks. Notice that, in contrast to SPARQL queries, Hadoop jobs are expected to take a long time and their execution can be pre-empted.

Moreover, scheduling has also been applied in wireless networks [5], grid computing [4] and distributed hash tables [11].

3 Scheduling

Figure 1 depicts a generic model for scheduling. There is a single server, and N job arrival streams, each feeding a different first in, first out (FIFO) queue.

[7] `http://hadoop.apache.org/docs/r1.2.1/fair_scheduler.html`
[8] `http://hadoop.apache.org/docs/r1.2.1/capacity_scheduler.html`

The scheduler decides which stream gets served at each moment and for how long. The goal is usually to maximise throughput while ensuring fair sharing of resources and avoiding long waiting times. Keeping a separate queue for each stream puts up "firewalls" protecting well-behaved streams against streams that might otherwise saturate the server capacity and drive delays to unacceptable levels.

3.1 Scheduling Algorithms

We next describe a number of scheduling algorithms used mainly for data networks and then discuss their applicability to SPARQL endpoints.

Ideal Scheduler A mathematical idealization of queuing, which was originally proposed as an idealization of time-slicing in computer systems (Kleinrock 1976 as cited in [6]), known as Processor Sharing (PS). In PS, the server cycles through the active jobs, giving each a small time quantum of service, and then preempting the job to work on the next. The PS mechanism allots resources fairly and provides full utilisation of the resources. However, in settings where pre-empting jobs is not feasible, such as data networks, PS cannot be applied. A number of "emulations" of it exist nevertheless. We next discuss two of them.

Fair Scheduler Described in [3] and [6]. Fair scheduling emulates the PS fair scheduling by maintaining a notion of virtual time (what time an input stream would have been served had PS been applied). Streams are then served in increasing order of their virtual finish time. It has been shown that this algorithm emulates the fair PS algorithm well [6].

Deficit Round-Robin Scheduler Described in [12]. Each input stream holds a deficit counter (a credit balance) and only gets served if the query cost is less than its balance. After execution, the cost is subtracted from the balance. If the queue is not served due to insufficient balance, it gets a quantum charge that can be used in the next round, i.e. a queue is compensated when its service is delayed.

3.2 Scheduling for SPARQL

We consider scheduling as a service running on top of SPARQL endpoints. The scheduler receives the queries and then decides in what order to send them to the endpoint. The goal is to: (i) minimize the effect that expensive queries can have on other queries (ii) maximize the utilization of the available resources (iii) reward socially-aware behaviour (e.g., simpler queries that set a limit on the number of required results).

However, contrary to the typical scheduling scenario depicted in Figure 1, a SPARQL endpoint can handle multiple queries at a time. In fact, as triple

stores, Web servers and the underlying operating systems each have their own resource utilisation and sharing facilities, it will be very inefficient to send only one query at a time to the endpoint. Therefore, SPARQL scheduler sends multiple queries to the endpoint as long as their total cost is under a configured threshold. Upon the completion of processing some query, its cost is substracted from the tracked total cost. The threshold is necessary to avoid overwhelming the endpoint as sending many queries simultaneously to an endpoint results in rejecting to process many queries by the endpoint.

In practice, three further challenges need to be addressed:

- Efficiently estimating the cost of an incoming SPARQL query. The cost involves multiple paramters such as CPU, memory, network and I/O cost. The cost also depends on the query, the data and the triple store. However, this problem is beyond the scope of this paper.
- Identifying the source of each query in order to define streams of inputs. In the absence of user authentication, IP addresses and session detection techniques can be used.
- Setting the threshold of total computing capacity. This can be set via experimenting.

4 Simulation Experiment

4.1 Implementation

We implemented three scheduling algorithms in Java [9]:

- **FIFO**: A First-In-First-Out queue. This scheduler serves input streams in order (one query at a time) while keeping track of the total cost of queries being processed at every point and ensuring that this cost is always kept lower than the computing capacity threshold.
- **Deficit**: Implements a deficit round-robin algorithm as described in Section 3.1.
- **Fair**: Implements a fair scheduling algorithm as described in Section 3.1.

4.2 Experiment Setup

We experimented with two triple stores, Virtuoso Open-Source Edition 7.0.0 Release[10] and Jena Fuseki 1.0.1[11]. Both triple stores were run on a Mac with 8GB memory and a 2.9GHz Intel Core i7 CPU. Jena Fuseki was run in memory with a maximum heap size of 3GB.

We used the Semantic Web Dog Food[12] data that contains information about papers published in the main conferences and workshops in the area of Semantic

[9] The code is avaialble at `https://gitlab.insight-centre.org/Maali/sparql-endpoints-scheduler`
[10] `http://virtuoso.openlinksw.com/`
[11] `http://jena.apache.org/documentation/serving_data/`
[12] `http://data.semanticweb.org/`

Web research. The data was downloaded[13] from the Semantic Web Dog Food website in Februaury 2014.

The Web logs of the Semantic Web Dog Food were made available as part of the USEWOD 2013 Data Challenge[14]. To get real-world SPARQL queries for our experiment, we extracted SPARQL queries from these Web logs. We ran each query three times on Fuseki (without any added scheduling) and classified it as simple or complex based on the time it took. All simple queries took an average of less than 90ms, while complex queries took more than 450ms. We use these numbers (90 and 450) as a proxy for the query costs.

We simulated two concurrent flows of queries, one with simple queries and the other with complex queries. This simulates two applications with different needs. Queries were selected randomly from the set of simple and complex queries respectively. Both flows follow a Poisson distribution. We experimented with differnet means of the Poisson distribution (a.k.a. interval) and with different thesholds for the computing capacity (i.e., total cost of queries being processed at a time). Each run was stopped after 5 minutes and logs were collected then.

We consider a query to be fully processed if it is sent to the endpoint and all results sent back from the endpoint are received (within the 5 minute cut-off). For each query we measure two durations:

Waiting Time: the time taken from the moment the query is received at the scheduler until it is sent to the endpoint.

Processing Time: the time taken from the moment the query is received at the scheduler until the moment at which the query is fully processed.

We wanted to include a no-scheduling setting as a baseline, however running these flows without any scheduling resulted in overwhelming the endpoint and therefore most of the queries were dropped without getting answered.

4.3 Results & Discussion

We experimented with intervals of 100, 200, 500 and 1000 milliseconds (ms). Queries sent with intervals 500 and 1000 ms were not frequent enough to require any waiting and therefore resulted in no scheduling. We report only on queries with intervals 100 and 200 ms. We experimented with different thresholds for the computing capacity. We report here only on the results when the threshold is set to 8000 because it showed the most effective results for the three scheduling algorithms tested; all larger values overwhelmed the endpoint. Notice that 8000 is just a proxy for the computation capacity available and it needs to be interpreted together with the cost of the queries (we used 90 and 450 for simple and complex queries respectively).

Table 1 shows the percentage of queries completed when sent at a 100 ms interval on Fuseki. On Virtuoso all queries were fully processed, while on Fuseki it can be noticed that the FIFO algorithm processed equivalent percentages of

[13] `http://data.semanticweb.org/dumps/`

[14] `http://data.semanticweb.org/usewod/2013/challenge.html`

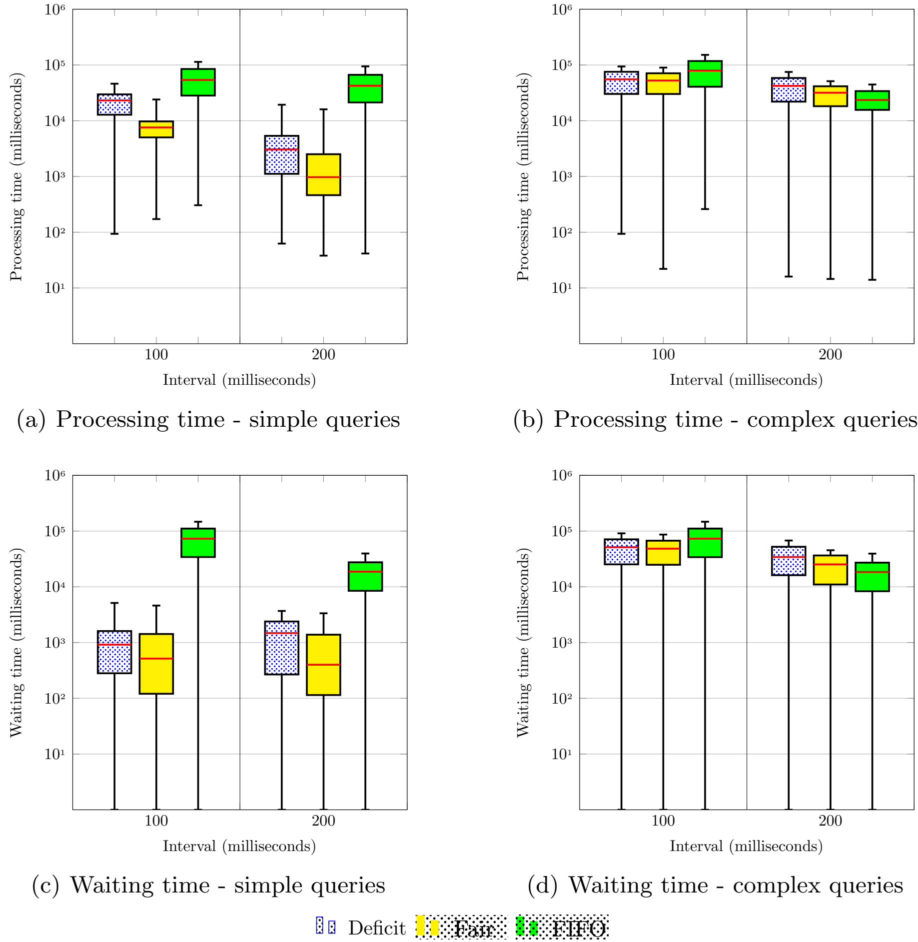
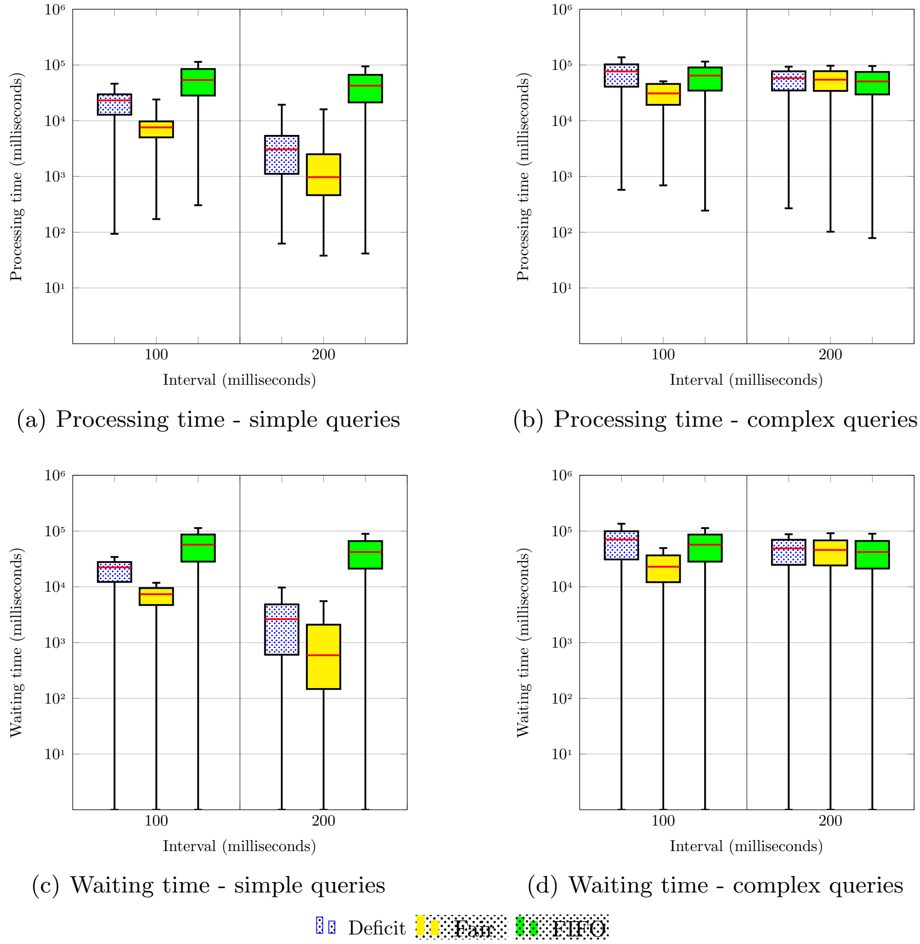


Fig. 2: Virtuoso - Processing and waiting times

both simple and complex queries while the other two algorithms “favoured” simple queries and “penalised” complex ones. Deficit scheduling showed better throughput.

Figures 2 and Figure 3 show the processing and waiting times for simple and complex queries when running against Virtuoso and Fuseki. Similar to the throughput results, it can be noticed how penalising complex queries results in smaller waiting and processing times for simple queries when using deficit or fair scheduling. On the other hand, complex queries have smaller waiting and processing times using FIFO scheduling with 200 milliseconds interval. The higher processing time of complex queries using FIFO scheduling was surprising. Our interpretaion of this is that prioritising simple queries allowed better utilisa-



(a) Processing time - simple queries

(b) Processing time - complex queries

(c) Waiting time - simple queries

(d) Waiting time - complex queries

Fig. 3: Fuseki - Processing and waiting times

tion of the available resources when queries arrive at a high frequency that can overwhelm the endpoint.

In summary, scheduling allowed getting better throughput by delaying queries instead of rejecting them (recall that running without scheduling resulted in rejecting most of the queries). Deficit and Fair scheduling algorithms favoured simpler queries. In general, deficit scheduling, the simpler algorithm, showed better results in our settings than fair scheduling.

	Simple Queries	Complex Queries
FIFO	47.3%	47.4%
Deficit	100%	42%
Fair	77%	40%

Table 1: Percentage of fully processed queries on Fuseki (throughput)

5 Conclusions & Future Work

We reported a simulation experiment to study the effects different scheduling algorithms can have when used to organise execution of SPARQL queries received at some endpoint. We note that simple scheduling can be implemented with minimal overhead and has effect only when queries are received at high frequency.

We consider this work as an initial step and hope to extend the experiment and deploy it in some real-world use case.

Acknowledgements. Fadi Maali is funded by the Irish Research Council, Embark Postgraduate Scholarship Scheme. This publication has emanated from research supported in part by a research grant from Science Foundation Ireland (SFI) under Grant Number SFI/12/RC/2289.

References

1. G. Aluc, M. T. Özsu, and K. Daudjee. Workload matters: Why rdf databases need a new design. *PVLDB*, 7(10):837–840, 2014.
2. C. Buil-Aranda, A. Hogan, J. Umbrich, and P.-Y. Vandenbussche. SPARQL Web-Querying Infrastructure: Ready for Action? In *ISWC 2013*, pages 277–293. Springer, 2013.
3. A. Demers, S. Keshav, and S. Shenker. Analysis and simulation of a fair queueing algorithm. In *ACM SIGCOMM Computer Communication Review*, volume 19, pages 1–12. ACM, 1989.
4. F. Dong and S. G. Akl. Scheduling Algorithms for Grid Computing: State of the Art and Open Problems. *School of Computing, Queen's University, Kingston, Ontario*, 2006.
5. H. Fattah and C. Leung. An Overview of Scheduling Algorithms in Wireless Multimedia Networks. *Wireless Communications, IEEE*, 9(5):76–83, 2002.
6. A. G. Greenberg and N. Madras. How fair is fair queuing. *Journal of the ACM (JACM)*, 39(3):568–598, 1992.
7. A. Hogan and C. Gutierrez. Paths towards the Sustainable Consumption of Semantic Data on the Web. In *AMW*, 2014.
8. L. Kleinrock. Queueing systems, volume II: Computer applications. 1976.
9. H. Kllapi, E. Sitaridi, M. M. Tsangaris, and Y. Ioannidis. Schedule Optimization for Data Processing Flows on the Cloud. In *Proceedings of the 2011 ACM SIGMOD International Conference on Management of data*, pages 289–300. ACM, 2011.
10. J. Pérez, M. Arenas, and C. Gutierrez. Semantics and complexity of SPARQL. *ACM Transactions on Database Systems (TODS)*, 34(3):16, 2009.

11. S. Rhea, B. Godfrey, B. Karp, J. Kubiatowicz, S. Ratnasamy, S. Shenker, I. Stoica, and H. Yu. OpenDHT: A Public DHT Service and its Uses. In *ACM SIGCOMM Computer Communication Review*, volume 35, pages 73–84. ACM, 2005.
12. M. Shreedhar and G. Varghese. Efficient fair queuing using deficit round-robin. *Networking, IEEE/ACM Transactions on*, 4(3):375–385, 1996.
13. R. Verborgh, M. Vander Sande, P. Colpaert, S. Coppens, E. Mannens, and R. Van de Walle. Web-scale Querying through Linked Data Fragments. In *Proceedings of the 7th Workshop on Linked Data on the Web*, 2014.
14. M. Yong, N. Garegrat, and S. Mohan. Towards a Resource Aware Scheduler in Hadoop. In *Proc. ICWS*, pages 102–109, 2009.
15. M. Zaharia. Job scheduling with the fair and capacity schedulers. *Hadoop Summit*, 9, 2009.

Querying Distributed RDF Graphs: The Effects of Partitioning

Anthony Potter, Boris Motik, and Ian Horrocks

Oxford University
first.last@cs.ox.ac.uk

Abstract. Web-scale RDF datasets are increasingly processed using distributed RDF data stores built on top of a cluster of shared-nothing servers. Such systems critically rely on their data partitioning scheme and query answering scheme, the goal of which is to facilitate correct and efficient query processing. Existing data partitioning schemes are commonly based on hashing or graph partitioning techniques. The latter techniques split a dataset in a way that minimises the number of connections between the resulting subsets, thus reducing the need for communication between servers; however, to facilitate efficient query answering, considerable duplication of data at the intersection between subsets is often needed. Building upon the known graph partitioning approaches, in this paper we present a novel data partitioning scheme that employs minimal duplication and keeps track of the connections between partition elements; moreover, we propose a query answering scheme that uses this additional information to correctly answer all queries. We show experimentally that, on certain well-known RDF benchmarks, our data partitioning scheme often allows more answers to be retrieved without distributed computation than the known schemes, and we show that our query answering scheme can efficiently answer many queries.

