TECHNISCHE UNIVERSITÄT DRESDEN

International Center for Computational Logic

KNOWLEDGE GRAPHS

Lecture 9: Rules for Querying Graphs

Markus Krötzsch

Knowledge-Based Systems

TU Dresden, 9 Jan 2025

Review: Datalog

A rule-based query language

- **Syntax:** Rules based on first-order atoms based on terms (constants or variables) and predicate symbols
- **Semantics:** Logical semantics based on first-order logic entailment from a database viewed as a set of facts; therefore set-based

Example: Recursively finding all ancestors of Alice:

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

Negation

Negation

Negation enables us to ask for the absence of some data or inference.

Example 9.1: SPARQL supports negation in the form of the **NOT EXISTS** filter:

```
SELECT ?person WHERE {
   ?person wdt:P19 wd:Q1731 . # born in Dresden
   FILTER NOT EXISTS { ?person wdt:P570 ?date } # no date of death
}
```

To achieve such expressivity in Datalog, we can add a form of logical negation.

Example 9.2: Using negation, a query for living people born in Dresden could be expressed as follows:

$$\begin{aligned}\text{HasDied}(x) &\mathrel{:-} \text{triple}(x, \mathtt{wdt{:}P570}, y)\\ \text{Result}(x) &\mathrel{:-} \text{triple}(x, \mathtt{wdt{:}P19}, \mathtt{wd{:}Q1731}), \neg\text{HasDied}(x)\end{aligned}$$

Semantics of negation

A negated ground atom $\neg A$ is true over a database D if $A \notin D$. So we can define:

$$\begin{aligned} T_P(I) = \{H\sigma \mid {} & H \coloneq B_1, \ldots, B_n, \neg A_1, \ldots, \neg A_m \in P, \\ & B_1\sigma, \ldots, B_n\sigma \in I, \text{ and } A_1\sigma, \ldots, A_m\sigma \notin I \\ & \text{for some ground substitution } \sigma\} \end{aligned}$$

We could then use this step-wise consequence operator to compute conclusions as before . . .

. . . but there are some problems with that.

Semantics of negation: Unsafe variables

What is the meaning of the following rule?

$$\text{Result}(x) \coloneq \text{triple}(x, \texttt{wdt:P19}, \texttt{wd:Q1731}), \neg\text{triple}(x, \texttt{wdt:P570}, y)$$

According to our definition of T_P: "Find all x, such that x is born in Dresden and there is a date y, such that x did not die on y." (All variables, including y, are universally quantified over the whole rule.)

Many systems do not allow this at all: they require that all variables in negated atoms are safe, i.e., occur in non-negated atoms as well.

Some systems allow unsafe variables but assume that their universal quantifier is below the negation: "Find all x, such that x is born in Dresden and for all dates y, we find that x did not die on y." (example systems: Clingo, Nemo)

Definition 9.3: A rule is safe if all of its variables occur in non-negated atoms in its body.

Requiring all rules to be safe does not restrict expressivity (exercise).

Semantics of negation: Recursion

Even if rules are safe, the unrestricted use of negation in recursive queries leads to semantic problems:

Example 9.4: Consider the following facts and query:

$$\text{human}(\mathit{greta})$$

$$\text{Adult}(x) \coloneq \text{human}(x), \neg\text{Child}(x)$$

$$\text{Child}(x) \coloneq \text{human}(x), \neg\text{Adult}(x)$$

What should be the result if Child were the query predicate?

If we define the sequence D_P^i as before, we obtain:

- $D_P^1 = D = \{\text{human}(\mathit{greta})\}$
- $D_P^2 = D \cup T_P(D_P^1) = D \cup \{\text{Adult}(\mathit{greta}), \text{Child}(\mathit{greta})\}$
- $D_P^3 = D \cup T_P(D_P^2) = D_P^1$
- $D_P^4 = D \cup T_P(D_P^3) = D_P^2 = D_P^\infty$

⇝ non-monotonic behaviour leads to unfounded conclusions (e.g., that all humans are both adults and children)

Stratified negation

Observation: Iterative evaluation of rules fails if negation is freely used in recursion

- Initially, when no facts were derived, many negated atoms are true
- However, these initially true atoms can become false when more inferences are computed

To avoid recursion through negation, one can try to organise rules in “layers” or “strata”:

Definition 9.5: Let P be a set of rules with negation. A function ℓ that assigns a natural number $\ell(p)$ to every predicate p is a stratification of P if the following are true for every rule $h(\mathbf{t}) \coloneq p_1(\mathbf{s_1}), \ldots, p_n(\mathbf{s_n}), \neg q_1(\mathbf{r_1}), \ldots, \neg q_m(\mathbf{r_m}) \in P$:

1. $\ell(h) \geq \ell(p_i)$ for all $i \in \{1, \ldots, n\}$
2. $\ell(h) > \ell(q_i)$ for all $i \in \{1, \ldots, m\}$

Intuition: The function s defines the “level” of the rule. By applying rules exhaustively level-by-level, we can avoid non-monotonic behaviour.

Evaluating stratified rules

Evaluation of stratified programs: Let D be a database and let P be a program with stratification ℓ, with values of ℓ ranging from 1 to h (without loss of generality).

