TECHNISCHE UNIVERSITÄT DRESDEN

International Center for Computational Logic

KNOWLEDGE GRAPHS

Lecture 8: Limits of SPARQL

Markus Krötzsch

Knowledge-Based Systems

TU Dresden, 19 Dec 2024

Review

SPARQL is:

- PSpace-complete for combined and query complexity
- NL-complete for data complexity

⇝ scalable in the size of RDF graphs, not really in the size of query

⇝ similar situation to other query languages

Hardness is shown by reducing from known hard problems

- Truth of quantified boolean formulae (QBF)
- Reachability in a directed graph

Membership is shown by (sketching) appropriate algorithms

- Naive, iteration-based solution finding procedure runs in polynomial space
- For fixed queries, the complexity drop to nondeterministic logspace

Expressive power

Expressive power and the limits of SPARQL

The expressive power of a query language is described by the question:
"Which sets of RDF graphs can I distinguish using a query of that language?"

More formally:

- Every query defines a set of RDF graphs: the set of graphs that it returns at least one result for
- However, not every set of RDF graphs corresponds to a query (exercise: why?)

Note: Whether a query has any results at all is not what we usually ask for, but it helps us here to create a simpler classification. One could also compare query results over a graph and obtain similar insights overall.

Definition 8.1: We say that a query language $\mathbf{Q}_1$ is more expressive than another query language $\mathbf{Q}_2$ if it can characterise strictly more sets of graphs.

Complexity limits expressivity

Intuition: The lower the complexity of query answering, the lower its expressivity.

Complexity limits expressivity

Intuition: The lower the complexity of query answering, the lower its expressivity.

Question: which complexity are we talking about here?

Complexity limits expressivity

Intuition: The lower the complexity of query answering, the lower its expressivity.

Question: which complexity are we talking about here? — data complexity!

- Given a set of RDF graphs that we would like to classify,
- we ask if there is one (fixed) query that accomplishes this.

If classifying the set of graphs encodes a computationally difficult problem, then the query evaluation has to be at least as hard as this problem with respect to data complexity.

Complexity limits expressivity

Intuition: The lower the complexity of query answering, the lower its expressivity.

Question: which complexity are we talking about here? — data complexity!

- Given a set of RDF graphs that we would like to classify,
- we ask if there is one (fixed) query that accomplishes this.

If classifying the set of graphs encodes a computationally difficult problem, then the query evaluation has to be at least as hard as this problem with respect to data complexity.

Example 8.2: We have argued that SPARQL queries can evaluate QBF, and we could encode QBF in RDF graphs (in many reasonable ways). However, there cannot be a SPARQL query that recognises all RDF graphs that encode a true QBF, since this problem is PSpace-complete, which is known to be not in NL.

Complexity is not the same as expressivity

Complexity-based arguments are often quite limited:

- They only apply to significantly harder problems
- Additional assumptions are often needed (e.g., it is assumed that NL $\neq$ NP, but it was not proven yet)
- Typically, query languages cannot even solve all problems in their own complexity class (i.e., they do not “capture” this class)

$\leadsto$ Direct arguments for non-expressivity need to be sought.

Example: Complexity $\neq$ expressivity

The problem of parallel reachability is defined as follows:

Given: An RDF graph G; two vertices s and t; and two RDF properties p and q

Question: Is there a directed path from s to t, where each two neighbouring nodes on the path are connected by both a p-edge and a q-edge?

Example: Complexity $\neq$ expressivity

The problem of parallel reachability is defined as follows:

Given: An RDF graph G; two vertices s and t; and two RDF properties p and q

Question: Is there a directed path from s to t, where each two neighbouring nodes on the path are connected by both a p-edge and a q-edge?

Proposition 8.3: Parallel reachability is in NL.

Example: Complexity $\neq$ expressivity

The problem of parallel reachability is defined as follows:

Given: An RDF graph G; two vertices s and t; and two RDF properties p and q

Question: Is there a directed path from s to t, where each two neighbouring nodes on the path are connected by both a p-edge and a q-edge?

Proposition 8.3: Parallel reachability is in NL.

Proof: The check can be done using a similar algorithm as for s-t-reachability, merely checking for two edges in each step. □

Example: Complexity $\neq$ expressivity

The problem of parallel reachability is defined as follows:
Given: An RDF graph G; two vertices s and t; and two RDF properties p and q
Question: Is there a directed path from s to t, where each two neighbouring nodes on the path are connected by both a p-edge and a q-edge?

Proposition 8.3: Parallel reachability is in NL.

Proof: The check can be done using a similar algorithm as for s-t-reachability, merely checking for two edges in each step. □

Proposition 8.4: SPARQL cannot express parallel reachability.

