Technische Universität Dresden

International Center for Computational Logic

Knowledge Graphs

Lecture 8: Advanced Features of Datalog

Markus Krötzsch

Knowledge-Based Systems

TU Dresden, 2 Dec 2025

Review: Datalog

Datalog rules P:

$$
\begin{aligned}
\text{parent}(x,y) &\leftarrow \text{father}(x,y)\\
\text{parent}(x,y) &\leftarrow \text{mother}(x,y)\\
\text{ancestor}(x,y) &\leftarrow \text{parent}(x,y)\\
\text{ancestor}(x,z) &\leftarrow \text{ancestor}(x,y), \text{parent}(y,z)\\
\text{commonAnc}(x) &\leftarrow \text{ancestor}(\text{alice},x), \text{ancestor}(\text{finley},x)
\end{aligned}
$$

Input database $\mathcal{D}$:

$$
\begin{aligned}
\text{father}(\text{alice},\text{bob})\\
\text{mother}(\text{alice},\text{cho})\\
\text{mother}(\text{cho},\text{eiko})\\
\text{mother}(\text{finley},\text{eiko})
\end{aligned}
$$

There are four equivalent ways of defining Datalog semantics:

- Operational semantics: least fixed point of consequence operator T_P
- Model-theoretic semantics: entailments of all/least Herbrand/FO model(s)
- Proof-theoretic semantics: every conclusion of some proof tree
- Second-order axiomatisation: Satisfying assignments in SO model checking

RDF data can be represented using a triple predicate (ternary) or dedicated property predicates (binary).

Stratification: Computation in Layers

Negation

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

Example 8.1: Using negation, a query for living people born in Dresden might be expressed as follows:

$$\begin{aligned}\mathsf{hasDied}(x) &\leftarrow \mathsf{dateOfDeath}(x,y)\\ \mathsf{result}(x) &\leftarrow \mathsf{bornIn}(x,\mathsf{dresden}), \neg\mathsf{hasDied}(x)\end{aligned}$$

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

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

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?

$$\text{result}(x) \leftarrow \text{bornIn}(x, \text{dresden}), \neg\text{dateOfDeath}(x, y)$$

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 8.2: 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 (why?).

Semantics of Negation: Recursion

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

Example 8.3: Consider the following facts and query:

$$\begin{aligned} &\text{human}(\text{alice}) \\ \text{underage}(x) &\leftarrow \text{human}(x), \neg\text{adult}(x) \\ \text{adult}(x) &\leftarrow \text{human}(x), \neg\text{underage}(x) \end{aligned}$$

What should be the result if adult and underage were output predicates?

If we define the sequence $\mathcal{D}_P^i$ as before, we obtain:

- $\mathcal{D}_P^0 = \mathcal{D} = \{\text{human}(alice))\}$
- $\mathcal{D}_P^1 = \mathcal{D} \cup T_P(\mathcal{D}_P^0) = \mathcal{D} \cup \{\text{underage}(\text{alice}), \text{adult}(\text{alice})\}$
- $\mathcal{D}_P^2 = \mathcal{D} \cup T_P(\mathcal{D}_P^1) = \mathcal{D}_P^0$
- $\mathcal{D}_P^3 = \mathcal{D} \cup T_P(\mathcal{D}_P^2) = \mathcal{D}_P^1 = \mathcal{D}_P^\infty$

$\leadsto$ non-monotonic behaviour leads to unfounded conclusions

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 8.4: 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 $h(\mathbf{t}) \leftarrow p_1(\mathbf{s_1}), \ldots, p_n(\mathbf{s_n}), \neg q_1(\mathbf{r_1}), \ldots, \neg q_m(\mathbf{r_m}) \in P$:

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 ℓ 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 L (without loss of generality).

- For $i \in \{1, \ldots, L\}$, we define sub-programs for each stratum:
  $P_i = \{h(\mathbf{t}) \leftarrow p_1(\mathbf{s_1}), \ldots, p_n(\mathbf{s_n}), \neg q_1(\mathbf{r_1}), \ldots, \neg q_m(\mathbf{r_m}) \in P \mid \ell(h) = i\}$
- Define $\mathcal{D}_0^\infty = \mathcal{D}$, and define for each $i = 1, \ldots, L$:
  - $\mathcal{D}_i^0 = \mathcal{D}_{i-1}^\infty$
  - $\mathcal{D}_i^{j+1} = \mathcal{D}_{i-1}^\infty \cup T_{P_i}(\mathcal{D}_i^j)$
  - $\mathcal{D}_i^\infty = \bigcup_{j \geq 1} \mathcal{D}_i^j$ is the limit of this process
- The evaluation of P over $\mathcal{D}$ is $\mathcal{D}_L^\infty$.

