Flexible querying for SPARQL

A. Calì, R. Frosini, A. Poulovassilis, P. T. Wood

Department of Computer Science and Information Systems,
Birkbeck, University of London
London Knowledge Lab

Overview of the presentation

1. Introduction
   - Motivation
   - Examples - Querying YAGO
2. Flexible Querying for SPARQL - SPARQL^{AR}
   - Query Approximation
   - Query Relaxation
   - Query Rewriting Algorithm
   - Complexity of Query Answering
   - Performance of Query Evaluation
3. Conclusions and Future Work

Motivation

Why SPARQL?

- Most prominent language for querying RDF data, which is a standard for Linked Open Data (LOD).
- SPARQL 1.1 enables Regular Path queries, useful for querying graph-structured data.

Querying with SPARQL - What does the user need to know?

- The structure of the RDF graph.
- The correct URIs to use.
- The labels appearing in the RDF graph.

Motivation

Querying with SPARQL - Possible problems

- Lack of full knowledge of the structure of the dataset.
- Complexity of the structure of the dataset.
- Lack of knowledge of the URIs used in the dataset.
- Changes in the structure and URIs of the dataset over time.

Querying YAGO

Example query

SPARQL query over the YAGO RDF dataset, aiming to find events taking place in London on 2/09/1666:

```
SELECT *
WHERE{
    ?x <in> "London" .
    ?x <on> "2/09/1666"
}
```

This query does not return any answers because there are no property edges named “in” or “on” in YAGO.

Querying YAGO

We need to change the query in order to get some answers (by *replacing* the two property labels `in` and `on`, and *inserting* another property label, `rdfs:label`, in the second triple):

Modified query

```
SELECT *
WHERE{
    ?x <happenedIn>/rdfs:label "London" .
    ?x <happenedOnDate> "2/09/1666"
}
```

This query returns an event: "The Great Fire of London".

Querying YAGO

A further possible modification is to *relax* the second triple), using the knowledge that the domain of `happenedOnDate` is `Event`:

Modified query

```
SELECT *
WHERE{
    ?x <happenedIn>/rdfs:label "London" .
    ?x rdf:type <Event>
}
```

This query returns every event that occurred in London.

Our language: SPARQL^{AR}

SPARQL^{AR} extends SPARQL 1.1 with two operators, APPROX and RELAX.

Example query

```
SELECT *
WHERE{
    APPROX(?x <in> "London") .
    RELAX(?x <happenedOnDate> "2/09/1666")
}
```

This query returns all the answers returned by the previous query, plus additional answers. Answers are returned in *ranked* order, according to their distance from the "exact" form of the query i.e. the query without the APPROX and RELAX operators.

How do we rank query answers?

Answers returned by a SPARQL^{AR} query Q have an associated cost: every *approximation* operation - i.e. every insertion, deletion or substitution of a property label in Q - is assigned an *edit cost*; every *relaxation* operation w.r.t. RDFS entailment rules applied to Q is assigned a *relaxation cost*; these costs are summed to given an overall cost for a modified query, Q', and hence an overall cost for the answers returned by evaluating Q'.
See our paper in ODBASE 2014 for details.

Research challenges:

- Semantics of SPARQL^{AR}.
- Complexity of Query Answering.
- Generating query answers and their cost.
- Proof of soundness and completeness.
- Performance of query evaluation.

SPARQL^{AR} - Approximation

Modifies the query using edit operations on property labels: insertion, deletion, substitution.

Example

Suppose we want to know every airport directly connected to Rome, using the YAGO dataset:

```
SELECT *
    WHERE{
    <Rome> isConnectedTo ?x
}
```

This query returns no answers.

SPARQL^{AR} - Approximation

If we apply approximation to the query it is possible to retrieve the answers:

Example

```
SELECT *
    WHERE{
    APPROX(<Rome> isConnectedTo ?x)
}
```

In YAGO, <Rome> is linked to its airports through the predicate linksTo. One step of approximation - inserting linksTo into the query, at an edit cost of c_i - automatically generates this query, which will return the required answers at a "distance" of c_i:

```
SELECT *
    WHERE{
    <Rome> linksTo/isConnectedTo ?x
}
```

SPARQL^{AR} - Relaxation

Exploits the RDFS entailment rules.