Example: Complexity $\neq$ expressivity

The problem of parallel reachability is defined as follows:
Given: An RDF graph G; two vertices s and t; and two RDF properties p and q
Question: Is there a directed path from s to t, where each two neighbouring nodes on the path are connected by both a p-edge and a q-edge?

Proposition 8.3: Parallel reachability is in NL.

Proof: The check can be done using a similar algorithm as for s-t-reachability, merely checking for two edges in each step. □

Proposition 8.4: SPARQL cannot express parallel reachability.

Proof: The only SPARQL feature that can check for paths are property path patterns, but:

- a match to a property path pattern is always possible using only vertices of degree 2 on the path; higher degrees can only be enforced for a limited number of nodes that are matched to query variables
- the query requires an arbitrary number of nodes of degree 4 on the path □

Other structural limits to SPARQL expressivity

SPARQL's regular recursions is also limited in many other cases:

- Non-regular path languages cannot be expressed
- "Wide" paths consisting of repeated graph patterns cannot be expressed
- Tree-like patterns and other non-linear patterns cannot be expressed
- "Nested regular expressions" with tests cannot be expressed

Limits by design

Besides mere expressivity, SPARQL also has some fundamental limits since it simply has no support for some query or analysis tasks:

- SPARQL is lacking some dataypes and matching filter conditions, most notably geographic coordinates (major RDF databases add this)
- SPARQL cannot talk about path lengths, e.g., one cannot retrieve the length of the shortest connecting path between two elements
- SPARQL cannot return paths (of a priori unknown length) in results
- SPARQL has no support for recursive/iterative computation, e.g., for Page Rank or other graph algorithms

Potential reasons: performance concerns (e.g., Page Rank computation would mostly take too long; longest path detection is NP-complete [in data complexity!]), historic coincidence (geo coordinates not in XML Schema datatypes); design issues (handling paths in query results would require many different constructs)

SPARQL: Outlook

A number of SPARQL features have not been discussed:

- Graphs: SPARQL supports querying RDF datasets with multiple graphs, and queries can retrieve graph names as variable bindings
- Updates: SPARQL has a set of features for inserting and deleting data

Example 8.5: The following query replaces all uses of the `hasSister` property with a different encoding of the same information:

```
DELETE { ?person eg:hasSister ?sister }
INSERT {
  ?person eg:hasSibling ?sister .
  ?sister eg:sex eg:female .
}
WHERE { ?person eg:hasSister ?sister }
```

- Result formats: SPARQL has several encodings for sending results, and it can also encode results as RDF graphs (**CONSTRUCT**).
- Federated queries: SPARQL can get sub-query results from other SPARQL services

Datalog

A rule-based query language

Datalog has been introduced as a rule-based query language in (relational) databases

Example 8.6: The following set of rules describes a query for all ancestors of the individual Alice from the given binary relations father and mother, where we assume that the predicate Result denotes the output relation:

$$\begin{aligned}
\text{Parent}(x, y) &:- \text{father}(x, y) \\
\text{Parent}(x, y) &:- \text{mother}(x, y) \\
\text{Ancestor}(x, y) &:- \text{Parent}(x, y) \\
\text{Ancestor}(x, z) &:- \text{Parent}(x, y), \text{Ancestor}(y, z) \\
\text{Result}(y) &:- \text{Ancestor}(\text{alice}, y)
\end{aligned}$$

Rules have their consequence on the left and preconditions on the right, so we can read :– as "if" and , as "and".

It is not hard to apply this approach to graphs.

Datalog Syntax

To define Datalog, we recall some basic definitions of (predicate) logic:

- We use three mutually disjoint (infinite) sets of constants, variables, and predicate symbols.
  Every predicate symbol has a fixed arity (a natural number ≥ 0).
- A term is a constant or a variable.
- An atom is a formula of the form $R(t_1, \ldots, t_n)$ with R a predicate symbol of arity n, and $t_1, \ldots, t_n$ terms.

Definition 8.7: A Datalog rule is an expression of the form:

$$H \text{ :- } B_1, \ldots, B_m$$

where H and $B_1, \ldots, B_m$ are atoms. H is called the head or conclusion; $B_1, \ldots, B_m$ is called the body or premise. A ground rule is one without variables (i.e., all terms are constants). A ground rule with empty body ($m = 0$) is called a fact.

A set of Datalog rules is a Datalog program. A Datalog query is a Datalog program together with a distinguished query predicate.

Datalog Semantics by FO Entailment

A Datalog query is evaluated on a given database, which we can view as a set of facts.

Example 8.8: If the database table for mother is given by

alice	barbara
barbara	christine
dave	emmy

then this data is represented by the facts mother(alice, barbara), mother(barbara, christine), and mother(dave, emmy).