- For $i \in \{1, \ldots, h\}$, we define sub-programs for each stratum:
  $P_i = \{h(\mathbf{t}) \coloneq p_1(\mathbf{s_1}), \ldots, p_n(\mathbf{s_n}), \neg q_1(\mathbf{r_1}), \ldots, \neg q_m(\mathbf{r_m}) \in P \mid \ell(h) = i\}$
- Define $D_0^\infty = D$
- Now for $i = 1, \ldots, h$, we define:
  - $D_i^1 = D_{i-1}^\infty$
  - $D_i^{j+1} = D_{i-1}^\infty \cup T_{P_i}(D_i^j)$
  - $D_i^\infty = \bigcup_{j \geq 1} D_i^j$ is the limit of this process
- The evaluation of P over D is D_h^∞.

Observations:

- For every i, the sequence $D_i^1 \subseteq D_i^2 \subseteq \ldots$ is increasing, since facts relevant for negated body literals are not produced in any D_i^j (due to stratification)
- Such increasing sequences must be finite (since the set of all possible facts is finite)

$\leadsto$ The limits D_i^∞ are computed after finitely many steps

The perfect model

Summary: The stratified evaluation of rules terminates after finitely many steps (bounded by the number of possible facts)

What is the set of facts that we obtain from this procedure?

Fact 9.6: For a database D and stratified program P, the set of facts M that is obtained by the stratified evaluation procedure is the least set of facts with the property that

$$M = D \cup T_P(M).$$

In particular, M does not depend on the stratification that was chosen.

M is called perfect model or unique stable model in logic programming.

Intuition: The stratified evaluation is the smallest set of self-supporting true facts that can be derived

- This is not the set of inferences under classical logical semantics! (exercise)
- But it is a good extension of negation in queries to the recursive setting.

Obtaining a stratification

To find a stratification, the following algorithm can be used:

Input: program P

- Construct a directed graph with two types of edges, $\xrightarrow{+}$ and $\xrightarrow{-}$:
  - The vertices are the predicate symbols in P
  - $p \xrightarrow{+} q$ if there is a rule with p in its non-negated body and q in the head
  - $p \xrightarrow{-} q$ if there is a rule with p in its negated body and q in the head
- Then P is stratified if and only if the graph contains no directed cycle that involves an edge $\xrightarrow{-}$
- In this case, we can obtain a stratification as follows:
  (1) produce a topological order of the strongly connected components of this directed graph (without distinguishing edge types), e.g., using Tarjan's algorithm
  (2) assign numerical strata bottom-up to all predicates in each component

Outlook: Beyond stratified negation

Stratified negation is usually sufficient for query answering.
Non-stratified negation is relevant in optimisation and constraint solving.

Handling non-stratified negation:

- Recursion through negation gives rise to multiple alternative interpretations
- Semantics can be defined in many ways, e.g., stable models (answer set programming), well-founded semantics, and classical semantics
- See various other courses (e.g., “Advanced Problem Solving and Search”)

Stratified negation allows us to express non-monotonic queries.
However, not all polynomial-time queries are expressible.

Capturing PTime:

- To express all polytime queries, in addition to stratified negation, Datalog needs a total order on the domain (defined by special predicates)
- See course “Database Theory” for details

Datalog in Practice

Datalog vs. SPARQL: Supported features

Datalog with stratified negation captures and extends important parts of SPARQL:

- Basic Graph Patterns: are simply conjunctions of triple-atoms
- Path expressions: Datalog does not support paths syntactically, but they can be captured in Datalog
- Union: disjunction can be expressed in Datalog using several rules (exercise)
- Minus and Not Exists: can be expressed with stratified negation in Datalog
- Values: can be declared by Datalog facts

Recall: Datalog always assumes set semantics (Distinct in SPARQL)

Example 9.7: The following rules are an alternative to express the property path pattern `eg:JSBach (^eg:hasFather|^eg:hasMother)+ ?x`:

$$\text{Result}(x) \coloneq \text{triple}(x, \text{eg:hasFather}, \text{eg:JSBach})$$

$$\text{Result}(x) \coloneq \text{triple}(x, \text{eg:hasMother}, \text{eg:JSBach})$$

$$\text{Result}(x) \coloneq \text{Result}(y), \text{triple}(x, \text{eg:hasFather}, y)$$

$$\text{Result}(x) \coloneq \text{Result}(y), \text{triple}(x, \text{eg:hasMother}, y)$$

Datalog vs. SPARQL: Missing features

Many other SPARQL features are not part of plain Datalog:

- Filters: filter conditions (and datatypes) are not part of the pure logical definition of Datalog
- Bind: computed functions are not part of plain Datalog
- Optional: Datalog (and logic in general) does not have a direct way to handle partial result mappings, and there is no equivalent to Optional
- Aggregates: Datalog does not support aggregates, as they introduce non-monotonic behaviour in general
- Subqueries: Datalog cannot express nested limit/offset/order by

$\leadsto$ practical implementations often add such features
(leading to many custom extensions of Datalog)

Implementations of Datalog

Many implementations of Datalog exist:

- In-Memory systems for query answering and data analysis: Graal, Nemo, RDFox, Soufflé, Vadalog, Rulewerk/VLog, . . .
- Answer set programming engines: Clingo, DLV(2), . . .
- Logic programming engines: Prolog implementations
- Data management frameworks: Datomix, Google Logica, . . .
- “Business Rule” engines with database backend support

$\leadsto$ many use cases; many different implementation approaches

Compatibility with knowledge graph formats:

- Rules can support RDF and related technologies (IRI, datatypes)
- Most common for in-memory systems: Graal, Nemo, RDFox, and Rulewerk support RDF
- Nemo and RDFox also support SPARQL filters and aggregates

Rules in Nemo

Nemo is a free rule engine that supports extensions of Datalog:

- Download, documentation, and source code online: `https://github.com/knowsys/nemo`
- Command-line client, programming APIs (Rust, Python, JavaScript), and as browser-based application: `https://tools.iccl.inf.tu-dresden.de/nemo/`
- Support for evaluating Datalog queries over RDF files and SPARQL query results
- Stratified negation and many additional features

Example 9.8: Nemo uses a textual syntax for rules, which is slightly different from the one we used so far. Variables are marked by ?, negation is written as ~, and rules end with a full stop:

```
  Parent(?x,?y) :- father(?x,?y) .
  Parent(?x,?y) :- mother(?x,?y) .
Ancestor(?x,?y) :- Parent(?x,?y) .
Ancestor(?x,?z) :- Parent(?x,?y), Ancestor(?y,?z) .
    Result(?y) :- Ancestor(alice,?y), ~profession(?y,composer) .
```

Filters and value assignments in Nemo

Nemo supports many SPARQL filters and functions as built-in predicates and built-in functions.

Example 9.9: Many expressions use the same names as in SPARQL. Common operators like + can also be written as usual. Instead of BIND, equality = can be used in rule bodies (logically, there is no distinction between assignment and comparison, so there is no ==).

```
@prefix rdfs: <http://www.w3.org/2000/01/rdf-schema#> .

label(?s,STR(?label)) :- triple(?s, rdfs:label, ?o), LANG(?o)="en" .
    shortLabel(?s,?l) :- label(?s,?l), STRLEN(?l) < 3 .

hasArea(?x,?A) :- rectangle(?x), width(?x,?W), length(?x,?L),
                  ?A = ?W * ?L .
```

Aggregates in Nemo

Nemo supports aggregate functions such as COUNT. Aggregates are used in rule heads and marked by a leading #.

Example 9.10: Aggregates are grouped by all variables that appear in the rule head outside of aggregate functions.

```
label(?givenName,#count(?id)) :- person(?id,?givenName,?familyName) .
```

This would correspond to the SPARQL query:

```
SELECT ?givenName (COUNT(DISTINCT ?id) AS ?count)
WHERE {
 ?id <http://example.org/givenName> ?givenName .
 ?id <http://example.org/familyName> ?familyName .
}
GROUP BY ?givenName
```

DISTINCT is implied by the set semantics of Datalog. Expressions like `#count(?id,?familyName)` provide control over what is projected away before building the set.

Other supported aggregates in Nemo include `#min`, `#max`, and `#sum`.

Aggregates require stratification

Like negation, aggregates can capture non-monotonic conditions.

Example 9.11: Counting can be directly used to express negation:

```
countDDate(?person,#count(?date)) :- dateOfDeath(?person,?date) .
alive(?person) :- countDDate(?person, 0) .
```

To prevent related issues, Nemo requires aggregation to be stratified:

- Strata can be assigned to predicates as before (based on the dependency graph)
- Rules with aggregates in heads must only use body predicates from lower strata

Most datalog tools likewise require stratification of aggregates (e.g., Soufflé).

Answer set programming tools allow for unstratified aggregates (under somewhat different semantics).

RDF and SPARQL in Nemo

Facts in Nemo can be loaded from RDF files or from CSV files returns from SPARQL services.

Example 9.12: The following lines load data into two predicates:

```
@import triple :- rdf{resource="file.nt"} .

@import awards :- tsv{ resource=f"{?endpoint}?query={URIENCODE(?q)}",
                       ignore_headers=true},
                  ?endpoint = "https://query.wikidata.org/sparql",
                  ?q = """
 SELECT ?award ?awardLabel WHERE {
   wd:Q42 wdt:P166 ?award .
   ?award rdfs:label ?awardLabel FILTER(LANG(?awardLabel) = "en")
 }
""" .
```

Note: The query works without prefix declarations since the Wikidata endpoint has default prefixes.

Nemo

Graph Rule Engine

Full documentation is avaiable online:
`https://knowsys.github.io/nemo-doc/`

Summary

Stratified negation is a simple way of adding negation to recursive queries

Datalog can capture and extend many basic features of SPARQL, and can be extended to include many datatypes, aggregates, and filters

SPARQL features that are uncommon in Datalog are optional, ordering, and multiset semantics

Nemo is a free RDF-compatible rule engine

What's next?

- Further KG query languages
- Knowledge Graph quality
- Schemas for knowledge graphs