1 Introduction

While the flexibility of the RDF data model offers many advantages, efficient management of large RDF datasets remains an open research topic. RDF data management systems can be conceived as single-machine systems constructed using techniques originating from relational databases [12, 2, 1, 20, 4], but the size of some RDF datasets exceeds the capacity of such systems. As a possible solution, distributed architectures based on a cloud of shared-nothing servers have been developed [9, 11, 21]. Such systems are promising, but research is still at a relatively early stage and scalability remains an open and critical problem.

The two main challenges in developing a distributed RDF system are (i) how to split up the data across multiple servers (i.e., data partitioning), and (ii) how to answer queries in a distributed environment (i.e., distributed query answering). These two challenges are closely connected: knowledge about data partitioning is relevant for query answering, and knowledge of typical query structure can be used to inform the data partitioning scheme. Nevertheless, one can investigate independently from query answering the extent to which a specific data

partitioning scheme reduces the need for distributed processing; for example, one can identify the percentage of the query answers that can be computed locally—that is, by evaluating queries on each server independently.

Data partitioning via hashing is a well-known partitioning scheme from the database community, and it has been applied in several RDF systems [21, 15, 4, 8, 13, 7]. In its simplest form, it distributes RDF data in a cluster by applying a hash function to a component of an RDF triple. Triples are commonly hashed by their subject to guarantee that *star queries* (i.e., queries containing only subject–subject joins) can be evaluated locally. Hashing has often been implemented using the MapReduce framework [3, 5, 19]; although hashing is typically not discussed explicitly in such systems, it is implicit in the map phase of distributed join processing. This approach, however, does not take into account the graph structure of RDF data, so nodes that are close together in the graph may not be stored on the same server, resulting in considerable distributed processing for many queries. As an alternative, an approach based on graph partitioning has been developed [9]. The goal of graph partitioning is to divide the nodes of the graph into several subsets while minimising the number of links with endpoints in different subsets. Thus, by partitioning RDF data using graph partitioning, one increases the chance that highly interconnected nodes are placed on the same server, which in turn increases the likelihood that query answers can be computed locally. This scheme, however, does not guarantee that common queries (such as star queries) can be evaluated locally, and it may thus require a significant amount of distributed computation to guarantee completeness, even in cases where locally computed answers fully answer the query.

These approaches can be augmented by duplicating data on partition boundaries [9, 11]. For sufficiently small queries, data duplication ensures that each query answer can be computed locally, and it can be used to provide the local evaluation guarantee for star-shaped queries in the graph partitioning setting [9]. Data duplication, however, can incur considerable storage overhead, potentially increasing the number of servers required to store a given RDF graph.

Our Approach We present a novel RDF data partitioning scheme that aims to reduce the need for distributed computation on common queries, but with minimal duplication and storage overhead; moreover, we present a query answering scheme that can correctly answer conjunctive queries over the partitioned data. Our main idea is to keep track of places where each data subset makes connections to other data subsets, and to exploit this information during query answering in order to identify possible non-local answers. In this way we can enjoy the benefits of graph partitioning and reduce the need for distributed processing, with only a minimal overhead of data duplication.

In this paper we focus mainly on the effects of our data partitioning scheme, which we experimentally show to be very promising on the LUBM [6] and the SP2B [16] benchmarks. For all queries from these two benchmarks, our data partitioning scheme ensures that a higher proportion of query answers can be retrieved without any distributed processing than with subject-based hashing [8, 13, 7] or the semantic hash partitioning (SHAPE) [11] approach. We do not

explicitly compare our approach with the original graph partitioning approach [9] because the SHAPE approach has already been shown to be more effective.

We also experimentally evaluate our query answering scheme, which we show can effectively answer many queries from the LUBM and SP2B benchmarks with little or no distributed processing. On some queries, however, our scheme becomes impractical as it requires the computation of a large number of partial matches (see Section 4 for details)—a drawback we look to overcome in our future work. These results are preliminary in the absence of a complete distributed system that would allow us to measure query processing times; instead, we measure the amount of work (in a sense that we make precise in Section 4) involved in distributed query processing.

2 Preliminaries

2.1 RDF

Let R be a set of *resources*. An *RDF term* is either a variable or a resource from R; a term is *ground* if it is not a variable. An *RDF atom* A is an expression of the form $\langle s, p, o\rangle$ where s, p, and o are RDF terms; the *vocabulary* of A is defined as $\mathsf{voc}(A) = \{s, p, o\}$; $\mathsf{var}(A)$ is the set of all variables in $\mathsf{voc}(A)$; atom A is *ground* if $\mathsf{var}(A) = \emptyset$; a *triple* is a ground atom; and an *RDF graph* G is a set of triples. The *vocabulary* of G is defined as $\mathsf{voc}(G) = \bigcup_{A \in G} \mathsf{voc}(A)$. As in the SPARQL query language, all variables in this paper start with a question mark. A *variable assignment* μ (or just *assignment*) is a partial mapping of variables to resources. For r a resource, let $\mu(r) = r$; for $A = \langle s, p, o\rangle$ an RDF atom, let $\mu(A) = \langle \mu(s), \mu(p), \mu(o)\rangle$; and for S a set of atoms, let $\mu(S) = \bigcup_{A \in S} \mu(A)$. The *domain* $\mathsf{dom}(\mu)$ of μ is the set of variables that μ is defined on; and the *range* $\mathsf{rng}(\mu)$ of μ is $\mathsf{rng}(\mu) = \{\mu(x) \mid x \in \mathsf{dom}(\mu)\}$. An *RDF conjunctive query* Q is an expression of the form (1), where each A_i, $1 \leq i \leq m$, is an RDF atom.

$$Q = A_1 \wedge \ldots \wedge A_m \tag{1}$$

By a slight abuse of notation, we often identify Q with the set of its atoms. The vocabulary and the set of variables of Q are defined as follows.

$$\mathsf{voc}(Q) = \bigcup_{1 \leq i \leq m} \mathsf{voc}(A_i) \qquad\qquad \mathsf{var}(Q) = \bigcup_{1 \leq i \leq m} \mathsf{var}(A_i) \tag{2}$$

We sometimes write queries using the SPARQL syntax. An *answer* to a query Q over an RDF graph G is an assignment μ such that $\mathsf{dom}(\mu) = \mathsf{var}(Q)$ and $\mu(Q) \subseteq G$; and $\mathsf{ans}(Q, G)$ is the set of all answers to Q over G. Note that our definitions do not support variable projection.

2.2 RDF Data Partitioning and Distributed Query Answering

A *partition* of an RDF graph G is an n-tuple of RDF graphs $\mathbf{G} = (G_1, \ldots, G_n)$ such that $G \subseteq G_1 \cup \ldots \cup G_n$. Each G_i is called a *partition element*, and n is the

size of $\mathbf{G}$; an *n-partition* is a partition of size n. One might expect a partition to satisfy $G = G_1 \cup \ldots \cup G_n$, but our relaxed condition allows us to capture our ideas in Section 3. Furthermore, we allow triples to be duplicated across partition elements—that is, we do not require $G_i \cap G_j = \emptyset$ for $i \neq j$. A *partitioning scheme* is a process that, given an RDF graph G, produces a partition $\mathbf{G}$. Given an RDF conjunctive query Q, an answer $\mu \in \mathsf{ans}(Q, G)$ is *local for* Q *and* $\mathbf{G}$ if some $1 \leq i \leq n$ exists such that $\mu \in \mathsf{ans}(Q, G_i)$; otherwise, μ is *non-local for* Q *and* $\mathbf{G}$. When Q and $\mathbf{G}$ are clear, we simply call μ local or non-local. Since non-local answers span partition elements, they are more expensive to compute if each partition element is stored on a separate server; hence, the main aim of a partitioning scheme is to maximise the number of local answers to common queries. The *quality* of a partition $\mathbf{G}$ for a set of queries $\mathbf{Q}$ is defined as the ratio of local answers to all answers for all of the queries in $\mathbf{Q}$ on $\mathbf{G}$. We often use this term informally, in which case we consider partition quality with respect to an unspecified set of queries that can be considered typical.

Allowing duplication of triples in partition elements can improve partition quality: given a non-local answer μ to a query Q, answer μ becomes local if we add $\mu(Q)$ to some partition element. However, duplication also increases storage overhead, and in the limit it can result in each partition element containing a complete copy of G. Hence, another aim of a partitioning scheme is to achieve a suitable balance between triple duplication and partition quality.

A *distributed query answering scheme*, or just a *query answering scheme*, is a process that, given a partition $\mathbf{G}$ and a query Q, returns a set of assignments $\mathsf{ans}(Q, \mathbf{G})$ such that $\mathsf{ans}(Q, \mathbf{G}) = \mathsf{ans}(Q, G)$. If the shape of Q guarantees that all answers are local, one can evaluate Q against each element of $\mathbf{G}$ and take the union of all answers; otherwise, additional work is required to identify non-local answers or to detect that no such answers exist. Query answering schemes differ mainly in how they handle the latter case: answer pieces are spread across multiple partition elements and they must be retrieved and joined together. We now have two clear goals for a query answering scheme: the first goal is to ensure correctness—that is, that $\mathsf{ans}(Q, \mathbf{G}) = \mathsf{ans}(Q, G)$—and the second goal is to minimise the amount of work required to construct non-local answers.

2.3 Existing Solutions

Now we present a brief overview of partitioning schemes and query answering schemes known in the literature.

Hashing is the simplest and most common data partitioning scheme [14, 18, 21]. Typically, a hashing function maps triples of an RDF graph to m buckets, each of which corresponds to a partition element. The hashing function is often applied to the triple's subject or the predicate, with subject being the most popular choice: this guarantees that triples with the same subject are placed together, ensuring that all answers to star queries are local.

Graph-based approaches exploit the graph structure of RDF data. In particular, one can use min-cut graph partitioning software such as METIS [10], which takes as input a graph G and the partition size n, and outputs n disjoint sets

of nodes of G such that all sets are of similar sizes and the number of edges in G connecting nodes in distinct sets is minimised. The approach by [9] reduces RDF data partitioning to min-cut graph partitioning to ensure that highly connected RDF resources are placed together into the same partition element, thus increasing the likelihood of a query answer being local.

One can combine an arbitrary data partitioning scheme with *n-hop duplication* to increase the proportion of local answers. Given an RDF graph G and a subgraph $H \subseteq G$, the n-hop expansion H_n of H with respect to G is defined recursively as follows: $H_0 = H$ and, for each $1 \leq i \leq n$,

$$H_i = H_{i-1} \cup \{\langle s, p, o\rangle \mid \langle s, p, o\rangle \in G \text{ and } \{s, o\} \cap \mathsf{voc}(H_{i-1}) \neq \emptyset\}. \tag{3}$$

While n-hop partitioning can considerably improve partition quality [9], it can also incur a substantial storage overhead. For example, even just 2-hop duplication can incur a storage overhead ranging from 67% to 435% [11]. Various optimisations have been developed to reduce this overhead, such as using directed expansion and excluding high degree nodes from the expansion.

Most data partitioning schemes are paired with a specific query answering scheme. Due to lack of space, we cannot present all such approaches in detail. Many of them have been implemented using MapReduce [3]—a framework for handling and processing large amounts of data in parallel across a cluster; a recent survey of MapReduce solutions can be found in [5]. Moreover, the Trinity.RDF system [21] uses Trinity [17]—a distributed in-memory key-value store.

3 Partitioning RDF Data

3.1 Aims

We now present our novel data partitioning scheme. Similar to [9], we use min-cut graph partitioning, but we extend the approach by recording the outgoing links in each partition element so as to facilitate the reconstruction of non-local answers. More specifically, we introduce a *wildcard* resource $*$, and use it to represent all resources 'external' to a given partition element. Thus, we know in each partition element which resources are connected to resources in other partition elements; we exploit this feature in Section 4 in order to obtain a correct query answering scheme. This allows us to attain a high degree of partition quality, while at the same time answering queries correctly without n-hop duplication.

The quality of partitions critically depends on the structure of the data and the anticipated query workload. Although application specific, we found the following assumptions to be common to a large number of applications.

Assumption 1. Subject–subject joins are common.

Assumption 2. Queries often constrain variables to elements of classes—that is, they often contain atoms of the form $\langle ?x, \mathit{rdf}{:}\mathit{type}, \mathit{class}\rangle$.

Assumption 3. Joins involving resources representing classes are uncommon—that is, queries rarely contain atoms $\langle ?x_1, \mathit{rdf}{:}\mathit{type}, ?y\rangle \wedge \langle ?x_2, \mathit{rdf}{:}\mathit{type}, ?y\rangle$.

Assumption 4. Joins on resources that are literals are uncommon—that is, if a query contains atoms $\langle ?x_1, {:}R, ?y\rangle \wedge \langle ?x_2, {:}S, ?y\rangle$, it is unlikely that a query answer will map variable $?y$ to a literal.

Assumption 5. The number of schema triples in G is small, so all schema triples can be replicated in each partition element.

Although n-hop duplication can increase partition quality [9, 11], it is often associated with a considerable storage overhead, particularly with real (as opposed to synthetic) RDF graphs. With this in mind, we formulate the following aims for our partitioning scheme:

Aim 1. maximise the number of local answers to star queries,

Aim 2. achieve similar, or better, partition quality than schemes employing n-hop duplication, and

Aim 3. minimise duplication, particularly compared to n-hop duplication.

3.2 Our Data Partitioning Scheme

Given an RDF graph G, let $*$ be a distinguished *wildcard* resource such that $* \notin \mathsf{voc}(G)$. Now let $V \subseteq \mathsf{voc}(G)$ be a subset of the vocabulary of G. Given a resource r, let $[r]_V = r$ if $r \in V$ and $[r]_V = *$ otherwise. Moreover, given an RDF atom $A = \langle s, p, o\rangle$, let $[A]_V = \langle [s]_V, [p]_V, [o]_V\rangle$. Finally, given an RDF graph G, let $[G]_V$ be the RDF graph defined by $[G]_V = \{[A]_V \mid A \in G\}$. In the rest of this section we formalise our data partitioning scheme, and in Section 4 we show how to use the wildcard resource to answer queries.

Instead of partitioning triples directly, we partition the vocabulary of the graph and use the result to construct a partition of the triples. More precisely, to construct an n-partition of G we first partition $\mathsf{voc}(G)$ into n subsets $V_1, \ldots, V_n$, and then we use these to construct a partition $\mathbf{G} = ([G]_{V_1}, \ldots, [G]_{V_n})$ of G. To ensure that $\mathbf{G}$ is a valid partition, we must select $V_1, \ldots, V_n$ such that

$$G \subseteq [G]_{V_1} \cup \ldots \cup [G]_{V_n} \tag{4}$$

holds. To achieve this, we first partition the vocabulary $\mathsf{voc}(G)$ into n disjoint sets $V'_1, \ldots, V'_n$, and then we extend these sets so that condition (4) is satisfied. This extension allows resources to be duplicated in multiple partition elements, which in turn means triples can also be duplicated. Typically, the duplicated triples are those that are on, or near, the border between partition elements. Since resource $*$ is not contained in $\mathsf{voc}(G)$, we use the subset relation in condition (4), rather than a more intuitive equality relation. Furthermore, our approach ensures that, for each partition element $G_i = [G]_{V_i}$ and each triple $\langle s, p, o\rangle \in G$, if $\{s, p, o\} \subseteq \mathsf{voc}(G_i)$, then $\langle s, p, o\rangle \in G_i$ holds. This property is not satisfied in previously known partitioning schemes, but it increases partition quality. We formalise these ideas using the following steps.

Step 1. Compute the undirected graph G' by removing from G all schema triples and triples containing class and literal resources (i.e., all triples of the form $\langle s, \mathit{rdf{:}type}, o\rangle$ and $\langle s, p, \ell\rangle$ with ℓ a literal), and by treating each remaining triple $\langle s, p, o\rangle$ as an undirected edge connecting s and o.

Step 2. Partition the nodes of G' into n disjoint sets using min-cut graph partitioning (e.g., using METIS), and let $V'_1, \ldots, V'_n$ be the resulting vocabularies.
Step 3. Extend each V'_i to $V^\star_i = V'_i \cup \{r \mid r$ occurs in a schema triple in $G\}$.
Step 4. Extend each $V^\star_i$ to $V_i = V^\star_i \cup \{o \mid \langle s, p, o\rangle \in G$ and $s \in V^\star_i\}$.
Step 5. Calculate $[G]_{V_i}$ for each V_i, and set $\mathbf{G} = \{[G]_{V_1}, \ldots, [G]_{V_n}\}$.

In Step 1, we takes into account Assumption 5 that schema triples can be replicated in each partition element. Furthermore, in line with our Assumptions 3 and 4 on our query workload, we do not expect triples to participate in joins on classes and literals; thus, we remove such triples in Step 1 so that the min-cut graph partitioning algorithm does not attempt to place resources connected via class or literal resources into the same partition element.

In order to satisfy (4), we must ensure that, for each triple $A \in G$, some V_i exists such that $\mathsf{voc}(A) \in V_i$. Thus, we must reintroduce the triples from G that correspond to edges in G' that were 'cut' during min-cut partitioning, as well as triples removed in Step 1. Thus, in Step 3 we introduce all resources occurring in the schema (including all classes and properties) into all partition elements; note that this ensures an efficient evaluation of queries mentioned in Assumption 2. Finally, since we assume that subject–subject joins are common (cf. Assumption 1), in Step 4 we reintroduce the missing triples into the partition element that contains the triple subject.

Partition element $[G]_{V_i}$ is the *core owner* of a resource r if $r \in V'_i$. Note that, if $[G]_{V_i}$ is the core owner of a resource r, then $[G]_{V_i}$ contains all triples in which r occurs in the subject position. Hence, if Q is a star query in which variable $?x$ participates in subject–subject joins, then all answers in which $?x$ is mapped to r can be obtained by evaluating Q in $[G]_{V_i}$; in other words, all answers to star queries are local, which is in line with our Aim 1.