Example

Given this ontology, K: happenedOnDate dom Event; endedOnDate sp happenedOnDate; startedOnDate sp happenedOnDate. And this query:

```
SELECT *
    WHERE{
    <World_War_I> startedOnDate ?x
}
```

The query will return the date in which the first world war started.

Example continued

Considering now this query:

```
SELECT *
    WHERE{
    RELAX(<World_War_I> startedOnDate ?x)
}
```

Applying the "subproperty" rexation rule - at a relaxation cost of c_{sp} - automatically generates this query, which will return the previous answer (at "distance" 0) and also the date on which the first world war ended at "distance" c_i:

```
SELECT *
    WHERE{
    <World_War_I> happenedOnDate ?x
}
```

Complexity of Query Answering

- In our ODBASE 2014 paper, we study the complexity of query answering considering the *combined complexity* (with both the query and the graph as input), the *data complexity* (with the graph as input and the query fixed), and the *query complexity* (with the query as input and the graph fixed) of this problem.
- We provide tight complexity bounds for several SPARQL fragments, and show that the data, query and combined complexity of SPARQL/SPARQL 1.1 are not impacted by our extensions with the APPROX and RELAX operators.

Complexity of Query Answering

Summary of results:

Operators	Data Complexity	Query Complexity	Combined Complexity
AND, FILTER	$O(\|E\|)$	$O(\|Q\|)$	$O(\|E\| \cdot \|Q\|)$
AND, FILTER, RegEx	$O(\|E\|)$	$O(\|Q\|^2)$	$O(\|E\| \cdot \|Q\|^2)$
RELAX, APPROX	$O(\|E\|)$	P-Time	P-Time
RELAX, APPROX, AND, FILTER, RegEx	$O(\|E\|)$	P-Time	P-Time
AND, SELECT	P-Time	NP-Complete	NP-Complete
RELAX, APPROX, AND, FILTER, RegEx, SELECT	P-Time	NP-Complete	NP-Complete

Flexible Querying

Query Rewriting Algorithm:

Given a query Q, this constructs a set of queries rew(Q) that do not contain APPROX and RELAX such that $[\![Q]\!]_G = \bigcup_{Q' \in rew(Q)} [\![Q']\!]_G$ for every graph G and ontology K.

- Starting from Q, we apply one step of approximation to triple patterns of Q that are APPROXed and one step of relaxation to triple patterns of Q that are RELAXed, generating a set of queries R each with some associated cost.
- We repeat this for each query in R, generating queries of greater cost and adding them to R.
- We limit the number of queries generated by supplying the rewriting algorithm with a maximum query cost as input.

In the extended version of our ODBASE 2014 paper, we specify the query rewriting algorithm in detail and show that it is sound and complete w.r.t the semantics of SPARQL^{AR}.

Flexible Querying

Query Evaluation:

To compute the query answers, we incrementally apply an evaluation function, *eval*, to each query generated by the rewriting algorithm, in ranked order of the cost of the queries. To each mapping returned by *eval* we assign the cost of the query.
If we generate a particular mapping more than once we keep the one with the lowest cost.

Experimental Results

- We have implemented our query rewriting and query evaluation algorithms and have conducted preliminary trials of query evaluation performance over the YAGO SPARQL endpoint (to be reported in an upcoming paper) and the Lehigh University Benchmark (LUBM) - reported in the OBDASE 2014 paper.
- Using the latter, we generated 3 datasets: D_1 with $149,973$ triples, D_2 with $421,562$ triples, and D_3 with $673,416$ triples.
- We ran our experiments on a Windows 7 computer with 4Gb of RAM and a quadcore-core i7 CPU at 2.0Ghz. The query evaluation algorithm was implemented in Java and we used Jena for the SPARQL query execution.

Experimental Results - 1

Trialled queries, showing number of answers/time for the exact form of each query on D_1-D_3 and the Approxed/Relaxed form likewise; for the latter, the cost of all edit and relaxation operations was set to 1 and the maximum cost was also set to 1.