We can then define Datalog query results based on the usual semantics of first-order logic:

Definition 8.9: The result of a Datalog query $\langle P, \text{Result}\rangle$ over a database D is the set of all facts over Result that are logically entailed by $D \cup P$ when reading Datalog rules as first-order logic implications (from right to left).

Note: Datalog semantics is set-based (no multiplicity of results).

Datalog Semantics by FO Entailment: Example

Database D:

$$(1) \quad \text{father}(\text{alice}, \text{bob})$$

$$(2) \quad \text{mother}(\text{alice}, \text{carla})$$

$$(3) \quad \text{mother}(\text{evan}, \text{carla})$$

$$(4) \quad \text{father}(\text{carla}, \text{david})$$

Datalog rules P:

$$(5) \quad \text{Parent}(x, y) :- \text{father}(x, y)$$

$$(6) \quad \text{Parent}(x, y) :- \text{mother}(x, y)$$

$$(7) \quad \text{Ancestor}(x, y) :- \text{Parent}(x, y)$$

$$(8) \quad \text{Ancestor}(x, z) :- \text{Parent}(x, y), \text{Ancestor}(y, z)$$

An example of a first-order logical consequence of P and D is the fact Ancestor(alice, david).

Datalog semantics: Applying rules

A more practical definition of semantics is based on "applying" rules.

To do this, we need to instantiate rules by replacing variables with specific constants:

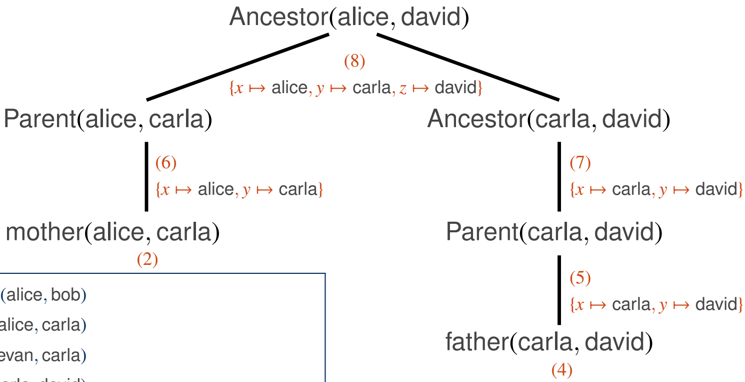
Definition 8.10: A ground substitution σ is a mapping from variables to constants. Given an atom A, we write $A\sigma$ for the atom obtained by simultaneously replacing all variables x in A with $\sigma(x)$.

Notation 1: To write substitutions, we give the (finitely many) relevant mappings directly. Example: $\sigma = \{x \mapsto \text{alice}, y \mapsto \text{david}\}$ is the substitution with $\sigma(x) = \text{alice}$ and $\sigma(y) = \text{david}$.

Notation 2: Substitutions are often written after the expression that they are applied to, e.g., $x\sigma = \sigma(x)$.

Datalog Deductions as Proof Trees

The rule applications used to derive a fact can be visualised as a proof tree:



(1) father(alice, bob)

(2) mother(alice, carla)

(3) mother(evan, carla)

(4) father(carla, david)

(5) Parent(x, y) :– father(x, y)

(6) Parent(x, y) :– mother(x, y)

(7) Ancestor(x, y) :– Parent(x, y)

(8) Ancestor(x, z) :– Parent(x, y), Ancestor(y, z)

Datalog semantics: The consequence operator

Definition 8.11: The immediate consequence operator T_P maps sets of ground facts I to sets of ground facts $T_P(I)$:

$$T_P(I) = \{H\sigma \mid \text{there is some } H \text{ :- } B_1, \ldots, B_n \in P \text{ such that } B_1\sigma, \ldots, B_n\sigma \in I \text{ for some ground substitution } \sigma\}$$

Given a database D, we can define a sequence of databases D^i as follows:

$$D_P^1 = D \qquad D_P^{i+1} = D \cup T_P(D_P^i) \qquad D_P^\infty = \bigcup_{i\geq 0} D_P^i$$

Datalog semantics: The consequence operator

Definition 8.11: The immediate consequence operator T_P maps sets of ground facts I to sets of ground facts $T_P(I)$:

$$T_P(I) = \{H\sigma \mid \text{there is some } H :- B_1, \ldots, B_n \in P \text{ such that } B_1\sigma, \ldots, B_n\sigma \in I \text{ for some ground substitution } \sigma\}$$

Given a database D, we can define a sequence of databases D^i as follows:

$$D_P^1 = D \qquad D_P^{i+1} = D \cup T_P(D_P^i) \qquad D_P^\infty = \bigcup_{i\geq 0} D_P^i$$

Observations:

- We obtain an increasing sequence $D_P^1 \subseteq D_P^2 \subseteq D_P^3 \subseteq \ldots \subseteq D_P^\infty$ (why?)
- Ground atom A is entailed by $P \cup D$ if and only if $A \in D_P^\infty$.
- Only a finite number of ground facts can ever be derived from $D \cup P$.
- Hence the sequence $D_P^1, D_P^2, \ldots$ is finite and there is some $k \geq 1$ with $D_P^k = D_P^\infty$.