3.3 An Example

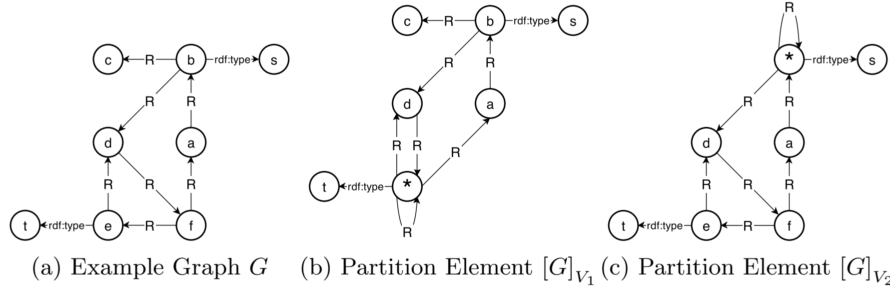
To make our scheme clear, we present an extended example. Let G be the RDF graph containing the following eight triples, shown schematically in Figure 1a.

$$G = \{\ \langle a, R, b\rangle, \langle b, R, c\rangle, \langle b, R, d\rangle, \langle d, R, f\rangle, \langle e, R, d\rangle, \\ \langle f, R, a\rangle, \langle f, R, e\rangle, \langle b, \mathit{rdf{:}type}, s\rangle, \langle e, \mathit{rdf{:}type}, t\rangle\} \quad (5)$$

To produce a 2-partition of G, in Step 1 we first remove all triples containing class and literal resources; in our example, we remove triples $\langle b, \mathit{rdf{:}type}, s\rangle$ and $\langle d, \mathit{rdf{:}type}, t\rangle$. In Step 2 we then apply min-cut graph partitioning to the resulting graph to split the resources into two sets while minimising the number of cut edges; let us assume that this produces the following vocabularies:

$$V'_1 = \{a, b, c\} \qquad V'_2 = \{d, e, f\} \quad (6)$$

In Steps 3 and 4 we then extend these vocabularies so that each partition element that is a core owner of a subject also contains all triples with that subject, and



(a) Example Graph G (b) Partition Element $[G]_{V_1}$ (c) Partition Element $[G]_{V_2}$

Fig. 1: Example Graph G and the Resulting Partitioning Elements

so that each partition element includes all class and property resources; in our example, this produces the following vocabularies:

$$V_1 = \{a, b, c, d, s, t, R, rdf{:}type\} \qquad V_2 = \{a, d, e, f, s, t, R, rdf{:}type\} \tag{7}$$

Due to this step, nodes a, d, s and t as duplicated in V_1 and V_2; we explain using node d why this is necessary. Graph $[G]_{V_1}$ must contain all triples whose subject is in V_1'; thus, since $\langle b, R, d\rangle$ is in G and b is in V_1' , we must add d to V_1. Finally, we construct $[G]_{V_1}$ and $[G]_{V_2}$ as shown in Figures 1b and 1c.

4 Distributed Query Answering

Although there is considerable variation in the details, existing query answering schemes, such as [9, 11], generally proceed via the following steps: a query is broken up into pieces, all of which can be evaluated independently within partition elements; each query piece is evaluated in the relevant partition element to obtain partial matches; and the partial matches are then joined into query answers. As an example, consider the following query:

$$Q = \langle ?x, rdf{:}type, s\rangle \wedge \langle ?x, R, ?y\rangle \wedge \langle ?z, R, ?x\rangle \tag{8}$$

The data partitioning scheme critically governs the first step. For example, if the data partitioning scheme guarantees that subject–subject joins can be evaluated locally, then the query must be broken up into pieces each of which involves only subject–subject joins; thus, query Q will be broken into the following pieces:

$$Q_1 = \langle ?x, rdf{:}type, s\rangle \wedge \langle ?x, R, ?y\rangle \qquad Q_2 = \langle ?z, R, ?x\rangle \tag{9}$$

If the data partitioning scheme employs n-hop duplication, one can break the query into pieces that involve joins with n hops; however, as we show in Section 5, duplication can incur a considerable storage overhead.

The main drawback of such approaches is that they do not take advantage of local answers. For example, answer $\mu = \{?x \mapsto b, ?y \mapsto d, ?z \mapsto b\}$ is local with

respect to the partition $[G]_{V_1}$ shown in Figure 1b, but it would not be retrieved by evaluating Q on $[G]_{V_1}$ directly; instead, one must evaluate Q_1 and Q_2 on $[G]_{V_1}$ and then join the results. This can be problematical since evaluating query pieces might be less efficient than evaluating the entire query at once.

Our query answering scheme uses a completely different approach. Roughly speaking, we first evaluate each query in each partition element independently, thus retrieving all local answers without any partial query evaluation or communication between the servers. However, in this step we may also retrieve answers containing the wildcard resource, each of which represents a potential match of the query across partition elements, so we join such answers to obtain all answers to the query. In this way, we restrict communication and partial query evaluation to (possible) non-local answers, rather than all answers.

4.1 Formalisation

To formalise our query answering scheme, we first introduce some notation. Let V be a vocabulary not containing $*$. For Q a conjunctive query of the form (1), let $[Q]_V = [A_1]_V \wedge \ldots \wedge [A_m]_V$. Furthermore, for μ a variable assignment, let $[\mu]_V$ be the variable assignment such that $\mathsf{dom}([\mu]_V) = \mathsf{dom}(\mu)$ and, for each variable $x \in \mathsf{dom}(\mu)$, we have $[\mu]_V(x) = \mu(x)$ if $\mu(x) \in V$, and $[\mu]_V(x) = *$ if $\mu(x) \notin V$.

In the rest of this section, we fix an arbitrary conjunctive query Q of the form (1) with m atoms, an arbitrary RDF graph G, and an arbitrary partition $\mathbf{G} = ([G]_{V_1}, \ldots, [G]_{V_n})$ of G. To evaluate Q in $\mathbf{G}$, we first evaluate $[Q]_{V_i}$ in $[G]_{V_i}$ for each $1 \leq i \leq n$. Note that query Q may contain resources not contained in V_i; therefore, in each partition element $[G]_{V_i}$ we evaluate $[Q]_{V_i}$, rather than Q. We then join all answers obtained in the previous step, while assuming that resource $*$ matches any other resource. We formalise the join procedure as follows.

Definition 1. *A variable assignment μ is a* join *of assignments μ_1 and μ_2, written $\mu = \mu_1 \bowtie \mu_2$, if $\mathsf{dom}(\mu) = \mathsf{dom}(\mu_1) = \mathsf{dom}(\mu_2)$ and, for each $x \in \mathsf{dom}(\mu)$, (i) $\mu_1(x) = \mu_2(x)$ implies $\mu(x) = \mu_1(x) = \mu_2(x)$, and (ii) $\mu_1(x) \neq \mu_2(x)$ implies $\mu_1(x) = *$ and $\mu(x) = \mu_2(x)$, or $\mu_2(x) = *$ and $\mu(x) = \mu_1(x)$.*

An assignment can be an answer to Q on $\mathbf{G}$ only if it instantiates all atoms of Q, which we formalise as follows.

Definition 2. *Let μ be a variable assignment with $\mathsf{dom}(\mu) = \mathsf{var}(Q)$. The set of* valid atoms *of Q under μ is defined as $\mathsf{val}_\mu(Q) = \{j \mid * \notin \mathsf{voc}(\mu(A_j))\}$. Moreover, μ is* valid *for Q if $|\mathsf{val}_\mu(Q)| = m$.*

Furthermore, when evaluating $[Q]_{V_i}$ in $[G]_{V_i}$, we can ignore any variable assignment $\mu_1 \in \mathsf{ans}([Q]_{V_i}, [G]_{V_i})$ that is redundant according to the following definition. Intuitively, μ_1 is redundant if, for each $\mu \in \mathsf{ans}(Q, G)$ that 'extends' μ_1 (i.e., such that $\mu_1 = [\mu]_{V_i}$), there exists a variable assignment μ_2 obtained by evaluating $[Q]_{V_j}$ in some partition element $[G]_{V_j}$ such that μ extends μ_2, and the set of atoms of Q fully instantiated by μ_2 strictly includes the set of atoms fully instantiated by μ_1. Note that this includes the case where no $\mu \in \mathsf{ans}(Q, G)$ extends μ_1. This idea is formally captured using the following definition.

Definition 3. *Consider arbitrary $1 \leq i \leq n$ and $\mu_1 \in \mathsf{ans}([Q]_{V_i}, [G]_{V_i})$. Assignment μ_1 is* redundant *for Q and i if, for each assignment $\mu \in \mathsf{ans}(Q, G)$ such that $\mu_1 = [\mu]_{V_i}$, there exist $1 \leq j \leq n$ and an assignment $\mu_2 \in \mathsf{ans}([Q]_{V_j}, [G]_{V_j})$ such that $\mu_2 = [\mu]_{V_j}$ and $\mathsf{val}_{\mu_1}([Q]_{V_i}) \subsetneq \mathsf{val}_{\mu_2}([Q]_{V_j})$. If such μ_1 is neither valid for Q nor redundant for Q and i, then μ_1 is a* partial match of Q in $[G]_{V_i}$.

As a simple consequence of Definition 3, note that the number of non-redundant answers in each partition element is at most equal to the number of non-local query answers. Theorem 1 captures the essence of our query answering scheme. Intuitively, it says that each answer μ to Q on G is obtained either by evaluating $[Q]_{V_i}$ on some partition element $[G]_{V_i}$ (i.e., it is a local answer), or by joining non-valid and non-redundant assignments $\mu_1, \ldots, \mu_n$ obtained by evaluating $[Q]_{V_i}$ on $[G]_{V_i}$ that instantiate all atoms of Q.

Theorem 1. *For a variable assignment μ, we have $\mu \in \mathsf{ans}(Q, G)$ if and only if*

1. *μ is valid for Q and $\mu \in \mathsf{ans}([Q]_{V_i}, [G]_{V_i})$ for some $1 \leq i \leq n$, or*
2. *variable assignments $\mu_1, \ldots, \mu_n$ exist such that*
   (a) *for each $1 \leq i \leq n$, either $\mathsf{dom}(\mu_i) = \mathsf{var}(Q)$ and $\mathsf{rng}(\mu_i) = \{*\}$, or we have $\mu_i \in \mathsf{ans}([Q]_{V_i}, [G]_{V_i})$ and μ_i is a partial match of Q in $[G]_{V_i}$,*
   (b) *for each $1 \leq j \leq m$, some $1 \leq k \leq n$ exists such that $j \in \mathsf{val}_{\mu_k}([Q]_{V_k})$, and*
   (c) *$\mu = \mu_1 \bowtie \ldots \bowtie \mu_n$.*

Proof. ($\Rightarrow$) Assume that $\mu \in \mathsf{ans}(Q, G)$. The claim holds trivially if μ satisfies (1), so assume that μ does not satisfy (1). For each $1 \leq i \leq n$, let $\xi_i = [\mu]_{V_i}$, and let μ_i be such that $\mathsf{dom}(\mu_i) = \mathsf{var}(Q)$ and $\mathsf{rng}(\mu_i) = \{*\}$ if ξ_i is redundant for Q and $\mu_i = \xi_i$ otherwise. We next show that each μ_i satisfies (2a)–(2c).

(2a) Consider an arbitrary $1 \leq i \leq n$. The claim holds trivially if ξ_i is redundant for Q and i, so we assume that $\mu_i = \xi_i$ is not redundant for Q and i. Since we assume that $\mu \notin \mathsf{ans}(Q, G)$, assignment ξ_i is not valid for Q. For each $1 \leq j \leq m$, since $\mu(A_j) \in G$, we clearly have $[\mu(A_j)]_{V_i} \in [G]_{V_i}$; furthermore, we have $[\mu(A_j)]_{V_i} = \mu_i([A_j]_{V_i})$, so $\mu_i([A_j]_{V_i}) \in [G]_{V_i}$ holds. Consequently, we have $\mu_i \in \mathsf{ans}([Q]_{V_i}, [G]_{V_i})$, as required.

(2b) Consider an arbitrary $1 \leq j \leq m$; then, $\mu \in \mathsf{ans}(Q, G)$ clearly implies $\mu(A_j) \in G$. Since $G \subseteq \bigcup_i [G]_{V_i}$, some $1 \leq i \leq n$ exists such that $\xi_i(A_j) \in [G]_{V_i}$, so clearly $j \in \mathsf{val}_{\xi_i}([Q]_{V_i})$. Now choose $1 \leq k \leq n$ such that $\mathsf{val}_{\mu_k}([Q]_{V_k})$ is a largest set satisfying $\mathsf{val}_{\xi_i}([Q]_{V_i}) \subseteq \mathsf{val}_{\xi_k}([Q]_{V_k})$. Since $\mathsf{val}_{\mu_k}([Q]_{V_k})$ is largest, such ξ_k is not redundant for Q and k, so $\mu_k = \xi_k$; but then, $j \in \mathsf{val}_{\mu_k}([Q]_{V_k})$, as required.

(2c) For each variable $x \in \mathsf{dom}(\mu)$, some $1 \leq i \leq n$ exists such that $\mu(x) \in V_i$; hence, it is obvious that $\mu = \xi_1 \bowtie \ldots \bowtie \xi_n$ holds. We next show that we can successively replace in this equation each ξ_i that is redundant for Q and i with μ_i. To this end, choose an arbitrary ξ_i that is redundant for Q and i, and choose an arbitrary $1 \leq j \leq n$ such that $\mathsf{val}_{\xi_i}([Q]_{V_i}) \subseteq \mathsf{val}_{\xi_j}([Q]_{V_j})$ and ξ_j is not redundant for Q and j; clearly, we have $[\xi_i]_{V_j} \bowtie \mu_j = \mu_j$. Now consider an arbitrary variable $x \in \mathsf{dom}(\xi_i)$ such that $\xi_i(x) \neq *$ and $\xi_i(x) \neq \xi_j(x)$. Since $\mathsf{val}_{\xi_i}([Q]_{V_i}) \subseteq \mathsf{val}_{\xi_j}([Q]_{V_j})$ holds, variable x occurs in Q only in atoms A_ℓ such

that $* \in \mathsf{voc}(\xi_i(A_\ell))$, so $\ell \notin \mathsf{val}_{\xi_i}([Q]_{V_i})$. But then, by (2b), some $1 \leq k \leq n$ exists such that $\xi_k(x) = \xi_i(x)$ and ξ_k is not redundant for Q and k. But then, $\mu = \xi_1 \bowtie \ldots \xi_{i-1} \bowtie \mu_i \bowtie \xi_{i+1} \bowtie \xi_n$ clearly holds. We can iteratively replace in this equation each ξ_i that is redundant for Q and i with μ_i without affecting the equality, as required.

($\Leftarrow$) Assume that (1) is true for some μ; then $\mu([A_j]_{V_i}) = \mu(A_j)$ for each $1 \leq j \leq m$, so clearly $\mu \in \mathsf{ans}(Q, G)$. Assume now that (2a)–(2c) are true, and consider an arbitrary $1 \leq j \leq m$. By (2a) and (2b), some $1 \leq i \leq n$ exists such that $\mu_i([A_j]_{V_i}) \in [G]_{V_i}$ and $* \notin \mathsf{voc}(\mu_i([A_j]_{V_i}))$. But then, $\mu_i([A_j]_{V_i}) \in G$; furthermore, by (2c), $\mu_i(x) = \mu(x)$ for each variable $x \in V_j$, so $\mu(A_j) \in G$. Consequently, $\mu \in \mathsf{ans}(Q, G)$. □

Hence, the answers to a query Q over $\mathbf{G}$ can be computed as follows. First, each i-th server computes $\mathsf{ans}([Q]_{V_i}, [G]_{V_i})$ in parallel, and it immediately returns all answers that are valid for Q. Second, the server identifies a subset $P_i \subseteq \mathsf{ans}([Q]_{V_i}, [G]_{V_i})$ of partial matches of Q in $[G]_{V_i}$, and it also extends P_i with assignment μ such that $\mathsf{dom}(\mu) = \mathsf{var}(Q)$ and $\mathsf{rng}(\mu) = \{*\}$. Third, all servers communicate P_i to one designated server, which then computes the join $P_1 \bowtie \ldots \bowtie P_n$ and returns each result that instantiates all atoms of Q. Our query answering scheme thus requires distributed computation only for answers spanning partition boundaries.

4.2 Identifying Redundant Answers

Checking whether some $\mu_1 \in \mathsf{ans}([Q]_{V_i}, [G]_{V_i})$ is redundant for Q and i requires one to consider each $\mu \in \mathsf{ans}(Q, G)$, which is clearly impractical. Thus, in this section we present an approximate redundancy check. Note that Theorem 1 holds even if some μ_i in Condition (2a) is redundant, so using an approximate check is safe from the correctness point of view.

Our optimisation is based on the fact that our data partitioning scheme ensures that answers to subject–subject joins are always local. Hence, if, for some μ, each 'star' in Q contains an atom that is not valid for μ, then μ is redundant. This is captured formally in Proposition 1; its proof is trivial, so we omit it for the sake of brevity.

Proposition 1. *Consider an arbitrary $1 \leq i \leq n$ and an arbitrary variable assignment $\mu \in \mathsf{ans}([Q]_{V_i}, [G]_{V_i})$. Then, μ is redundant for Q and i if, for each term $s \neq *$ occurring in $\mu([Q]_{V_i})$ in a subject position, an atom $A \in Q$ exists such that s occurs in the subject position of $\mu([A]_{V_i})$ and $* \in \mathsf{voc}(\mu([A]_{V_i}))$.*

4.3 Limitations of our Query Answering Strategy

Practical applicability of our approach depends critically on effective removal of redundant answers. As we show in Section 5, the optimisation from Proposition 1 is effective on some, but not on all queries. The latter is often the case for long chain queries (i.e., queries of the form $\langle x_0, R_1, x_1 \rangle \wedge \ldots \wedge \langle x_{n-1}, R_n, x_n \rangle$): in each

partition element, the wildcard resource typically has a large fan-out and fan-in, and it also occurs in triples of the form $\langle *, R_i, * \rangle$, which can give rise to a large number of answers that are not redundant.

