TECHNISCHE UNIVERSITÄT DRESDEN

KNOWLEDGE GRAPHS

Lecture 9: Rules for Querying Graphs

Markus Krötzsch

Knowledge-Based Systems

TU Dresden, 10th Dec 2019

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:

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

Datalog semantics revisited

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

Definition 9.1: 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 $\sigma(x)$.

Datalog semantics revisited

A more practical definition of semantics is based on “applying” rules:

Definition 9.1: 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 $\sigma(x)$.

Definition 9.2: The immediate consequence operator T_P maps sets of ground facts I to sets of ground facts $T_P(I)$:

$$T_P(I) = \{H\sigma \mid H :\!- B_1, \ldots, B_n \in P \text{ and } B_1\sigma, \ldots, B_n\sigma \in I\}$$

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 revisited

A more practical definition of semantics is based on “applying” rules:

Definition 9.1: 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 $\sigma(x)$.

Definition 9.2: The immediate consequence operator T_P maps sets of ground facts I to sets of ground facts $T_P(I)$:

$$T_P(I) = \{H\sigma \mid H :- B_1, \ldots, B_n \in P \text{ and } B_1\sigma, \ldots, B_n\sigma \in I\}$$

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 $D_P^k = D_P^\infty$.

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

$$\begin{aligned}
\text{Reach}(x, y) &:- \text{triple}(x, p, y), \text{triple}(x, q, y) \\
\text{Reach}(x, z) &:- \text{Reach}(x, y), \text{Reach}(y, z) \\
\text{Result}() &:- \text{Reach}(s, t)
\end{aligned}$$

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 9.4: 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 9.4: 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 9.5: 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."

Negation

Negation

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

Example 9.6: 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 9.7: Using negation, a query for living people born in Dresden could be expressed as follows:

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

Semantics of negation (1)

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

$$\begin{aligned} T_P(I) = \{H\sigma \mid {} & H \mathrel{:-} B_1, \ldots, B_n, \neg A_1, \ldots, \neg A_m \in P, \\ & B_1\sigma, \ldots, B_n\sigma \in I, \text{ and } A_1\sigma, \ldots, A_m\sigma \notin I\} \end{aligned}$$

Semantics of negation (1)

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

$$T_P(I) = \{H\sigma \mid H \coloneq B_1, \ldots, B_n, \neg A_1, \ldots, \neg A_m \in P, \\ B_1\sigma, \ldots, B_n\sigma \in I, \text{ and } A_1\sigma, \ldots, A_m\sigma \notin I\}$$

Example 9.8: What is the meaning of the following rule?

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

Semantics of negation (1)

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

$$T_P(I) = \{H\sigma \mid H \coloneq B_1, \ldots, B_n, \neg A_1, \ldots, \neg A_m \in P, \\ B_1\sigma, \ldots, B_n\sigma \in I, \text{ and } A_1\sigma, \ldots, A_m\sigma \notin I\}$$

Example 9.8: What is the meaning of the following rule?

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

"Find all x, such that x is born in Dresden and there is a date y, such that x did not die on y."

Semantics of negation (1)

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

$$T_P(I) = \{H\sigma \mid H \coloneq\!- B_1, \ldots, B_n, \neg A_1, \ldots, \neg A_m \in P,$$
$$B_1\sigma, \ldots, B_n\sigma \in I, \text{ and } A_1\sigma, \ldots, A_m\sigma \notin I\}$$

Example 9.8: What is the meaning of the following rule?

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

"Find all x, such that x is born in Dresden and there is a date y, such that x did not die on y."

Observation: If variables appear only in negated atoms, then it is not clear which values they range over (e.g., which bindings for y should be considered in the rule above?).

Definition 9.9: A rule is safe if all of its variables occur in non-negated atoms in its body.

It is common to require all rules to be safe, and this does not restrict expressivity (exercise).