	Q_1	Q_2	Q_3	Q_4
D_1	23/0.001s	34/0.004s	0/0.197s	331/0.129s
D_2	63/0.006s	34/0.005s	0/0.64s	883/0.296s
D_3	100/0.007s	34/0.006s	0/1.67s	1381/0.517s
D_1/AR	840/0.015s	605/0.421s	743/0.924s	348/60.7s
D_2/AR	2326/0.055s	605/1s	756/3.17s	925/451s
D_3/AR	3706/0.105s	605/1.6s	767/4.44s	1461/1492s

Q_1:

```
SELECT *
WHERE {RELAX(?X ub:headOf ?Z)}
```

Experimental Results - 2

	Q_1	Q_2	Q_3	Q_4
D_1	23/0.001s	34/0.004s	0/0.197s	331/0.129s
D_2	63/0.006s	34/0.005s	0/0.64s	883/0.296s
D_3	100/0.007s	34/0.006s	0/1.67s	1381/0.517s
D_1/AR	840/0.015s	605/0.421s	743/0.924s	348/60.7s
D_2/AR	2326/0.055s	605/1s	756/3.17s	925/451s
D_3/AR	3706/0.105s	605/1.6s	767/4.44s	1461/1492s

Q_2:

```
SELECT * WHERE {
APPROX(?X ub:worksFor ub:University0) . ?X ub:name ?Y1 .
?X ub:emailAddress ?Y2 . ?X ub:telephone ?Y3}
```

Experimental Results - 3

	Q_1	Q_2	Q_3	Q_4
D_1	23/0.001s	34/0.004s	0/0.197s	331/0.129s
D_2	63/0.006s	34/0.005s	0/0.64s	883/0.296s
D_3	100/0.007s	34/0.006s	0/1.67s	1381/0.517s
D_1/AR	840/0.015s	605/0.421s	743/0.924s	348/60.7s
D_2/AR	2326/0.055s	605/1s	756/3.17s	925/451s
D_3/AR	3706/0.105s	605/1.6s	767/4.44s	1461/1492s

Q_3:

```
SELECT * WHERE { RELAX(?Y ub:subOrganizationOf* ?Z) .
APPROX(?Z ub:affiliatedOrganizationOf ub:University0) }
```

Experimental Results - 4

	Q_1	Q_2	Q_3	Q_4
D_1	23/0.001s	34/0.004s	0/0.197s	331/0.129s
D_2	63/0.006s	34/0.005s	0/0.64s	883/0.296s
D_3	100/0.007s	34/0.006s	0/1.67s	1381/0.517s
D_1/AR	840/0.015s	605/0.421s	743/0.924s	348/60.7s
D_2/AR	2326/0.055s	605/1s	756/3.17s	925/451s
D_3/AR	3706/0.105s	605/1.6s	767/4.44s	1461/1492s

Q_4:

```
SELECT * WHERE { RELAX(?X ub:advisor/ub:teacherOf ?Z) .
APPROX(?X ub:takesCourse ?Z)}
```

Conclusions

- We have extended a fragment of SPARQL 1.1 with approximation and relaxation operators, showing that this does not increase the complexity of query answering, and that such operators can be useful in finding more answers than would be returned by the exact form of a user's query.
- An advantage of our query rewriting approach is that existing techniques for SPARQL query optimisation and evaluation can be reused.
- Even though the initial experimental study is promising, we are investigating optimizing the rewriting algorithm since it can generate a large number of queries.
- We are also investigating optimization techniques to deal with queries such as Q_4 which cannot be evaluated efficiently using a naive approach.

Future Work - Query Containment

Intuition

Given a query $Q = Q_1$ AND Q_2, if it is the case that $[\![Q_1]\!]_G \subseteq [\![Q_2]\!]_G$ for every G, then $[\![Q]\!]_G = [\![Q_1]\!]_G$.

Through this intuition we can reduce the number of queries generated by the Rewriting Algorithm.

Issues

- Query containment incorporating query costs.
- Query containment for regular path queries, and conjunctive regular path queries, incorporating APPROX and RELAX.

Future Work - Similarity Matching and FLEX

Smilarity Matching

- Exploits the FILTER operator of SPARQL binding a variable to a set of similar strings.
- e.g. FILTER $sim(?x, \text{“}cat\text{”})$

- String similarity e.g. “color” similar to “Colour”.
- Semantic similarity e.g. “Cat” similar to “Kitten”.
  - ⋆ Uses RDF dictionaries: WordNet.
  - ⋆ Can be extended to full English sentences.

We are also investigating a new FLEX operator which combines all the aforementioned flexible querying operators, including the Similarity Matching.

Questions?