As a possible remedy, we shall explore the possibility of adapting the approach presented in [21]. We envisage an algorithm that evaluates a query in each partition element using nested index loop joins; however, as soon as the algorithm matches some variable to $*$, the algorithm sends the variable matches identified thus far to other servers for continued evaluation. Such an algorithm would still produce all local answers locally and without breaking the query up into pieces, thus reaping the same benefits as the the approach we presented in this paper, but it would not explore any redundant answers. The main open question is to develop a suitable query planning algorithm.

5 Experimental Evaluation

In this section we experimentally evaluate our approach using the Lehigh University Benchmark (LUBM) and the SPARQL Performance Benchmark (SP2B). Each test dataset was split into a partition of size 20, and we used the queries available in the respective benchmarks. This size was chosen to make it directly comparable to related works such as [9, 11]. For a fixed dataset, increasing the partition size is likely to increase the number of non-local answers as the data becomes more fragmented; in contrast, fixing partition size while increasing the size of the dataset is likely to reduce the proportion of non-local answers. The extent to which these changes affect partition quality is out of the scope of this paper and we leave it for our future work. As we have already mentioned, we have not yet implemented a complete system that would allow us to measure end-to-end query answering times; hence, we only conducted the following experiments.

For each $\mathbf{G} = (G_1, \ldots, G_{20})$ of a test dataset G, we calculated (i) the percentage of local answers to test queries, (ii) the storage overhead—that is, the percentage $\frac{|G_1|+\ldots+|G_{20}|-|G|}{|G|}$, and (iii) the number of partial matches to test queries, according to Proposition 1. While experiment (i) determines how many non-local answers must be constructed, experiment (iii) provides us with an indication of how much work is required for this construction. This is critical because, in order to ensure the completeness of query answers, all partial matches in all partition elements must be computed and joined together.

We compared our approach with subject-based hash partitioning (written *Hash*) as in [8, 21], and semantic hash partitioning (written *SHAPE*) [11], which uses an optimised form of subject hashing and directed 2-hop duplication. We did not consider the graph partitioning approach by [9] because SHAPE was shown to offer superior performance. All of these partitioning approaches ensure that all answers to all star queries are local. Furthermore, Proposition 1 ensures there are no partial matches to star queries so we did not consider them in our tests. We used the RDFox system[1] to compute non-local answers, and so we use *RDFox* as the name of our approach.

[1] http://www.cs.ox.ac.uk/isg/tools/RDFox/

5.1 Test Datasets

The Lehigh University Benchmark (LUBM) [6] is a commonly used Semantic Web benchmark. It consists of a synthetic data generator for a simple university domain ontology, and 14 test queries, nine of which are star queries. The generator is parameterised by a number of universities, for which it creates data from the university domain. We used LUBM-2000, containing approximately 267 million triples. The main drawback of LUBM is that the data for each university is highly modular: entities in each university contain many more links amongst themselves than to entities in other universities. We used the five non-star benchmark queries and the following manually created circular query Qc:

```
SELECT DISTINCT ?w ?x ?y ?z WHERE {
    ?x ub:worksFor ?y . ?y ub:subOrganizationOf ?z .
    ?w ub:undergraduateDegreeFrom ?z . ?w ub:advisor ?x }
```

Some LUBM queries have non-empty results only if the data is extended according to the axioms from the LUBM ontology; however, since distributed reasoning is out of scope of this paper, we rewrote the test queries into unions of conjunctive queries in order to take the ontology axioms into account.

The SPARQL Performance Benchmark (SP2B) [16] is another synthetic benchmark that produces DBLP-like bibliographic data. We used an SP2B dataset with approximately 200 million triples. The benchmark provides 12 queries, of which we have used the five non-star queries for our comparison. Some of these queries contain OPTIONAL clauses, which we simply deleted because optional matches are currently not supported in our framework.

5.2 Partition Quality

Table 1 shows the percentage of local answers for each LUBM query. RDFox and SHAPE were able to answer all queries completely, which is in part due to the modular nature of the data; however, hashing performs poorly on all queries. Table 2 shows the results for SP2B. Again, hashing performs very poorly. Furthermore, both RDFox and SHAPE handled queries 4 and 6 well; however, RDFox significantly outperformed SHAPE on queries 5, 7, and 8.

One can intuitively understand these results as follows. Hashing by subject, although effective for star queries, performs very poorly for other types of query: in most cases, it provides almost no local answers. Thus, hashing is likely to be a poor data partitioning scheme for applications with diverse query loads. SHAPE considerably improves hashing, to the extent that only two benchmark queries are problematic. However, by partitioning the data based on its structure, one can further improve the overall performance: our approach is weakest on query Q5 from SP2B, but it still provides a high percentage of local answers.

Table 1: LUBM Percentage of Local Answers

System	Q2	Q8	Q9	Q11	Q12	Qc
RDFox	100.00%	100.00%	100.00%	100.00%	100.00%	100.00%
SHAPE	100.00%	100.00%	100.00%	100.00%	100.00%	100.00%
Hash	0.44%	4.96%	0.23%	5.80%	0.00%	0.04%

Table 2: SP2B Percentage of Local Answers

System	Q4	Q5	Q6	Q7	Q8
RDFox	95.95%	73.00%	99.90%	92.41%	91.45%
SHAPE	95.23%	9.72%	100.00%	41.97%	73.72%
Hash	0.01%	0.77%	0.25%	0.08%	0.26%

Table 3: Storage Overhead

	RDFox	SHAPE	Hash
LUBM	3.60%	84.23%	0.00%
SP2B	0.60%	38.63%	0.00%

5.3 Storage Overhead

As we have already discussed, the percentage of local answers can be increased using n-hop duplication, but at the expense of storage overhead. For example, with 2-hop duplication, the approach by [9] can incur an overhead up to 430%.

Table 3 shows the overhead for all partitioning schemes and data sets we considered in our experiments. Hashing clearly incurs no overhead; moreover, although SHAPE incurs a considerable overhead, that can be acceptable for some applications. Our partitioning scheme, however, exhibits a negligible overhead. Intuitively, this is due to the fact that min-cut graph partitioning tries to minimise the number of cut edges, which leads to a small level of duplication.

5.4 Query Evaluation

We evaluated each test query on each partition element, and we discarded all valid assignments and some redundant assignments (according to Proposition 1). For each query, we computed the mean, minimum, maximum, and the sum of the numbers of partial matches across all partition elements.

On LUBM, queries 2, 8, 11 and 12 had no partial matches, so they can be evaluated fully locally without the need for any distributed processing. Queries 9 and c had 6 and 11, respectively, partial matches in total, so the necessary distributed processing is negligible.

On SP2B, evaluating queries 4 and 8 on all partition elements did not finish within an hour, producing very large numbers of partial matches. Since the number of partial matches in each partition element is bounded by the number of non-local answers and the latter is small (cf. Table 2), this result shows that Proposition 1 is not very efficient in identifying redundant answers for queries 4 and 8. Table 4 summarises the results for the remaining queries; in order to better understand these numbers, the table also shows the numbers of total and non-local answers. For queries 5 and 7, the numbers of partial matches are much smaller than the numbers of non-local answers, suggesting that joining the partial matches should be practically feasible. In contrast, the number of partial

Table 4: Partial Matches for SP2B

Query	Total Answers	Non-local Answers	Partial Matches			
			Mean	Min	Max	Total
Q5	2,970,234	801,958	19,128	1,847	43,755	382,564
Q6	38,111,881	38,048	819,709	117,781	1,321,781	16,394,172
Q7	184,193	13,989	2,874	642	6,528	57,471

matches to query 6 is orders of magnitude larger than the number of non-local answers, suggesting that joining the partial answers might be difficult.

To summarise, our approach produces no or few partial matches on many types of query, but it runs into problems with long chain queries such as SP2B query 8. We shall try to improve on this using the ideas outlined in Section 4.3.

6 Conclusion

We have presented a new scheme for partitioning RDF data across a cluster of shared-nothing servers. Our main goal is to minimise the number of connections between partition elements so as to ensure that most answers to typical queries are local (i.e., they can be obtained by evaluating the query locally in all partition elements). We encode in each partition element links to other partition elements, and we use this information in a novel query answering scheme to correctly compute all answers to queries. Unlike existing systems, our query answering scheme retrieves all local answers by simply evaluating the query in each partition element, and it uses the encoded links to reduce the need for distributed processing. We have shown that, on the LUBM and SP2B benchmarks, test queries have more local answers under our data partitioning scheme than with subject-based hashing or semantic partitioning [11], and that our data partitioning scheme incurs a negligible storage overhead. Finally, we have shown that our query answering scheme is effective on many, but not all test queries.

We see two main challenges for our future work. First, we shall try to adapt the graph exploration technique by [21] to obtain a more robust query answering scheme. Second, we shall extend the RDFox system to a fully fledged distributed RDF data store and compare it with existing systems.

Acknowledgements Our work was supported by a doctoral grant by Roke Manor Research Ltd, an EPSRC doctoral training grant, and the EPSRC project MaSI[3].

References

1. Abadi, D.J., Marcus, A., Madden, S.R., Hollenbach, K.: SW-Store: A Vertically Partitioned DBMS for Semantic Web Data Management. The VLDB Journal (2009)

2. Carroll, J.J., Dickinson, I., Dollin, C., Reynolds, D., Seaborne, A., Wilkinson, K.: Jena: implementing the semantic web recommendations. In: WWW (Alternate Track Papers & Posters) (2004)
3. Dean, J., Ghemawat, S.: MapReduce: Simplified Data Processing on Large Clusters. Commun. ACM (2008)
4. Erling, O., Mikhailov, I.: Virtuoso: RDF Support in a Native RDBMS. In: Semantic Web Information Management. Springer Berlin Heidelberg (2010)
5. Giménez-García, J.M., Fernández, J.D., Martínez-Prieto, M.A.: MapReduce-based Solutions for Scalable SPARQL Querying. Open Journal of Semantic Web (OJSW) (2014)
6. Guo, Y., Pan, Z., Heflin, J.: LUBM: A Benchmark for OWL Knowledge Base Systems. Web Semantics (2005)
7. Harris, S., Lamb, N., Shadbol, N.: 4store: The Design and Implementation of a Clustered RDF Store. In: Proc. of the 5th Int. Workshop on Scalable Semantic Web Knowledge Base Systems (SSWS2009). Washington DC, USA (October 26 2009)
8. Harth, A., Umbrich, J., Hogan, A., Decker, S.: YARS2: A Federated Repository for Querying Graph Structured Data from the Web. In: Aberer, K., Choi, K.S., Noy, N.F., Allemang, D., Lee, K.I., Nixon, L.J.B., Golbeck, J., Mika, P., Maynard, D., Mizoguchi, R., Schreiber, G., Cudré-Mauroux, P. (eds.) Proc. of the 6th Int. Semantic Web Conf. (ISWC 2007). LNCS, vol. 4825, pp. 211–224. Busan, Korea (November 11–15 2007)
9. Huang, J., Abadi, D.J., Ren, K.: Scalable SPARQL Querying of Large RDF Graphs. In: Proceedings of the VLDB Endowment (2011)
10. Karypis, G., Kumar, V.: A fast and high quality multilevel scheme for partitioning irregular graphs. SIAM Journal on Scientific Computing (1999)
11. Lee, K., Liu, L.: Scaling Queries over Big RDF Graphs with Semantic Hash Partitioning. In: Proceedings of the VLDB Endowment (2013)
12. Neumann, T., Weikum, G.: RDF-3X: a RISC-style Engine for RDF. In: Proceedings of the VLDB Endowment (2008)
13. Owens, A., Seaborne, A., Gibbins, N., Schraefel, M.C.: Clustered TDB: A Clustered Triple Store for Jena. http://eprints.ecs.soton.ac.uk/16974/
14. Papailiou, N., Konstantinou, I., Tsoumakos, D., Koziris, N.: H2RDF: adaptive query processing on RDF data in the cloud. In: WWW (2012)
15. Rohloff, K., Schantz, R.E.: High-performance, Massively Scalable Distributed Systems Using the MapReduce Software Framework: The SHARD Triple-store. In: Programming Support Innovations for Emerging Distributed Applications (2010)
16. Schmidt, M., Hornung, T., Lausen, G., Pinkel, C.: SP2Bench: A SPARQL Performance Benchmark. CoRR (2008)
17. Shao, B., Wang, H., Li, Y.: The Trinity graph engine. Tech. rep., Microsoft Research (2012)
18. Sun, J., Jin, Q.: Scalable RDF Store Based on HBase and MapReduce. In: International Conference on Advanced Computer Theory and Engineering (2010)
19. Urbani, J., Kotoulas, S., Maassen, J., van Harmelen, F., Bal, H.E.: WebPIE: A Web-scale Parallel Inference Engine using MapReduce. Journal of Web Semantics 10, 59–75 (2012)
20. Weiss, C., Karras, P., Bernstein, A.: Hexastore: Sextuple Indexing for Semantic Web Data Management. In: Proceedings of the VLDB Endowment (2008)
21. Zeng, Z., Yang, J., Wang, H., Shao, B., Wang, Z.: A Distributed Graph Engine for Web Scale RDF Data. In: Proceedings of the VLDB Endowment (2013)

A Distributed Query Execution Method for RDF Storage Managers

Kiyoshi Nitta[1], Iztok Savnik[2,3]

[1] Yahoo JAPAN Research, Tokyo, Japan
[2] University of Primorska &
[3] Jozef Stefan Institute, Slovenia

knitta@yahoo-corp.jp, iztok.savnik@upr.si

Abstract. A distributed query execution method for Resource Description Framework (RDF) storage managers is proposed. Method is intended for use with an RDF storage manager called *big3store* to enable it to perform efficient query execution over large-scale RDF data sets. The storage manager converts SPARQL queries into tree structures using RDF algebra formalism. The nodes of those tree structures are represented by independent processes that execute the query autonomously and in a highly parallel manner by sending asynchronous messages to each other. The proposed data and query distribution method decreases the amount of inter-server messages during query executions by use of the semantic properties of RDF data sets.

1 Introduction

There is a growing interest to gather, store, and query data from various aspects of human knowledge. Such data includes geographical data; data about various aspects of human activities such as music, literature, and sport; scientific data from biology, chemistry, astronomy, and other scientific fields; and data related to the activities of governments and other influential institutions.

There is a consensus among Semantic Web researchers that data should be presented in some form of *graph data model* in which simple and natural abstractions are used to represent data as *subjects* and their *properties* described by *objects* — that is, by means of the nodes and edges of a graph. Considering this from the point of view of knowledge developed in the fields of data modeling and knowledge representation, all existing data models and languages for the representation of knowledge can be transformed, in many cases quite naturally, into some incarnation of a *graph*.

A number of practical projects that allow for the gathering and storing of graph data already exist. One of the most famous examples is the Linked Open Data (LOD) project, which gathered more than 32 giga triples from areas including the media, geography, government, life sciences and others. In that project, the Resource Description Framework (RDF), which is a form of graph data model, was used to represent the data.

Storing and querying such huge amounts of structured data has created a problematic scenario that could be compared to the problem of querying huge amounts of text

that appeared after the advent of the Internet. The differences are in the degree of structure and semantics that data formats such as RDF and OWL encompass compared to HTML. HTML data published on the Internet represents a huge hypergraph of documents interconnected with links. Links between documents do not carry any specific semantics except those representing URIs.

In contrast to HTML, RDF is a *data model* in which all data is represented by triples (subject, predicate, object). In this format, we can represent entities and their properties similarly to object-oriented models or AI frames. Moreover, we can represent objects at different levels of abstraction: RDF can model not only ordinary data and data modeling schemata but also meta-data.

The primary modeling principle of RDF is the assignment of special meaning to properties with selected names. In this way, we can define the exact meaning of properties that are commonly used to describe documents, persons, relationships, and others. *Vocabularies* are employed to standardize the meaning of properties. The Dublin Core [6] project is an example of defining a set of common properties of things. The XML-schema [23] vocabulary defines the properties that can specify types of objects, and the vocabularies of properties and things are used to define higher-level data models realized on top of RDF. The RDF Schema [17] and the OWL [16] are two examples of providing object-oriented data modeling facilities and constructs for the representation of logic.

The contributions of this paper are as follows. Firstly, we propose an architecture of RDF query processor that gives rise to novel *distributed query execution method* for RDF storage managers. While the architecture is rooted in relational database technologies, we propose flexible and highly parallel solution allowing allocation and execution of very large number of queries expressed as data-flow programs on cluster of servers. Secondly, we propose the use of *semantic distribution* of triples that distributes data based on relationships of triples to the conceptual schema. Semantic distribution provides very general means for partitioning triple-store into non-overlapping portions, and, allows efficient distribution of query processing.

The paper is organized as follows. Section 2 presents architecture of storage system big3store. This section introduces data distribution method and main building blocks of storage system such as front servers and data servers. Conceptual design of query execution is presented in Section 3. Query execution engine is based on efficient parallelisation of query trees. Related work is described in Section 4. Finally, concluding remarks are given in Section 5.

2 Architecture of big3store

To provide fast access to big RDF databases and to allow a heavy workload, a storage manager has to provide facilities for flexible distribution and replication of RDF data. To this end, the storage manager has to be re-configurable to allow many servers to work together in a cluster and to allow for different configurations of clusters to be used when executing different queries.

The storage manager for big RDF databases should be based on SPARQL and on the algebra of RDF graphs [19]. To provide a more general and durable storage manager,

its design should be based on the concept of graph databases [2]. Such design would enable adding interfaces for popular graph data models other than RDF to be added later.

2.1 Storage manager as cluster of data servers

Possible distribution and replication is crucial for the design of the storage manager in order for it to be available globally and to enable heavy workload that is expected if LOD data is going to be used by the masses.

Heavy distribution and replication is currently possible because of the availability of inexpensive commodity hardware for servers with huge RAM (1–100 GB) and relatively large disks. The same concept was used by Google while bootstrapping and remains the main design direction of Google data centers [10].

As we discuss in more detail later, a cluster of data servers can easily be configured into a very fast data-flow machine answering a particular SPARQL query. A similar idea has recently appeared in the area of super-computers [9], where advances in hardware technologies now allow compiler preprocessors to configure hardware facilities for a specific program. The program then runs on specially configured hardware that gains considerable speed.