Semantics of negation (2)

The unrestricted use of negation in recursive queries leads to semantic problems:

Example 9.10: The following facts and query model a stereotypical gender-binary world view:

$$\text{human}(evelyn) \qquad \text{human}(jo)$$

$$\text{Male}(x) \coloneq \text{human}(x), \neg\text{Female}(x)$$

$$\text{Female}(x) \coloneq \text{human}(x), \neg\text{Male}(x)$$

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

Semantics of negation (2)

The unrestricted use of negation in recursive queries leads to semantic problems:

Example 9.10: The following facts and query model a stereotypical gender-binary world view:

$$\text{human}(\mathit{evelyn}) \qquad \text{human}(\mathit{jo})$$

$$\text{Male}(x) \coloneq \text{human}(x), \neg\text{Female}(x)$$

$$\text{Female}(x) \coloneq \text{human}(x), \neg\text{Male}(x)$$

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

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

- $D_P^1 = D = \{\text{human}(\mathit{evelyn}), \text{human}(\mathit{jo})\}$
- $D_P^2 = D \cup T_P(D_P^1) = D \cup \{\text{Male}(\mathit{evelyn}), \text{Female}(\mathit{evelyn}), \text{Male}(\mathit{jo}), \text{Female}(\mathit{jo})\}$
- $D_P^3 = D \cup T_P(D_P^2) = D_P^1$
- $D_P^4 = D \cup T_P(D_P^3) = D_P^2 = D_P^\infty$

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

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

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 9.11: 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:
  $P_i = \{h(\mathbf{t}) \text{ :- } p_1(\mathbf{s_1}), \ldots, p_n(\mathbf{s_n}), \neg q_1(\mathbf{r_1}), \ldots, \neg q_m(\mathbf{r_m}) \in P \mid \ell(h) = i\}$
- 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^∞.

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?

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 9.12: 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.

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., “Problem Solving and Search in AI”)

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., “Problem Solving and Search in AI”)

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

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., “Problem Solving and Search in AI”)

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

Comparing Datalog and SPARQL

Supported SPARQL 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 9.13: The following rules are an alternative to express the property path pattern `eg:JSBach (ˆeg:hasFather|ˆeg:hasMother)+ ?x`:

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

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

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

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

Missing SPARQL features

Many other SPARQL features are not part of Datalog:

- Filters: filter conditions (and datatypes) are not part of the pure logical definition of Datalog, but can easily be added as built-in predicates
- Bind: computed functions are not usually found in Datalog but can be added
- 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; Datalog extensions with restricted aggregation exist
- Subqueries: Datalog cannot express nested limit/offset/order by

Datalog in practice

Implementations of Datalog

Many implementations of Datalog exist:

- In-Memory systems for query answering and data analysis: Graal, RDFox, Vadalog, VLog4j, . . .
- Answer set programming engines: Clingo, DLV(2), . . .
- Logic programming engines: Prolog implementations
- Database-backed (business) rule engines

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

Implementations of Datalog

Many implementations of Datalog exist:

- In-Memory systems for query answering and data analysis: Graal, RDFox, Vadalog, VLog4j, . . .
- Answer set programming engines: Clingo, DLV(2), . . .
- Logic programming engines: Prolog implementations
- Database-backed (business) rule engines

⇝ many use cases; many different implementation approaches

Compatibility with knowledge graph formats:

- Typically support for RDF and related technologies (IRI, datatypes)
- Most common for in-memory systems: Graal, RDFox, and VLog4j support RDF
- VLog4j also supports SPARQL as source of external data

Rules in VLog4j

VLog4j is a free rule engine that supports extensions of Datalog:

- Download, documentation, and source code online:
  `https://github.com/knowsys/vlog4j`
- Java library and command-line client based on the VLog rule engine (C++)
- Support for evaluating Datlog queries over RDF files and SPARQL query results
- Stratified negation and value invention (existential rules)

Example 9.14: VLog4j 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) .
```

RDF and SPARQL in Datalog

Facts in VLog4j can be specified as part of the rules, loaded from files (RDF or CSV), or loaded from SPARQL query services. RDF terms and prefixes can be used.

The following example evaluates data from Wikidata with simple rules:

```
@prefix wdqs: <https://query.wikidata.org/> .
@source mother[2] : sparql(wdqs:sparql, "child,mother",
                                   "?child wdt:P25 ?mother") .
% Rules to find maternal ancestors:
matAnc(?X, ?Y) :- mother(?X, ?Y) .
matAnc(?X, ?Z) :- matAnc(?X, ?Y), matAnc(?Y, ?Z) .
% Query for maternal ancestors of Ada Lovelace
Result(?X) :- matAnc(<http://www.wikidata.org/entity/Q7259>, ?X) .
```

Summary

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

Datalog can capture and extend many basic features of SPARQL

SPARQL features not in Datalog include many datatypes, aggregates, optional, and multiset semantics

VLog4j is a free RDF-compatible rule engine

What's next?

- Property graph
- The Cypher query language
- Knowledge Graph quality