The consequence operator: Example

$$
\begin{aligned}
(1) &\quad \text{father}(\text{alice}, \text{bob}) \\
(2) &\quad \text{mother}(\text{alice}, \text{carla}) \\
(3) &\quad \text{mother}(\text{evan}, \text{carla}) \\
(4) &\quad \text{father}(\text{carla}, \text{david}) \\
(5) &\quad \text{Parent}(x, y) \text{ :- } \text{father}(x, y) \\
(6) &\quad \text{Parent}(x, y) \text{ :- } \text{mother}(x, y) \\
(7) &\quad \text{Ancestor}(x, y) \text{ :- } \text{Parent}(x, y) \\
(8) &\quad \text{Ancestor}(x, z) \text{ :- } \text{Parent}(x, y), \text{Ancestor}(y, z)
\end{aligned}
$$

$$
\begin{aligned}
D_P^1 &= \{\text{father}(\text{alice}, \text{bob}), \text{mother}(\text{alice}, \text{carla}), \text{mother}(\text{evan}, \text{carla}), \text{father}(\text{carla}, \text{david})\} \\
D_P^2 &= D_P^1 \cup \{\text{Parent}(\text{alice}, \text{bob}), \text{Parent}(\text{alice}, \text{carla}), \text{Parent}(\text{evan}, \text{carla}), \text{Parent}(\text{carla}, \text{david})\} \\
D_P^3 &= D_P^2 \cup \{\text{Ancestor}(\text{alice}, \text{bob}), \text{Ancestor}(\text{alice}, \text{carla}), \text{Ancestor}(\text{evan}, \text{carla}), \\
&\qquad\qquad \text{Ancestor}(\text{carla}, \text{david})\} \\
D_P^4 &= D_P^3 \cup \{\text{Ancestor}(\text{alice}, \text{david}), \text{Ancestor}(\text{evan}, \text{david})\} \\
D_P^5 &= D_P^5 = D_P^\infty
\end{aligned}
$$

Using Datalog on RDF

Datalog assumes that databases are given as sets of (relational) facts.

How to apply Datalog to graph data?

Using Datalog on RDF

Datalog assumes that databases are given as sets of (relational) facts.

How to apply Datalog to graph data?

Option 1: Properties as binary predicates

- An RDF triple $s\ p\ o$ can be represented by a fact $p(s, o)$
- Both predicate names and constants are IRIs
- Datalog "sees" no relation between properties (predicates) and IRIs in subject and object positions

Option 2: Triples as ternary hyperedges

- An RDF triple $s\ p\ o$ can be represented by a fact $\text{triple}(s, p, o)$
- triple is the only predicate needed to represent arbitrary databases
- IRIs on any triple position can be related in Datalog

Queries beyond SPARQL

Datalog can express many queries that are not expressible in SPARQL.

Example 8.12: The following query expresses parallel s-t-reachability for predicates p and q (for triple encoding):

$$\mathsf{Reach}(x, y) :- \mathsf{triple}(x, p, y), \mathsf{triple}(x, q, y)$$
$$\mathsf{Reach}(x, z) :- \mathsf{Reach}(x, y), \mathsf{Reach}(y, z)$$
$$\mathsf{Result}() :- \mathsf{Reach}(s, t)$$

Note the use of a nullary result predicate: this is a boolean query.

Many other forms of recursion are possible:

- Non-regular (context-free) patterns
- Non-linear (e.g., tree-shaped) patterns
- Recursive pattern definitions (e.g., reachability along path of elements that can reach a specific element via some relation)

Datalog complexity

Fact 8.13: Datalog query answering is

- ExpTime-complete in combined and query complexity
- P-complete in data complexity

See course "Database Theory" for details and proofs.

Datalog complexity

Fact 8.13: Datalog query answering is

- ExpTime-complete in combined and query complexity
- P-complete in data complexity

See course "Database Theory" for details and proofs.

As with SPARQL "P in data" does not imply that all P-computable problems can be solved with a Datalog query.

Example 8.14: Datalog is monotonic: the more input facts given, the more results derived. Clearly, there are P problems that are not monotonic, e.g., "Check if there is an even number of triples in the database."

Summary

SPARQL expressivity is still limited, partly by design

Datalog is a rule-based query language that can express more powerful recursive queries

What's next?

- Datalog: comparision to SPARQL, extensions, practical use
- Property graph
- The Cypher query language