The leading idea for the distribution of SPARQL query processing is splitting a SPARQL query into parts that are executed on different data servers, thus minimizing the processing time. Data servers executing parts of the SPARQL query are connected by streams of data to form a cluster configuration defined for a particular SPARQL query. Similar to the way in which some super-computers are based on configuring intelligent hardware, we also have a strict separation between two phases: 1) compiling the program into a hardware configuration and 2) executing the program on the selected hardware configuration.

Figure 1 shows a cluster composed of two types of servers: *front servers* represented as the nodes of plane A, and *data servers* represented as the nodes of plane B. Data servers are configured in *columns* labeled from (a) to (f). A complete database is distributed to columns. with each column storing a portion of the complete database. The methods for distributing the RDF data are discussed in the following sections.

The portion of the database stored in a column is replicated into rows labeled from 1 to 5. The number of rows for a particular column is determined dynamically based on the query workload for each particular column. The heavier the load on a given column, the greater the number of row data servers chosen for replication. The particular row used for executing a query is selected dynamically based on the current load of servers in a column.

A particular cluster configuration for answering a particular SPARQL query is programmed by front servers. This is also where the optimization of the SPARQL query takes place. The front server receives a SPARQL query, parses it to the query tree, and performs optimization based on the algebraic properties of the SPARQL set algebra operations. Parts of the query tree are sent to internal data servers to define the cluster configuration used for a particular query execution.

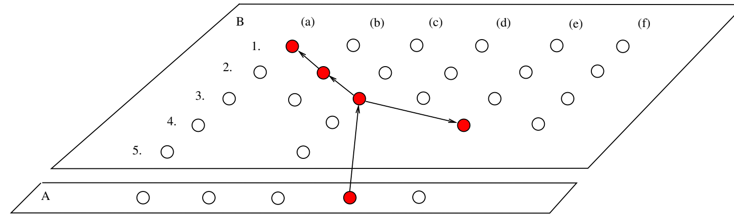


Fig. 1. Configuration of servers for a particular query

2.2 Data distribution

RDF data stored in a data center is distributed to *columns* of data servers that form the cluster. Each data server includes a triple store accessible through TCP/IP. Each column is composed of an array of data servers referred to as *rows* that are replicas storing the identical portion of the big3store database.

Distribution of RDF data to columns can be defined in more ways. First, data can be split manually by assigning larger data sets (databases) to columns. One example of such a dataset is DBpedia [4]. Second, RDF data can be split into columns automatically by using SPARQL queries as the means to determine groups of RDF triples that are likely to be accessed by one query. In this context, RDFS classes can be employed as the main subject of distribution, as suggested in [18]. Groups of classes that are usually accessed together can be assigned to *columns* where similar class instances are stored.

The benefits of splitting a triple store into separate data stores (tables) have been shown in [24]. Basically, queries can be executed a few times faster. The reason for this can only be the size and height of indexes defined for tables representing triples. This means that fewer blocks have to be read from a database if RDF data is distributed to different tables.

There are two scenarios in which the automatic reconfiguration of an RDF database can be implemented. First, a complete database may be automatically distributed into columns, as described above. Second, the degree of replication of the portion of the database stored in a column needs to be determined. In other words, we have to determine how many rows (replicas) are needed to process queries targeting a particular column efficiently.

2.3 Front servers

Front servers are servers where SPARQL queries initiated by remote users are accepted, parsed, optimized, and then distributed to data servers.

A SPARQL parser checks the syntax of a query and returns a diagnosis to the user as well as prepares the query tree for the optimization phase. The most convenient approach to optimizing a SPARQL query is to transform queries into algebra and then use the algebraic properties for optimization. The algebra of RDF graphs [19] designed for big3store is based on the work of Angles and Gutierrez [1] and of Schmidt et al. [20].

The *algebra of RDF graphs* reflects the nature of the RDF graph data model. While it is defined on sets, the arguments of algebraic operation and its result are RDF graphs. Furthermore, the expressions of RDF graph algebra are graphs themselves. Triple patterns represent the leaves of expressions. Graph patterns are expressions that stand for graphs with variables in place of nodes and edges.

In order to ship the partial results of a distributed query tree among data servers, the algebra of RDF graphs uses operation `copy`, first introduced in [5]. Operation `copy` can be well integrated with operations defined on graphs due to the simple set of algebraic rules that can be used for `copy`.

A query optimizer rooted in relational rule-based query optimization has been proposed by Savnik in [18] for handling RDF queries. Similarly to our approach Schmidt and Lausen [20] also use relational rules for the optimization of SPARQL queries. Optimization in big3store is based on a variant of dynamic programming algorithm for optimizing the algebraic expressions called *memoisation*. Since the search space grows exponentially with the number of the rules, we experiment with beam search selecting only the most promising transformations. Query cost estimation, that is vital for guiding beam search, is also rooted in cost estimation of relational database management systems.

The result of query optimization for a given SPARQL query is a query tree where operations `copy` are placed optimally representing the points where triples are shipped from one data server to another. The global query is therefore split into parts that are executed on different data servers. Initially, the front server sends a query to a data server from a column that includes data needed to process the top level of the query tree. Note that all query parts are already in optimized form.

2.4 Data servers with local triple store

In this section, we present the main features of a distributed query evaluation. We first give an overview of the distributed query evaluation and then present some of the properties of the local triple store and the evaluation of queries within it.

Evaluation of distributed query The primary job of a data server is to evaluate the query tree received from either the front server or some other data server. The query tree includes detailed information about access paths and methods for the implementation of joins used for processing the query. We refer to such a query tree as an *annotated query tree*. The data server evaluates the annotated query tree as it is without further optimization.

The triple store of the data server accepts queries via TCP/IP and returns the results to the return address of the calling server. The communication between the calling server and a given data server is realized by means of streams of triples representing the results of the query tree evaluation. When needed, the materialization of stream results is handled by the calling server.

The query tree can include parts that have to be executed on some other data servers if data needed for a particular query part is located at some other columns. Such query parts are represented by query sub-trees with root nodes that denote operation `copy`.

Again, query sub-trees can include additional instances of operation `copy` so that the resulting structure of data servers constructed for a particular SPARQL query can form a tree.

Since operation `copy` is implemented by using a stream of triples, the query parts that form a complete query tree can be executed in parallel. While the data server is processing the query sub-tree that computes the next triple to be consumed by a given data server, it can also process previously read triples and/or perform other tasks such as accessing local triples. Moreover, big3store can process many query parts in parallel as a parallel data-flow machine.

Local evaluation of queries Let us now present the evaluation of queries on the local data server. Assume the data server receives an annotated query tree `qt`. Recall that `qt` includes information about access paths to the tables of triples and algorithms to be used for implementing algebra operations.

The local triple store includes the implementations of algebra operations and of access paths, i.e., methods for accessing indexed tables of triples. Algebraic operations include selection with or without the use of index; projection; set operations for union, intersection, and difference; and variants of nested-loop join with or without index, where the index supports either equality joins or range queries.

A non-distributed storage manager for storing triples and indexes for accessing triples has to deal with similar problems to those faced by relational storage managers. We use a local database management system called *Mnesia*, which is a part of Erlang programming language [3] distribution, to store and manage tables of triples, referred to as *triple-stores*. Triple-store of big3store is a table including four attributes: triple id, subject, property and object. Adding triple ids to triple-store is the decision that we expect will allow more consistent and uniform storage of various data related to triples, such as, named graphs and other groupings of triples, properties of triples (reification), and the like. Each triple-store maintains 6 indexes for accessing SPO attributes and additional index for triple ids.

We tend to use low levels of Mnesia storage manager including access to tables and indexes since optimization is performed by global query optimizer of big3store. Furthermore, lower levels of relational storage manager can be easily replaced by some other storage manager or even with file-based storage system. We relate this level of storage manager to data storage facilities of Hadoop [22]. Maps, for instance, represent main indexing mechanism of Mnesia while they are well comparable to Hadoop Maps. In this way, we achieve the simplicity of lower parts of storage manager in comparison to the complexity of RDBMs—this may represent a trend started with Hadoop. We can compare our work on compiling and executing high-level data-flow programs with programs and scripts written using Hadoop. Finally, for practical reasons, some features of Mnesia will be used for implementation of caching in big3store. Data about user context, including the results of all his queries, can be easily stored in Mnesia RAM tables increasing in this way significantly the speed of query evaluation.

Let us now give some more details about implementation of operation `copy`. As stated briefly before, operation `copy` implements a stream between two data servers. This stream is realized by first initiating the execution of a sub-tree of `copy` (i.e. a

query part) and then requesting that the results be sent back to the calling data server by means of a stream. On the caller side, access to the stream, i.e., the results of operation `copy`, is realized as an access method that reads triples from the stream.

2.5 Distribution of Triples and Triple Patterns

The main idea of *semantic distribution* is to distribute database triples on the basis of database schema. In other words, the distribution of triples to data servers (columns) is based on the relationship of triples to triple-base schema.

We suppose that we have triple-base such that complete schema for ground triples is defined and stored in database. An example of such RDF datasets is YAGO [13] which includes classes of all subjects, schema for all properties as well as taxonomy of classes.

In the sequel we will first present semantic distribution function in general. Class-based and property-based semantic distribution functions will be described in more detail. Some properties and trade-offs of semantic distribution function will be given.

Distribution function The distribution of triples can be achieved by means of a *distribution function* dist(), which maps a triple or a triple pattern into a set of column identifiers. Each column identifier represents a portion of the complete triple-base. The sizes of distributed portions must be similar.

The proposed method for semantic distribution is general since it allows various subsets of $\{S, P, O\}$ to be used as the means for distribution.

For instance, distribution can be defined on S part of the schema: instances of S's type c are stored in a column assigned to type c. Similarly, triples can also be stored on the basis of P part. In this case, each property has a column where its instances are stored. Furthermore, if we would like to separate properties defined for particular classes, distribution might be defined on the S and P parts of the triple schema.

Let us now consider two types of semantic distribution in more detail. Firstly, we present *class-based* semantic distribution which uses S part of triples, and secondly, we consider *property-based* semantic distribution which uses P part of triples.

Class-based distribution The first possibility of triple distributions is to distribute triples in partitions on the basis of the S part of triple schema, i.e., based on RDF class. A triple belongs to a RDF class if it describes the property of an instance of that class. The RDF class of the instance is determined by means of the $rdf{:}type$ property. We assume that all instances of classes have an $rdf{:}type$ relationship defined in a given triple store.

What are the effects of triple distribution based on classes in terms of query evaluation? In the case of queries related to the properties of two classes, the queries are evaluated on two different servers. In the case of queries tackling the properties of instances of three RDF classes, they are evaluated on three servers, etc.

In the case in which spreading the evaluation of queries to more data servers is desired, the properties pertaining to a particular class must not be stored in one column but rather must be distributed into additional columns.

Continuing to distribute the properties of RDF classes to different columns would in the end result in a distribution function based on the P-part of triples and not on RDF classes. This would make sense if we could determine experimentally that it is more efficient to distribute a query tree to data servers such that one query node is executed on one query node.

Property-based distribution The second type of semantic distribution can be defined on the basis of P part of triples, i.e., based on RDF properties. Since each ground triple includes the property part same as does the schema triple denoting given ground triple, it is easier to define the distribution based on properties. Columns are assigned to each property so that database triples are distributed uniformly to data servers.

In this case the distribution function can be simply defined by extracting P part of triple or triple pattern, and, then, mapping property to columns using predefined table. However, a problem appears in the case P part of triple pattern includes a variable. The only possible way to query properties when using distribution based on properties is by sending “broadcast” query to data servers of all columns.

Trade-offs of distribution function It remains to be determined experimentally what kind of distribution function behaves optimally for a given triple-base and query workload.

The first aspect of query evaluation that needs to be considered is weather it is better to store data and evaluate all query nodes related to a given class c on one data server or, is it better to split data and query nodes related to c to separate data servers (i.e., columns).

Another variable of query evaluation, that is not directly related to semantic distribution, is to determine how many query nodes should be assigned to one data server in average to give optimal performances.

In one extreme, complete query is executed on one data server and the other extreme is that each query node is executed on a separate data server, each of which is connected by streams. The optimal distribution of queries to an array of data servers may require assigning query nodes to data servers such that each data server executes an average of two to three query nodes.

The patterns in mapping nodes of query trees to data servers that achieve fast execution of query trees on a given triple-base need to be excavated through experiments. The patterns can be used as a target structure of query optimization. One way to do that is to use patterns for tuning parameters of query optimization such as cost of moving intermediate results of queries among data servers. Another way to use patterns in query optimization is to restrict search space by focusing search to queries that match excavated patterns.

3 Conceptual Design of Query Execution

The query execution module (QEM) of the big3store system takes query tree structures as inputs and produces streams of result messages as outputs. When a SPARQL query

is requested to be execute on the big3store system, the query is translated into a graph structure called a *query tree* (an example is shown in Figure 3). The query tree can be modified by the query optimizer module of big3store to make the structure better for efficient execution. While query trees are represented in some data structures in big3store, QEM takes query trees represented by processes. These processes are dynamically generated by the preparation procedure of the big3store system. Query trees are executed by those processes cooperating with other big3store processes. Therefore, QEM is not a single process but rather consists of several processes including query tree processes to be executed. These processes can be distributed to multiple physical server machines that are connected by an ordinary network. A process is identified by a combination of server id and process id.

While most parts of the big3store system have not been developed, the prototype of QEM was developed and tested using small example data on a single server (non-distributed) environment. The reason for developing QEM first is that the investigation of query execution efficiency seems to be the most important challenge. The next step of our research will be to experiment the QEM execution in distributed environments.

In this section, we introduce processes for performing such query executions and then give an example of the entire flow of a query execution.

3.1 Query tree

Process `query_tree` accepts requests for managing query nodes. A physical server machine has only one `query_tree` process in big3store. It manages active query node processes, which are described in the next subsection; accepts requests for creating and deleting query nodes on any server; and provides unique process ids in the server on which the `query_tree` is running for creating query nodes on the server.

3.2 Query node

Process `query_node` implements a query node that constructs a query tree with other query nodes to represent a query. A `query_node` process can run on any physical server machine on which a `query_tree` process is running. Each `query_node` process has its own hash table on which to store multiple property values. It provides put and get interfaces for accessing property values. While various types of relations between query nodes can be represented by properties, *parent* relations are used for representing the stem structures of query trees. Each query tree has a *root* query node that has no value for a parent property. Each query node excepting the root query node must have a parent property for representing its parent query node by a combination of server id and process id. A `query_node` process must have a type property that indicates an operation type of RDF algebra [19]. Currently, *triple pattern* and *join* types are implemented.

Triple pattern query node A triple pattern `query_node` process represents a matching condition for triples. It must have a *triple_pattern* property for representing a triple pattern that consists of IRIs, literals, or variables in subject, predicate, and object slots.

It can handle *list_vars*, *eval*, and *result* asynchronous messages. If it receives a *list_vars* message, it sends a list of variables contained in the triple pattern to its parent by a *construct_vars* asynchronous message. It also sets a *vars* property for reminding the list of variables. If it receives an *eval* message, it sends a *find_stream* asynchronous message to data server processes for invoking streamer processes that repeatedly send *result* messages to the triple pattern process, each of which contains a concrete triple matching the triple pattern. If it receives a *result* message, it sends the message to its parent query node.

Join query node A join `query_node` process represents a conjunctive condition of two query nodes. It must have *nodInner* and *nodOuter* properties for representing the target query nodes of the inner and outer edges, respectively. It can handle *eval*, *construct_vars*, and *result* asynchronous messages.

If it receives an *eval* message, it sends *list_vars* messages to its inner and outer query nodes for listing all variables that appear in the sub tree. At the same time, it sends an *eval* message to the outer query node. Granted query tree structures are left-deep style and only permit outer edges to have join query nodes (Figure 3). Therefore, this eval-propagation strategy successfully constructs a variable list for any granted query tree. If a join query node process receives *construct_vars* messages from inner and outer query nodes, it merges both variable lists and sends the merged list to its parent query node by another *construct_vars* message.

If it receives a *result* message from its outer query node, it sends synchronous messages to data servers for inquiring whether the triple set in the result message satisfies the join condition or not. A result message consists of an *alpha map* and a *val map*, the structures of which are shown in Tables 1 and 2, respectively. The alpha map associates the set of triples and their origin query nodes, each of which has a matching triple pattern, and the val map represents variable bindings that were determined by the set of triples. When the query node asks the data servers to process the result message, the node fetches a triple pattern from its inner query node, substitutes the val map variable bindings in the inner triple pattern, and sends the substituted triple pattern to the data servers. If the data servers find no matching triple, the node does nothing. Otherwise, the node generates new result messages for each matched triple and sends them to a parent query node asynchronously. The new result messages are made by a new alpha map and a new val map. The alpha map is added by an element that maps the found triple with the inner query node and the val map is added by variable bindings determined by the found triple.

Table 1. Alpha map structure

No.	Field name	Description
1	triple	triple id in the triple table
2	query_node	process id of corresponding query node

Table 2. Val map structure

No.	Field name	Description
1	variable	string_id coded id of the variable string
2	value	string_id coded id of the IRI or literal value

3.3 Other processes

There are several other processes that are necessary to execute QEM successfully.

Data server Each process of the `data_server` holds a chunk of triple data, and multiple data server processes are invoked in running the b3s server of a distributed configuration. A data server process dispatches storing and retrieving requests of held triple data. It also accepts requests for producing streams of data that match specific triple patterns. It uses a Mnesia table for storing triple data.

Streamer A data server streamer process is invoked by a data server process when it receives a request for generating streams for a given triple pattern. The streamer retrieves matching triples from the data server's triple table. After sending all stream data, the streamer closes the stream and terminates itself.

Map between string and id Process `string_id` maintains a mapping table and dispatches access operations for the table.

Translating all IRIs and literals into integer ids makes the triple tables smaller. However, the cost of processing the translation might create a bottleneck affecting the execution efficiency of distributed systems. One solution is to run multiple `string_id` processes in different servers, but this would also increase the number of translate operations for identifying the same strings between different `string_id` processes.

3.4 An Example

In this subsection, we describe a message stream flow using the example query shown in Figure 2 in order to explain the conceptual design of the big3store query execution mechanism.

```
SELECT * WHERE {
  ?c <hasArea>       ?a .
  ?c <hasLatitude>   ?l .
  ?c <hasInfration> ?i
}
```

Fig. 2. SPARQL query of **q01a**.

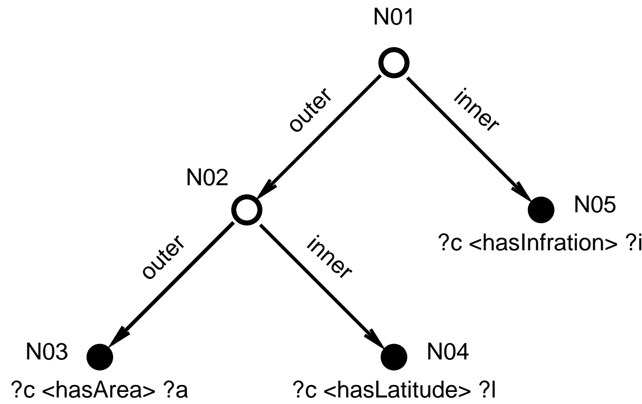


Fig. 3. Query tree structure of query **q01a**.

A query tree can be executed by sending an *{eval, Root, Parent}* asynchronous message to a `query_tree` process. The argument *Root* shows the *root* query node of a query tree to be executed. The `query_tree` process sends an *eval* message to the root node. The argument *Parent* shows a process that receives the result of the query. Figure 3 depicts the query tree structure of the example SPARQL query shown in Figure 2, where the black and white circles show the triple pattern and join query nodes, respectively. Query nodes are numbered N01, N02, ... in this figure for convenience. Query node N01 is a root join query node that is connected to query nodes N02 and N05 by outer and inner edges, respectively. Query node N02 is a join query node that is connected to query nodes N03 and N04 by outer and inner edges, respectively. Query nodes N03, N04, and N05 are triple pattern query nodes. Because query node N01 is the root of the query tree, it receives an *eval* asynchronous message for starting the execution of the query tree.

After receiving the initial *eval* message, the join query nodes propagate *eval* messages following outer edges. Figure 4 shows the messages sent by the processes for executing the query tree of Figure 3. As in Figure 3, black and white circles are used for representing the triple pattern and join query nodes. Each circle also represents an independent process in this figure. Two gray circles are added to represent a data server and a streamer processes. Edges drawn in solid lines show asynchronous messages and edges drawn in dashed lines show synchronous messages or function calls. After query node N01 receives an *eval* message, N01 sends another *eval* message to its outer query node N02. Query node N02 then performs the same action to N03. Because N03 is a triple pattern query node, it sends a *{find_stream, TriplePattern, QueryNode}* asynchronous messages to the data server process. The argument *TriplePattern* is the triple pattern that was set to N03. The argument *QueryNode* is used for specifying the caller of the message. It is N03 in this example. When the data server process receives the *find_stream* message, it invokes a new streamer process on the same physical server machine with

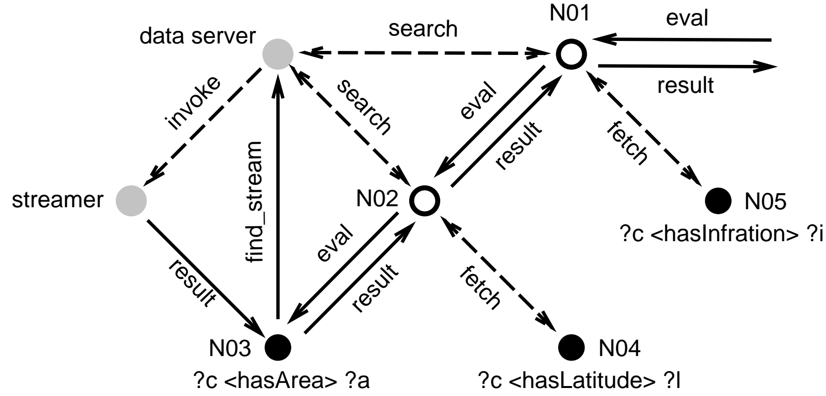


Fig. 4. Message stream flow of the execution of **q01a**.

the data server. The streamer process repeatedly sends *result* asynchronous messages to N03, each of which has a triple that matches the triple pattern of N03.

Each triple is represented in an *alpha map* entry in a corresponding result message. When query node N03 receives a *result* message, N03 checks a triple in the message for the selection condition. If the triple satisfies the condition, query node N03 sends the *result* message to its parent query node N02. When query node N02 receives a *result* message, N02 substitutes the triple pattern of N04 with *val map* variable bindings in the message. N02 then sends a synchronous *search* message to the data server process for retrieving triples that match the substituted N04 triple pattern. When query node N02 retrieves a matching triple from the data server, N02 modifies alpha and val maps in the received message and sends the new *result* message to its parent query node N01. Note that plural triples from the data server produce plural *result* messages from query node N02. Query node N01 performs actions similar to N02 for retrieving triples that match the triple pattern of N05. The answers for the query of Figure 2 are sent from query node N01 as a stream of *result* messages, each of which includes a set of triples that is a concrete solution of the query.

4 Related Work

In this section, we present some of the more relevant systems for querying RDF data, including 3store, 4store, Virtuoso, and Hexastore. See the survey presented in [14] for a more complete overview of RDF storage managers.

3store 3store [11] was originally used for Semantic Web applications, particularly for storing the hyphen.info RDF dataset describing computer science research in the UK. The final version of the database consisted of 5,000 classes and about 20 million triples. 3store was implemented on top of a MySQL database management system and included simple inferential capabilities (e.g., class, sub-class, and sub-property queries) mainly

implemented by means of MySQL queries. Hashing was used to translate URIs into the internal form of representation.

The query engine of 3store used RDQL query language originally defined in the frame of the Jena project. RDQL triple expressions are first translated into relational calculus and constraints are added to relational calculus expressions that are then translated into SQL. Inference is implemented by a combination of forward and backward chaining that computes the consequences of asserted data.

4store 4store [12] was designed and implemented to support a range of novel applications emerging from the Semantic Web. RDF databases were constructed from Web pages including people-centric information, resulting in ontology with billions of RDF triples. The requirements were to store and manage 15×10^9 triples.

4store is designed to operate on clusters of low-cost servers. It is implemented in ANSI C. It was estimated that the complete index for accessing quads would require around 100 GB of RAM, which is why data was distributed to a cluster of 64-bit multicore x86 Linux servers, each storing a portion of RDF data. The architecture of the cluster is a "Shared Nothing" type. Cluster nodes are divided into processing and storage nodes. Data segments stored on different nodes are determined by a simple formula that calculates the RID of the subject modulo number of segments. The benefit of such design is parallel access to RDF triples distributed to nodes holding segments of RDF data. Furthermore, segments can be replicated to distribute the total workload to the nodes holding replicated RDF data. Communication between nodes is directed by processing nodes via TCP/IP. There is no communication between data nodes.

The 4store query engine is based on relational algebra. The Primary source of optimization is conventional ordering on the joins. However, they also use common subject optimization and cardinality reduction. In spite of considerable work on query optimization, 4store lacks complete query optimization as it is provided by relational query optimizers.

Virtuoso Virtuoso [7, 8, 15] is a multi-model database management system based on relational database technology. The approach of Virtuoso is to treat a triple store as a table composed of four columns. The main concept of the approach to the management of RDF data is to exploit existing relational techniques and to add functionality to RDBMS in order to deal with features specific to RDF data. The most important aspects considered by Virtuoso designers include extending SQL types with RDF data types, dealing with unpredictable object sizes, providing efficient indexing, extending relational statistics to cope with an RDF store based on a single table, and ensuring efficient storage of RDF data.

Virtuoso integrates SPARQL into SQL. SPARQL queries are translated into SQL during parsing. SPARQL has in this way all aggregation functions. SPARQL union is translated directly into SQL and SPARQL optional is translated into left outer join. Since RDF triples are stored in one quad table, relational statistics is not useful. Virtuoso uses sampling during query translations to estimate the cost of alternative plans. Basic RDF inference on TBox is done using query rewriting. For ABox reasoning, Virtuoso expands the semantics of $owl : same - as$ by transitive closure.

Hexastore The Hexastore [21] approach to RDF storage system uses triples as the basis for storing RDF data. The problems of existent triple stores pursued are the scalability of RDF databases in a distributed environment and the complete implementation of query processors including query optimization, persistent indexes, and other issues inherent in database technology.

Six indexes are defined on top of a table with three columns, one for each combination of three columns. The index used for the implementation has three levels ordered by a particular combination of SPO attributes. Each level is sorted in this way, which enables the use of ordering for optimizations during query evaluation. The proposed index provides a natural representation of multi-valued properties and allows for the fast implementation of merge-join, intersection, and union.

5 Conclusion and Future Work

The rough design of the big3store system and precise implementation of query execution module (QEM) were presented. A semantic distribution method of RDF data and a distributed query execution method for RDF storage managers were presented. The storage manager big3store converts SPARQL queries into query tree structures using RDF algebra formalism. Nodes of those tree structures are represented by independent query node processes that execute the query autonomously and in a highly parallel manner while sending asynchronous messages to each other. The semantic data distribution method decreases the number of inter-server messages during query executions by using the semantic properties of RDF data sets.

This research is currently at the preliminary stage, and so far only the query execution module has been implemented and tested. While we are currently focusing on the effective execution of SPARQL queries in distributed computational environments, our future work will include the implementation of the minimum big3store system, benchmarks using large-scale triple data, and the confirmation of the efficiency of the proposed methods.

6 Acknowledgments

This work was supported by the Slovenian Research Agency and the ICT Programme of the EC under PlanetData (ICT-NoE-257641).

References

1. R. Angles and C. Gutierrez. The expressive power of sparql. In *Proceedings of the 7th International Conference on The Semantic Web*, ISWC '08, pages 114–129, Berlin, Heidelberg, 2008. Springer-Verlag.
2. R. Angles and C. Gutierrez. Survey of graph database models. *ACM Comput. Surv.*, 40(1):1:1–1:39, Feb. 2008.
3. J. Armstrong. *Programming Erlang: Software for a Concurrent World*. Pragmatic Bookshelf, 2013.

4. C. Bizer, J. Lehmann, G. Kobilarov, S. Auer, C. Becker, R. Cyganiak, and S. Hellmann. {DBpedia} - a crystallization point for the web of data. *Web Semantics: Science, Services and Agents on the World Wide Web*, 7(3):154 – 165, 2009. The Web of Data.
5. D. Daniels, P. G. Selinger, L. M. Haas, B. G. Lindsay, C. Mohan, A. Walker, and P. F. Wilms. Introduction to distributed query compilation in r*. IBM Research Report RJ3497 (41354), IBM, June 1982.
6. Dublin core metadata inatiative. http://dublincore.org/, 2013.
7. O. Erling and I. Mikhailov. Rdf support in the virtuoso dbms. In *CSSW*, pages 59–68, 2007.
8. O. Erling and I. Mikhailov. Rdf support in the virtuoso dbms. In *Networked Knowledge - Networked Media*, volume 221 of *Studies in Computational Intelligence*, pages 7–24, 2009.
9. M. J. Flynn, O. Mencer, V. Milutinovic, G. Rakocevic, P. Stenstrom, R. Trobec, and M. Valero. Moving from petaflops to petadata. *Commun. ACM*, 56(5):39–42, May 2013.
10. S. Ghemawat, H. Gobioff, and S.-T. Leung. The google file system. In *Proceedings of the nineteenth ACM symposium on Operating systems principles*, SOSP '03, pages 29–43, New York, NY, USA, 2003. ACM.
11. S. Harris and N. Gibbins. 3store: Efficient bulk rdf storage. In *1st International Workshop on Practical and Scalable Semantic Systems (PSSS'03)*, pages 1–15, 2003. Event Dates: 2003-10-20.
12. S. Harris, N. Lamb, and N. Shadbolt. 4store: The design and implementation of a clustered rdf store. In *Proceedings of the The 5th International Workshop on Scalable Semantic Web Knowledge Base Systems*, 2009.
13. J. Hoffart, F. M. Suchanek, K. Berberich, and G. Weikum. Yago2: A spatially and temporally enhanced knowledge base from wikipedia. *Artificial Intelligence*, 194(0):28 – 61, 2013. Artificial Intelligence, Wikipedia and Semi-Structured Resources.
14. K. Nitta and I. Savnik. Survey of rdf storage managers. Technical Report (In preparation), Yahoo Japan Research & FAMNIT, University of Primorska, 2013.
15. OpenLink Software Documentation Team. *OpenLink Virtuoso Universal Server: Documentation*, 2009.
16. Owl 2 web ontology language. http://www.w3.org/TR/owl2-overview/, 2012.
17. Rdf schema. http://www.w3.org/TR/rdf-schema/, 2004.
18. I. Savnik. On using object-relational technology for querying lod repositories. In *The Fifth International Conference on Advances in Databases, Knowledge, and Data Applications*, DBKDA 2013, pages 39–44, Jan. 2013. Dates: from January 27, 2013 to February 1, 2013.
19. I. Savnik and K. Nitta. Algebra of rdf graphs. Technical Report (In preparation), FAMNIT, University of Primorska, 2013.
20. M. Schmidt, M. Meier, and G. Lausen. Foundations of sparql query optimization. In *Proceedings of the 13th International Conference on Database Theory*, ICDT '10, pages 4–33, New York, NY, USA, 2010. ACM.
21. C. Weiss, P. Karras, and A. Bernstein. Hexastore: sextuple indexing for semantic web data management. *Proc. VLDB Endow.*, 1(1):1008–1019, Aug. 2008.
22. T. White. *Hadoop: The Definitive Guide*. OâĂŹReilly Media, Inc., 2009.
23. Xml schema. http://www.w3.org/XML/Schema, 2012.
24. Y. Yan, C. Wang, A. Zhou, W. Qian, L. Ma, and Y. Pan. Efficient indices using graph partitioning in rdf triple stores. In *Proceedings of the 2009 IEEE International Conference on Data Engineering*, ICDE '09, pages 1263–1266, Washington, DC, USA, 2009. IEEE Computer Society.

Distributed OWL EL Reasoning: The Story So Far

Raghava Mutharaju, Pascal Hitzler, and Prabhaker Mateti

Wright State University, OH, USA.
{mutharaju.2, pascal.hitzler, prabhaker.mateti}@wright.edu

Abstract. Automated generation of axioms from streaming data, such as traffic and text, can result in very large ontologies that single machine reasoners cannot handle. Reasoning with large ontologies requires distributed solutions. Scalable reasoning techniques for RDFS, OWL Horst and OWL 2 RL now exist. For OWL 2 EL, several distributed reasoning approaches have been tried, but are all *perceived* to be inefficient. We analyze this perception. We analyze completion rule based distributed approaches, using different characteristics, such as dependency among the rules, implementation optimizations, how axioms and rules are distributed. We also present a distributed queue approach for the classification of ontologies in description logic $\mathcal{EL}^+$ (fragment of OWL 2 EL).

1 Introduction

The rate at which data is generated is increasing at an alarming rate in this age of Big Data. Data processing techniques should also scale up correspondingly. This also holds true in the case of OWL ontologies and reasoning. Manually constructed ontologies would most likely remain small or medium-sized, in the order of several thousands or up to a few million axioms. Generating axioms automatically from streaming data such as traffic [13] or text [7] can result in very large ontologies. Also, in case of reasoning tasks such as classification, the number of inferred axioms keep increasing until the reasoning task terminates. In some cases the size of the result is 75 times that of the input axioms [24]. This turns out to be problematic for current reasoners in case of very large ontologies. A distributed approach to reasoning not only accommodates large ontologies but also provides more processing power.

While some progress has been made regarding scalable reasoning over RDFS, OWL Horst and OWL 2 RL [22, 23, 21, 20], applying similar techniques to OWL 2 EL turns out to be inefficient. In this paper, we investigate the reasons behind it as well as analyze other distributed approaches to reasoning over ontologies in description logic $\mathcal{EL}^+$, which is a fragment of OWL 2 EL. We also present a distributed version of the reasoning algorithm used in CEL reasoner [5]. The distributed approaches mentioned in [15] are explored in detail here.

Rest of the paper is as follows. Section 2 contains a brief description of $\mathcal{EL}^+$ and classification. In Section 3, three approaches to distributed reasoning are

Normal Form	Completion Rule
$A_1 \sqcap \cdots \sqcap A_n \sqsubseteq B$	**R1** If $A_1, \ldots, A_n \in S(X)$, $A_1 \sqcap \cdots \sqcap A_n \sqsubseteq B \in \mathcal{O}$, and $B \notin S(X)$ then $S(X) := S(X) \cup \{B\}$
$A \sqsubseteq \exists r.B$	**R2** If $A \in S(X)$, $A \sqsubseteq \exists r.B \in \mathcal{O}$, and $(X, B) \notin R(r)$ then $R(r) := R(r) \cup \{(X, B)\}$
$\exists r.A \sqsubseteq B$	**R3** If $(X, Y) \in R(r)$, $A \in S(Y)$, $\exists r.A \sqsubseteq B \in \mathcal{O}$, and $B \notin S(x)$ then $S(X) := S(X) \cup \{B\}$
$r \sqsubseteq s$	**R4** If $(X, Y) \in R(r)$, $r \sqsubseteq s \in \mathcal{O}$, and $(X, Y) \notin R(s)$ then $R(s) := R(s) \cup \{(X, Y)\}$
$r \circ s \sqsubseteq t$	**R5** If $(X, Y) \in R(r)$, $(Y, Z) \in R(s)$, $r \circ s \sqsubseteq t \in \mathcal{O}$, $(x, Z) \notin R(t)$ then $R(t) := R(t) \cup \{(X, Z)\}$

Table 1. Completion rules for classifying $\mathcal{EL}^+$ ontologies

described. Section 4 offers alternative evaluation strategies. In Section 5, some possible future directions are mentioned and Section 6 contains some related work. We conclude in Section 7.

2 Preliminaries

We briefly introduce the description logic $\mathcal{EL}^+$. Let the concept names be denoted by N_C, role names by N_R and $N_C^\top$ denotes N_C including $\top$. Concepts in $\mathcal{EL}^+$ are formed according to the grammar

$$A ::= C \mid \top \mid A \sqcap B \mid \exists r.B$$

where $C \in N_C^\top$, $r \in N_R$, and A, B over (possibly complex) concepts. An $\mathcal{EL}^+$ ontology is a finite set of *general concept inclusions* (*GCIs*) $A \sqsubseteq B$ and *role inclusions* (*RIs*) $r_1 \circ \cdots \circ r_n \sqsubseteq r$, where $A, B \in N_C^\top$, n is a positive integer and $r, r_1, \ldots, r_n \in N_R$.

The reasoning task that we consider here is *classification* – the computation of the complete subsumption hierarchy of all concept names occurring in the ontology. Other reasoning tasks such as concept satisfiability can be reduced to classification. Classification is computed using a set of completion rules shown in Table 1. It requires the input ontology $\mathcal{O}$ to be in *normal form*, where all concept inclusions have one of the forms

$$A \sqsubseteq B \mid A_1 \sqcap \ldots \sqcap A_n \sqsubseteq B \mid A \sqsubseteq \exists r.B \mid \exists r.A \sqsubseteq B$$

and all role inclusions have the form $r \sqsubseteq s$ or $r \circ s \sqsubseteq t$. $A, A_1, \ldots, A_n, B \in N_C^\top$ and $r, s, t \in N_R$.

The transformation into normal form can be done in linear time [2], and the process potentially introduces concept names not found in the original ontology. The normalized ontology is a *conservative extension* of the original, in the sense that every model of the original can be extended into one for the normalized ontology. In the rest of the paper, we assume that all of the ontologies we deal with are already in normal form.

The classification rules make use of two mappings S and R, $S\colon N_C^\top \mapsto 2^{N_C^\top}$, $R\colon N_R \mapsto 2^{(N_C^\top \times N_C^\top)}$. Intuitively, $B \in S(A)$ implies $A \sqsubseteq B$, while $(A, B) \in R(r)$ implies $A \sqsubseteq \exists r.B$. Before applying the rules, for each element $X \in N_C^\top$, $S(X)$ is initialized to contain $\{X, \top\}$, and $R(r)$, for each role name r, is initialized to $\varnothing$. The sets $S(X)$ and $R(r)$ are then extended by applying the completion rules shown in Table 1. Here we consider two ways in which these rules are applied to the axioms, which are described in later sections.

Classification of ontologies using the completion rules is guaranteed to terminate in polynomial time relative to the size of the input ontology, and it is also sound and complete. Proofs can be found in [2]. For further background on description logics and how they relate to the Web Ontology Language OWL, please see [3, 11].

In this paper, we consider only completion rule based distributed approaches for classification.

We use the terms node and machine interchangeably throughout the paper.

3 Distributed Classification

We analyze three different distributed approaches to $\mathcal{EL}^+$ classification. Among them two have been published previously.

3.1 Prologue

Before embarking on a parallelization effort, it will be useful to check how amenable it is for parallelization. Note that we are using the term parallelization to mean the following – group of processes co-operating with each other to accomplish a common goal. These processes could either be running on the same machine or on different machines. Here, we are interested in the latter, in which case, there is no shared memory.

Dependency among the completion rules of Table 1 is shown in Figure 1. Every rule is dependent on one or more rules. So, in this case, applying rules to axioms cannot be an embarrassingly parallel computation. When these rules are applied in a distributed environment, some amount of communication is required among the nodes handling the rules which slows down the system.

Contrary to this, in RDFS and OWL Horst, little or no dependency exists among the rules and thus embarrassingly parallel computations are possible [23, 21]. As a result of this, in these cases, linear or sometimes better than linear performance with respect increasing nodes was possible.

3.2 MapReduce Approach

Taking the lead from the application of MapReduce to RDFS and OWL Horst reasoning, an attempt was made in [17, 24] to use it for $\mathcal{EL}^+$ reasoning.

MapReduce is a programming model for distributed processing of data on clusters of machines [8]. MapReduce task consists of two main phases: map

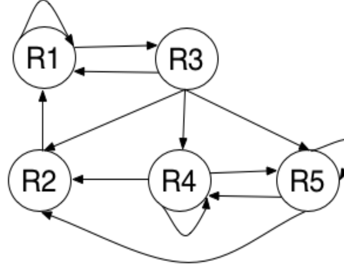


Fig. 1. Dependency among the rules is shown by a directed arrow. A → B indicates that the input of A is dependent on the output of B.

and reduce. In map phase, a user-defined function receives a key-value pair and outputs a set of key-value pairs. All the pairs sharing the same key are grouped and passed to the reduce phase. A user-defined reduce function is set up to process the grouped pairs. Completion of a task might involve several such map and reduce cycles. The map and reduce functions can be represented as

$$\text{Map} : (k_1, v_1) \mapsto \text{list}(k_2, v_2)$$

$$\text{Reduce} : (k_2, \text{list}(v_2)) \mapsto \text{list}(v_3)$$

The completion rules in Table 1 are slightly modified to suit the key-value nature of MapReduce approach. The modified rules are given in Table 2. In the map phase, preconditions of the rules are checked and in the reduce phase, conclusion of the rules are computed. For each concept $X \in N_C^{\top}$ and $r \in N_R$, $S(X)$ is initialized to $\{X, \top\}$ and $R(r), P(X), Q(X)$ are initialized to $\varnothing$. Rules R1, R3 and R5 from Table 1 cannot be dealt with using MapReduce approach, since they have multiple join conditions, which is the reason to split them in the modified rules.

The general strategy used in this approach is given in Algorithm 1. At the end of each iteration, duplicates are removed from S, R, P, Q. Algorithm terminates when there are no changes made by the application of all the rules.

From [24], the performance of this approach and comparison with other reasoners is given in Table 3. The experiments were run on a Hadoop cluster with 8 nodes. Each node has a 2-core, 3GHz processor with 2GB RAM. Although the experiments were conducted on machines with less memory, the evaluation of [17] on machines with more memory (16GB) and larger ontologies has similar performance.

3.2.1 Analysis

1. *Pros* Aspects such as parallelization and fault tolerance are taken care of by the framework.

Normal Form		Completion Rule	Key
$A_1 \sqcap A_2 \sqsubseteq B$	**R1-1**	If $A_1 \in S(X)$ and $A_1 \sqcap A_2 \sqsubseteq B \in \mathcal{O}$ then $P(X) := P(X) \cup \{(A_2, B)\}$	A_1
$(A, B) \in P(X)$	**R1-2**	If $A \in S(X)$ and $((A, B) \in P(X)$ or $A \sqsubseteq B \in \mathcal{O})$ then $S(X) := S(X) \cup \{B\}$	A
$A \sqsubseteq \exists r.B$	**R2**	If $A \in S(X)$ and $A \sqsubseteq \exists r.B \in \mathcal{O}$ then $R(r) := R(r) \cup \{(X, B)\}$	A
$\exists r.A \sqsubseteq B$ for A	**R3-1**	If $A \in S(X)$ and $\exists r.A \sqsubseteq B \in \mathcal{O}$ then $Q(X) := Q(X) \cup \{\exists r.X \sqsubseteq B\}$	A
$\exists r.A \sqsubseteq B$ for r	**R3-2**	If $(X, Y) \in R(r)$ and $\exists r.Y \sqsubseteq B \in Q(X)$ then $S(X) := S(X) \cup \{B\}$	r (or Y)
$r \sqsubseteq s$	**R4**	If $(X, Y) \in R(r)$ and $r \sqsubseteq s \in \mathcal{O}$ then $R(s) := R(s) \cup \{(X, Y)\}$	r
$r \circ s \sqsubseteq t$	**R5**	If $(X, Z) \in R(r)$ and $(Z, Y) \in R(s)$ then $R(r \circ s) := R(r \circ s) \cup \{(X, Y)\}$	Z

Table 2. Revised completion rules for $\mathcal{EL}^+$. The keys are used in the MapReduce algorithm. Note that in R4, r is allowed to be compound, i.e., of the form $s \circ t$.

2. *Cons* There are several disadvantages of this approach. a) Duplicates are generated and an extra step is required to remove the duplicates. b) Since there are dependencies among the rules (Figure 1), MapReduce approach may not best suited. c) In each iteration, the algorithm needs to consider only the newly generated data (compared to last iteration). It is difficult to detect and filter axioms that generate redundant inferences. d) In every iteration, axioms are again reassigned to the machines in the cluster. In the case of RDFS reasoning, schema triples are loaded in-memory and this assignment of schema triples to machines takes place only once. This is not possible in the case of $\mathcal{EL}^+$ reasoning.
3. *Axiom Distribution* Axioms are distributed randomly.
4. *Rule Distribution* In each iteration, all the machines in the cluster, apply the same rule on the local axioms.
5. *Optimizations* Rule R4 is taken care of in the reduce phase of rules R2 and R5. So rule R4 need not be applied again.

3.3 Distributed Queue Approach

Compared to the fixpoint iteration method of rule application, it is claimed that the queue based approach is efficient on a single machine [4]. In this section, we describe a distributed implementation of the queue approach and verify whether the claim also holds true in a distributed setting. First we briefly explain the queue approach on a single machine from [4] and then describe the distributed implementation of it.

For each concept in $N_C^\top$, a queue is assigned. Instead of applying the rules mechanically, in the queue approach, appropriate rules are *triggered* based on the type of entries in the queue. The possible entries in the queue are of the

$S(X) \leftarrow \{X, \top\}$, for each $X \in N_C^\top$
$R(r) \leftarrow \{\}$, for each $r \in N_R$
$P(X) \leftarrow \{\}$, for each $X \in N_C^\top$
$Q(X) \leftarrow \{\}$, for each $X \in N_C^\top$
repeat
 Old.$S(X) \leftarrow S(X)$;
 Old.$R(r) \leftarrow R(r)$;
 Old.$P(X) \leftarrow P(X)$;
 Old.$Q(X) \leftarrow Q(X)$;
 $P(X) := P(X) \cup$ apply R1-1;
 $S(X) := S(X) \cup$ apply R1-2;
 $R(r) := R(r) \cup$ apply R2;
 $Q(X) := Q(X) \cup$ (apply R3-1);
 $S(X) := S(X) \cup$ apply R3-2;
 $R(r) := R(r) \cup$ apply R4;
 $R(r) := R(r) \cup$ apply R5;
until *((Old.$S(X) = S(X)$) and (Old.$R(r) = R(r)$) and (Old.$P(X) = P(X)$) and (Old.$Q(X) = Q(X)$));*

Algorithm 1: General strategy for applying rules in MapReduce approach

Ontology	#Axioms	ELK	jCEL	Pellet	MR Approach
1-GALEN	90000	2.3	116.2	742.4	6552.5
2-GALEN	178000	5.5	243.7	OOM	11952.5
4-GALEN	352000	11.6	OOM	OOM	19908.3
8-GALEN	703000	OOM	OOM	OOM	38268.7

Table 3. Classification time (in seconds) of MapReduce (MR) approach. OOM indicates Out Of Memory.

form $B_1, \ldots, B_n \rightarrow B'$ and $\exists r.B$ with $B_1, \ldots, B', B \in N_C^\top$ and $r \in N_R$. If $n = 0$, $B_1, \ldots, B_n \rightarrow B'$ is simply written as B'. $\widehat{\mathcal{O}}$ is a mapping from a concept to sets of queue entries as follows.

- if $A_1 \sqcap \ldots \sqcap A_n \sqsubseteq B \in \mathcal{O}$ and $A_i = A$, then $A_1 \sqcap \ldots \sqcap A_{i-1} \sqcap A_{i+1} \sqcap \ldots \sqcap A_n \rightarrow B \in \widehat{\mathcal{O}}(A)$
- if $A \sqsubseteq \exists r.B \in \mathcal{O}$, then $\exists r.B \in \widehat{\mathcal{O}}(A)$
- if $\exists r.A \sqsubseteq B \in \mathcal{O}$, then $B \in \widehat{\mathcal{O}}(\exists r.A)$

For each concept $A \in N_C^\top$, queue(A) is initialized to $\widehat{\mathcal{O}}(A) \cup \widehat{\mathcal{O}}(\top)$. For each queue, an entry is fetched and Algorithm 2 is applied. The procedure in Algorithm 3 is called by process(A, X) whenever a new pair of (A, B) is added to $R(r)$. Note that, for any concept A, $\widehat{\mathcal{O}}(A)$ does not change during the application of the two procedures (process, process-new-edge); $S(A)$, queue(A) and $R(r)$ keep changing.

In the distributed setup, axioms are represented as key-value pairs as shown in Table 4. For axioms of the form $A_1 \sqcap \ldots \sqcap A_n \sqsubseteq B$, for each A_i (key) in the conjunct, $(A_1, \ldots, A_{i-1}, A_{i+1}, \ldots, A_n, B)$ is associated as its value.

if $X = B_1, \ldots, B_n \rightarrow B'$ *and* $B' \notin S(A)$ **then**
  if $B_1, \ldots, B_n \in S(A)$ **then**
    continue with $X \leftarrow B'$;
  else
    return;
if X *is a concept name and* $X \notin S(A)$ **then**
  $S(A) \leftarrow S(A) \cup \{X\}$;
  queue$(A) \leftarrow$ queue$(A) \cup \widehat{\mathcal{O}}(X)$;
  forall the *concept names* B *and role names* r *with* $(B, A) \in R(r)$ **do**
    queue$(B) \leftarrow$ queue$(B) \cup \widehat{\mathcal{O}}(\exists r.X)$;
if X *is an existential restriction* $\exists r.B$ *and* $(A, B) \notin R(r)$ **then**
  process-new-edge(A, r, B);

Algorithm 2: process(A, X)

forall the *role names* s *with* $r \sqsubseteq^*_{\widehat{\mathcal{O}}} s$ **do**
  $R(s) \leftarrow R(s) \cup \{(A, B)\}$;
  queue$(A) \leftarrow$ queue$(A) \cup \bigcup_{\{B' \mid B' \in S(B)\}} \widehat{\mathcal{O}}(\exists s.B')$;
  forall the *concept names* A' *and role names* t, u **do**
    $t \circ s \sqsubseteq u \in \mathcal{O}$ and $(A', A) \in R(t)$ and $(A', B) \notin R(u)$ do
    process-new-edge(A', u, B);
  forall the *concept names* B' *and role names* t, u **do**
    $s \circ t \sqsubseteq u \in \mathcal{O}$ and $(B, B') \in R(t)$ and $(A, B') \notin R(u)$ do
    process-new-edge(A, u, B');

Algorithm 3: process-new-edge(A, r, B)

Axioms are distributed across the machines in the cluster based on their keys. A hash function, H maps a unique key, K, to a particular node, N, in the cluster.

$$H \colon K \mapsto N$$

For each concept A, care is taken to map $\widehat{\mathcal{O}}(A)$, queue(A) and $S(A)$ to the same node. This localizes the interaction (read/write) between these three sets, which in turn improves the performance. In order not to mix up the keys among these three sets, unique namespace is used along with the key. For example, $O : A, Q : A, S : A$, for $\widehat{\mathcal{O}}(A)$, queue(A) and $S(A)$ respectively, but, for the hash function, A is used in all the three cases.

After the axioms are loaded, each machine applies Algorithm 2 to only the queues local to it. In order to read/write to the non-local values, each machine uses the hash function, H. Each machine acts as a reasoner and cooperates with other machines to get the missing values and perform the classification task.

A single process called *Termination Controller* (TC), keeps track of the status of computation across all the nodes in the cluster. TC receives either *DONE* or *NOT-DONE* message from each machine. A double check termination strategy is

Axiom	Key	Value
$A \sqsubseteq B$	A	B
$A_1 \sqcap \ldots \sqcap A_n \sqsubseteq B$	A_i	$(A_1, \ldots, A_{i-1}, A_{i+1}, \ldots, A_n, B)$
$A \sqsubseteq \exists r.B$	A	(r, B)
$\exists r.A \sqsubseteq B$	(r, A)	B
$r \sqsubseteq s$	r	s
$r \circ s \sqsubseteq t$	r	(s, t)
	s	(r, t)

Table 4. Key-Value pairs for axioms

```
msgCount ← 0;
currentState ← NO-CHECK;
on pid ? status-msg → {
if status-msg = DONE then
    msgCount ← msgCount + 1;
    if msgCount = TOTAL-NODES then
        if currentState = NO-CHECK then
            currentState ← SINGLE-CHECK-DONE;
            broadcast(CHECK-AND-RESTART);
        else if currentState = SINGLE-CHECK-DONE then
            currentState ← DOUBLE-CHECK-DONE;
            broadcast(TERMINATE);
else if status-msg = NOT-DONE then
    msgCount ← 0;
    currentState ← NO-CHECK;
    pid ! CONTINUE-WORKING;
}
```

Algorithm 4: Termination Controller, TC

followed here. TC waits till it receives a *DONE* message from all the machines in the cluster. It then asks all the nodes to check if any local queues are non-empty. This is required because, after a node is done with OneIteration (Algorithm 5), there is a possibility of other nodes inserting values in the queues of this node. If this condition does indeed arise then a *NOT-DONE* message is sent to TC. TC resets its state to *NO-CHECK* and implements the double check termination strategy again. The pseudocode of TC is given in Algorithm 4. To simplify, TC is single threaded and works on only one message at a time. Process named *Job Controller* runs on each node of the cluster and implements the queue based algorithm. This is shown in Algorithm 6.

This approach is implemented in Java and the key-value store used is Redis[1]. Our system is called *DQuEL* and the source code is available at `https://github.com/raghavam/DQuEL`. We used a 13-node cluster with each node hav-

[1] `http://redis.io`

```
queues ← GetNonEmptyLocalQueues();
forall the queue A ∈ queues do
    forall the entry X ∈ A do
        process(A, X);
TC ! DONE;
```

Algorithm 5: OneIteration()

```
OneIteration();
on TC ? CHECK-AND-RESTART → {
queues ← GetNonEmptyLocalQueues();
if queues is ∅ then
    TC ! DONE;
else
    TC ! NOT-DONE;
} on TC ? CONTINUE-WORKING → OneIteration();
on TC ? TERMINATE → terminate-self;
```

Algorithm 6: Job Controller

ing two quad-core AMD Opteron 2300MHz processors and 12GB of heap size is available to JVM. Timeout limit was set to 2 hours.

Not-Galen, GO, NCI, SNOMED CT and 2-SNOMED ontologies were used for testing. The first three are obtained from `http://lat.inf.tu-dresden.de/~meng/toyont.html` and SNOMED CT can be obtained from `http://www.ihtsdo.org/snomed-ct`. 2-SNOMED is SNOMED replicated twice. The time taken by some popular reasoners such as Pellet, jCEL and ELK on these ontologies is given in Table 5. Table 6 shows the classification times of *DQuEL* with varying nodes.

3.3.1 Analysis Although the results are good for smaller ontologies, this approach turns out to be inefficient for larger ontologies such as SNOMED CT. The following two factors contributes to the inefficiency. a) Batch processing of axioms is not possible because each entry in the queue could be different from the one processed before. It is a known fact that batch processing especially involving communication over networks improves the performance drastically. Batch

Ontology	#Axioms	Pellet	jCEL	ELK
Not-Galen	8,015	12.0	3.0	1.0
GO	28,897	5.0	5.0	2.0
NCI	46,870	6.0	7.0	3.0
SNOMED CT	1,038,481	1,845.0	327.0	24.0
2-SNOMED	2,076,962	OOM	687.0	64.0

Table 5. Classification time (in seconds) of Pellet, jCEL and ELK reasoners

Ontology	1 node	7 nodes	13 nodes
Not-Galen	153.77	147.07	41.37
GO	165.94	147.28	43.12
NCI	205.62	55.52	30.21
SNOMED CT	TimeOut	TimeOut	TimeOut
2-SNOMED	TimeOut	TimeOut	TimeOut

Table 6. Classification time (in seconds) of DQuEL

processing has been used to the extent possible, but the approach described in the next section is more amenable to that. b) Not all data required for the queue operations is available locally. For example, data of the form $\widehat{\mathcal{O}}(\exists r.B)$ might be present on a different node.

1. *Pros* Good load balancing is possible since the hash function makes an attempt to distribute axioms across the cluster equally.
2. *Cons* Large ontologies like SNOMED CT generate many (X, Y) values which makes rule R3 (Table 1) computation slow compared to other rules. This problem is alleviated in the approach described next, by choosing r as the key in $R(r)$. This spreads $R(r)$ across the cluster and enables more nodes to work on it.
3. *Axiom Distribution* It is a random distribution of axioms.
4. *Rule Distribution* In each iteration, all the rules are applied by every node in the cluster.
5. *Optimizations* $R(r)$ sets involving role chains are duplicated as show in Table 4. This makes it easy to retrieve (X, Y) pairs associated with either r or s in $r \circ s \sqsubseteq t$.

3.4 Distributed Fixpoint Iteration Approach

In fixpoint iteration approach, the completion rules are applied on the axioms iteratively until there are no changes to $S(X), R(r)$. This idea is extended to the distributed setting [16]. The completion rules from Table 2 and the general strategy mentioned in Algorithm 1 are used here. Axioms are represented as key-value pairs.

All the axioms in a normalized ontology fall into one of the normal form categories mentioned in Table 1. This allows axioms to be split into disjoint collections. Each such collection is assigned to a group of nodes in the cluster. Only one rule can be applied on axioms of a particular normal form. So this leads to a clear assignment of axioms and rules to nodes.

Architecture of this approach is shown in Figure 2. The number of nodes per group need not be the same across all the groups. Higher number of nodes are generally assigned to groups handling rules involving roles since they are generally slower.

$U(X)$ is used instead of $S(X)$ in this approach. In the naive fixpoint iteration approach, in order to apply a rule such as R1 from Table 1 on axioms of the form

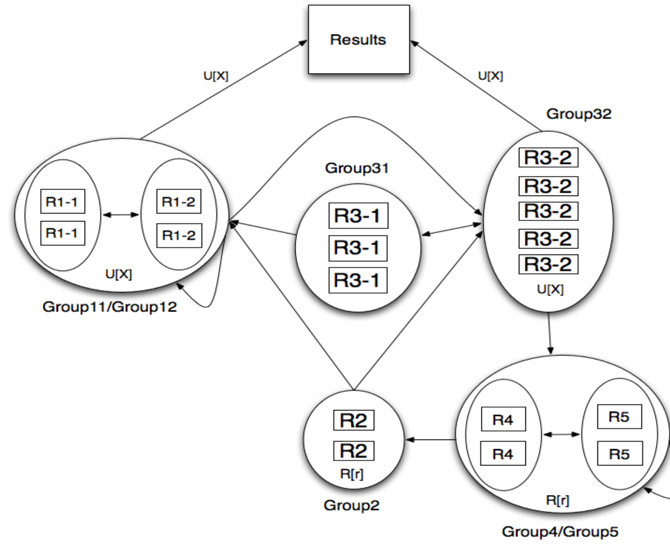


Fig. 2. Node assignment to rules and dependency among the completion rules. Each oval is a collection of nodes (rectangles).

Ontology	7 nodes	9 nodes	12 nodes
Not-Galen	43.0	42.27	41.06
GO	46.20	49.39	51.83
NCI	275.0	168.96	157.36
SNOMED CT	1,610.00	1,335.81	865.89
2-SNOMED	3,238.19	2,687.75	1,699.73

Table 7. Classification time (in seconds) of DistEL

$A_1 \sqcap \ldots \sqcap A_n \sqsubseteq B$, each $S(X)$ needs to be checked for the presence of $A_1, \ldots, A_n$. With $U(X)$, it turns into set intersection of conjuncts, which are generally small in number.

Termination is achieved with the help of barrier synchronization. At the end of each iteration, every node broadcasts a status message indicating whether any changes were made in this iteration and then waits for the other nodes to finish its current iteration. After receiving the update messages from all the nodes, if at least one node made an update then all the nodes continue with their next iteration. Algorithm terminates when no updates are made.

This approach is implemented in Java and makes use of Redis as the key-value store. The same cluster set up and ontologies as mentioned in the previous section are used here. This system is named *DistEL* and is available at `https://github.com/raghavam/DistEL`. The classification time of *DistEL* across several nodes is given in Table 7.

3.4.1 Analysis Although the runtimes of *DistEL* are high compared to existing reasoners, it is better than the other approaches described so far. With more optimizations such as dynamic load balancing, the results can be further improved.

MapReduce approach and the approach described here, use fixpoint iteration for classification. But compared to MapReduce approach, an important advantage is that nodes can communicate with each other. Since the completion rules are interdependent, inter-node communication makes this approach efficient relatively. Since a node deals with all axioms of the same type, batch processing can be used here. Communications involving network and database can be improved a lot with the help of batch processing.

1. *Pros* Axioms and rules are neatly divided across the cluster based on their type.
2. *Cons* Improper load balancing.
3. *Axiom Distribution* Axioms are distributed based on their normal form type.
4. *Rule Distribution* Group of nodes in the cluster handle the same type of rule. All the rules are applied in parallel across the cluster.
5. *Optimizations* a) Batch processing b) only the newly made changes in the previous iteration are considered for the next iteration and c) for $R(r) = \{(X, Y)\}$, instead of considering r as the key (as done in Queue approach), (Y, r) is chosen as the key and value is X. Since distribution of set $R(r)$ is based on the key, this leads to a better spread of $R(r)$ across the cluster.

4 Evaluation Strategy

Comparing the performance of single machine reasoners with distributed reasoners is unfair; not only due to the nature of the computation involved but also due to the following reason. All things being equal, if the time taken on a single machine is t then on n nodes, it takes $p * t/n$ where p is the overhead, $p \geq 1$. In the case of super linear speedup, $p < 1$. But, as we have seen from the results presented here, this reduction in runtime does not happen at all in the case of distributed reasoners. But are all things equal? Considering the steps in the algorithm in both the cases is the same, one main difference is the computations in case of single machine reasoners takes place in-memory whereas for distributed approaches mentioned here either a database or a file system was used. The performance varies vastly in these cases.

A simple comparison of speeds is shown in Table 8. Integers are read and written to a HashMap in the case of RAM. For Redis, pipeline (batch operation) read and write are used. The code used for this experiment is available at `https://gist.github.com/raghavam/2be48a98cae31c418678`. Admittedly, this is a rather simple experiment, but it shows the difference in read/write speeds for simple operations. For read operation, usage of RAM is 43 times faster than Redis and 26 times faster than file. For write operation also, there is a similar variation in performance. For random read and write operations, Redis performs better than a file.

Operation	#Items	RAM	Redis	File
Read	1,000,000	0.0861	3.719	
Write	1,000,000	0.1833	4.688	0.2181

Table 8. Comparison of speed (in seconds) for simple read, write operations when using RAM, Redis and File

A comparison should not be made between a distributed computation and a single machine computation since it is not a like-for-like comparison. But, since some sort of baseline is required, following two strategies can be considered.

1. Re-implement existing reasoners by making use of the storage system that is used in the distributed model. For example, use Redis or a file in jCEL or ELK.
2. Simulate distribution using existing reasoners by running a reasoner on each node of the cluster. A messaging system can be used to facilitate communicate and exchange of missing data on each node. But the performance in this case depends on the axiom distribution. So care should be taken to follow the same axiom distribution model in case of the distributed approach.

5 The Road Ahead

Although some progress has been made in distributed OWL 2 EL reasoning, current results clearly indicate that more needs to be done. Apart from further optimizations to the approaches presented here, following can be tried.

- Module based axiom distribution. For a given set of entities, a module includes all the axioms that are relevant to them [10]. If axioms are distributed based on modules, then perhaps inter-node communication can be reduced.
- Axiom distribution based on graph partitioning. If a graph of an ontology can be constructed then distribution of axioms based on vertex partitioning reduces the dependencies among the axioms.
- Hadoop variants. There are several variants to the core MapReduce approach such as Apache Spark[2], Iterative MapReduce[3], HaLoop[4] which might be more suitable than the core Hadoop's MapReduce.
- Other distributed frameworks. Peer-to-peer techniques such as use of MPI and alternative distributed frameworks such as Akka[5] can be tried. As mentioned earlier, peer-to-peer networks offer more control over communication between nodes when compared to MapReduce.
- Shared memory supercomputers. Since the completion rules are interdependent, may be it would be more efficient if shared memory, massively parallel supercomputers are used. But the disadvantage in this case is that these are specialized machines which are not commonly available.

[2] http://spark.apache.org
[3] http://www.iterativemapreduce.org
[4] https://code.google.com/p/haloop
[5] http://akka.io

6 Related Work

Apart from the work presented here, other approaches to distributed reasoning of OWL 2 EL ontologies have been tried. Distributed resolution technique was applied to $\mathcal{EL}^+$ classification in [19]. A peer-to-peer distributed reasoning approach was presented in [6]. But, both of them do not provide any evaluation. There is some work on parallelization of tableau algorithms related to various description logics [1, 14].

Though not distributed, parallelization of OWL 2 EL classification has been studied in [12, 18]. Classifying EL ontologies on a single machine using a database instead of doing it in memory has been tried in [9].

7 Epilogue

It is possible to have very large ontologies if axioms are generated automatically from streaming data or text. Reasoners should be capable of scaling up to these large ontologies. But, existing reasoners are severely handicapped by their use of only one machine. Scalable and distributed approaches to ontology reasoning is required. We reviewed and analyzed three distributed approaches to OWL EL ontology classification. Apart from this, we discussed some possible future directions and also evaluation strategies that can be followed to make the comparison fair between distributed and single machine approaches.

Acknowledgements. This work was supported by the National Science Foundation under award 1017225 "III: Small: TROn – Tractable Reasoning with Ontologies." Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the National Science Foundation.

References

1. Aslani, M., Haarslev, V.: Parallel TBox Classification in Description Logics – First Experimental Results. In: Proceedings of the 19th European Conference on Artificial Intelligence. pp. 485–490. IOS Press, Amsterdam, The Netherlands (2010)
2. Baader, F., Brandt, S., Lutz, C.: Pushing the EL Envelope. LTCS-Report LTCS-05-01, Chair for Automata Theory, Institute for Theoretical Computer Science, Dresden University of Technology, Germany (2005), `http://lat.inf.tu-dresden.de/research/reports.html`
3. Baader, F., Calvanese, D., McGuinness, D., Nardi, D., Patel-Schneider, P. (eds.): The Description Logic Handbook: Theory, Implementation, and Applications. Cambridge University Press (2007)
4. Baader, F., Lutz, C., Suntisrivaraporn, B.: Is Tractable Reasoning in Extensions of the Description Logic EL Useful in Practice? In: Proceedings of the 2005 International Workshop on Methods for Modalities (M4M-05) (2005)
5. Baader, F., Lutz, C., Suntisrivaraporn, B.: CEL—A Polynomial-time Reasoner for Life Science Ontologies. In: Proceedings of the 3rd International Joint Conference on Automated Reasoning (IJCAR'06), Seattle, WA, USA, August 17-20, 2006.

Lecture Notes in Artificial Intelligence, vol. 4130, pp. 287–291. Springer-Verlag (2006)

6. Battista, A.D.L., Dumontier, M.: A Platform for Reasoning with OWL-EL Knowledge Bases in a Peer-to-Peer Environment. In: Proceedings of the 5th International Workshop on OWL: Experiences and Directions, Chantilly, VA, United States, October 23-24 2009. CEUR Workshop Proceedings, vol. 529. CEUR-WS.org (2009)
7. Cimiano, P.: Ontology Learning and Population from Text: Algorithms, Evaluation and Applications. Springer-Verlag New York, Inc., Secaucus, NJ, USA (2006)
8. Dean, J., Ghemawat, S.: MapReduce: Simplified data processing on large clusters. In: Proceedings of the 6th Symposium on Operating System Design and Implementation (OSDI 2004), December 6-8, 2004, San Francisco, California, USA. pp. 137–150. USENIX Association (2004)
9. Delaitre, V., Kazakov, Y.: Classifying ELH Ontologies In SQL Databases. In: Proceedings of the 5th International Workshop on OWL: Experiences and Directions (OWLED 2009), Chantilly, VA, United States, October 23-24, 2009
10. Grau, B.C., Horrocks, I., Kazakov, Y., Sattler, U.: Just the Right Amount: Extracting Modules from Ontologies. In: Proceedings of the 16th International Conference on World Wide Web. pp. 717–726. WWW '07, ACM, New York, NY, USA (2007), http://doi.acm.org/10.1145/1242572.1242669
11. Hitzler, P., Krötzsch, M., Rudolph, S.: Foundations of Semantic Web Technologies. Chapman & Hall/CRC (2010)
12. Kazakov, Y., Krötzsch, M., Simancik, F.: Concurrent Classification of EL Ontologies. In: 10th International Semantic Web Conference, Bonn, Germany, October 23-27. Lecture Notes in Computer Science, vol. 7031, pp. 305–320. Springer (2011)
13. Lécué, F., Tucker, R., Bicer, V., Tommasi, P., Tallevi-Diotallevi, S., Sbodio, M.L.: Predicting Severity of Road Traffic Congestion using Semantic Web Technologies. In: Proceedings of the 11th Extended Semantic Web Conference (ESWC2014), Anissaras, Crete, Greece, May 25–May 29, 2014. Springer (2014)
14. Liebig, T., Müller, F.: Parallelizing Tableaux-based Description Logic Reasoning. In: Proceedings of the 2007 OTM Confederated International Conference on On the Move to Meaningful Internet Systems - Volume Part II. pp. 1135–1144. OTM'07, Springer-Verlag, Berlin, Heidelberg (2007), http://dl.acm.org/citation.cfm?id=1780453.1780504
15. Mutharaju, R.: Very Large Scale OWL Reasoning through Distributed Computation. In: International Semantic Web Conference (2). Lecture Notes in Computer Science, vol. 7650, pp. 407–414. Springer (2012)
16. Mutharaju, R., Hitzler, P., Mateti, P.: DistEL: A Distributed EL+ Ontology Classifier. In: Liebig, T., Fokoue, A. (eds.) Proceedings of the 9th International Workshop on Scalable Semantic Web Knowledge Base Systems, Sydney, Australia. CEUR Workshop Proceedings, vol. 1046, pp. 17–32. CEUR-WS.org (2013)
17. Mutharaju, R., Maier, F., Hitzler, P.: A MapReduce Algorithm for EL+. In: Proceedings of the 23rd International Workshop on Description Logics (DL 2010), Waterloo, Ontario, Canada, May 4-7, 2010. CEUR Workshop Proceedings, vol. 573. CEUR-WS.org (2010)
18. Ren, Y., Pan, J.Z., Lee, K.: Parallel ABox reasoning of EL ontologies. In: Proceedings of the 2011 Joint International Conference on the Semantic Web. pp. 17–32. JIST'11, Springer, Heidelberg (2012)
19. Schlicht, A., Stuckenschmidt, H.: MapResolve. In: Web Reasoning and Rule Systems – 5th International Conference, RR 2011, Galway, Ireland, August 29-30, 2011. Lecture Notes in Computer Science, vol. 6902, pp. 294–299. Springer (2011)

20. Urbani, J., van Harmelen, F., Schlobach, S., Bal, H.E.: QueryPIE: Backward Reasoning for OWL Horst over Very Large Knowledge Bases. In: 10th International Semantic Web Conference, Bonn, Germany, October 23-27, 2011. Lecture Notes in Computer Science, vol. 7031, pp. 730–745. Springer (2011)
21. Urbani, J., Kotoulas, S., Maassen, J., Van Harmelen, F., Bal, H.: WebPIE: A Web-scale Parallel Inference Engine Using MapReduce. Journal of Web Semantics. 10, 59–75 (Jan 2012), http://dx.doi.org/10.1016/j.websem.2011.05.004
22. Urbani, J., Kotoulas, S., Oren, E., van Harmelen, F.: Scalable Distributed Reasoning Using MapReduce. In: Proceedings of the 8th International Semantic Web Conference, ISWC 2009, Chantilly, VA, USA, October 25-29, 2009
23. Weaver, J., Hendler, J.A.: Parallel Materialization of the Finite RDFS Closure for Hundreds of Millions of Triples. In: 8th International Semantic Web Conference, ISWC 2009, Chantilly, VA, USA, October 25-29, 2009. Lecture Notes in Computer Science, vol. 5823, pp. 682–697. Springer (2009)
24. Zhou, Z., Qi, G., Liu, C., Hitzler, P., Mutharaju, R.: Scale reasoning with fuzzy-EL+ ontologies based on MapReduce. In: Proceedings of the IJCAI-2013 Workshop on Weighted Logics for Artificial Intelligence, WL4AI-2013, Beijing, China, August 2013. pp. 87–93 (2013)