Observations:

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

$\rightsquigarrow$ The limits $\mathcal{D}_i^\infty$ 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 8.5: For a database $\mathcal{D}$ and stratified program P, the set of facts $\mathcal{M}$ that is obtained by the stratified evaluation procedure is the least set of facts with the property that

$$\mathcal{M} = \mathcal{D} \cup T_P(\mathcal{M}).$$

In particular, $\mathcal{M}$ does not depend on the stratification that was chosen.

$\mathcal{M}$ 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! (why?)
- 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

Discussion: Declarativity of Stratified Negation

How far have we moved away from the semantic elegance of pure Datalog?

- Operational semantics: modified consequence operator based on stratification
- Model-theoretic semantics: entailments based on perfect model
- Proof-theoretic semantics: unclear/unsatisfactory
- Second-order axiomatisation: conceivable, but (even) more technical

Simpler special cases:

- Input negation: Only atoms with EDB predicates may be negated
- Inequality: Support a predicate $\neq$ that expresses non-identity

$\leadsto$ proof trees are more compelling for these cases

Hands-on: Finding last common ancestors

We work with the Wikidata-based common ancestor example as before:
`https://tud.link/wkrajt`

Task: Moby and Ada have a lot of common ancestors. Modify the program to find only the latest (most recent) among them.

Datatypes

Putting the Data into Datalog

Obviously, we want datatypes . . . and related functions/predicates.

Typical datatypes and functions:

- Strings and string functions (concatenation, substring, toUpper, . . .)
- Integers/floats and arithmetic functions
- RDF-specific terms (IRIs, language literals) and related functions
- . . . (calendar dates, geographic coordinates, . . .)
- . . . conversion functions (toString, toInt, round, . . .) and hash functions

$\leadsto$ available as built-in functions and predicates in Datalog systems

Strong Typing vs. Weak Typing

Two alternative ways to handle typed data in computer languages.

Strong typing

- Declare a type for all places where data may occur (typed schema)
- Ensure schema compliance statically (“at compile time”)
- Advantages: type safety, easier to implement
- Typical in: RDBMS, description logics, sorted logics
- In Datalog: Soufflé, Logica, Yedalog, Datomic (?), . . .

Weak typing

- Do not assign types to places (schema remains untyped)
- Handle type mismatches dynamically (“at runtime”)
- Advantages: flexibility, less declaration effort
- Typical in: RDF, JSON, CSV, logic programming (Prolog, ASP)
- In Datalog: Clingo, Nemo, RDFox, N3, . . .

Compromises exists, e.g., OWL ontologies distinguish “data properties” from “object properties” (strong typing), but allow mixed data values for data properties (weak typing)

Datatypes: What could go wrong?

Problem 1: Finding rule matches might be much harder with unrestricted datatype built-ins.

Possible solutions:

- Require safety: variables in built-ins must also occur in normal body atoms
- Implement datatype-related reasoning: constraint logic programming, SAT modulo theories, theorem proving, . . . (but not Datalog)

Problem 2: Inferences may contain unbounded amounts of new (computed) terms, and $\mathcal{D}_P^\infty$ may be infinite.

Possible solutions:

- Accept: let users worry about this (most systems do that)
- Enforce: add extra-logical termination mechanisms
- Analyse: restrict to cases where termination is certain (though undecidable in general)

Complex Value Types

Rule engines may also support structured types (a.k.a. complex values):

- Abstract function terms (such as $f(a, g(b))$)
- Lists, records, and tuples
- Maps and sets
- Graphs
- …

$\leadsto$ uneven support in current systems, mainly lists/tuples/records

Datatype Support in Nemo

Nemo implements weak typing with an RDF-inspired type system and syntax

Datatype support as of Nemo v0.9:

- Integer types and functions (up to signed int 64)
- Floating point types and functions (f32, f64)
- Strings and language-tagged strings and functions, but no lexicographic orderings
- All other datatypes (RDF standard and custom types) faithfully handled, only few functions supported
- Complex value types under development (`https://github.com/knowsys/nemo/issues/699`)

Full documentation: `https://knowsys.github.io/nemo-doc/reference/builtins/`

Aggregates

Aggregation in Datalog

Aggregation functions are functions from a set of tuples (a relation) to a single tuple

Examples:

- `count` computes the cardinality of the input relation
- `sum` computes the sum of the first column in the input relation
- `max` computes the largest value of the first column in the input relation
- `min` computes the least value of the first column in the input relation

$\leadsto$ useful and possible in Datalog

Aggregation Step by Step

Essential inputs needed for aggregation:

- let R be a relation (set of tuples) of arity n,
- let G (“group by”) and A (“aggregate”) be disjoint lists of positions from $\{1, \ldots, n\}$,
- let f be an aggregation function.

Notation: For a tuple $\vec{t} = \langle t_1, \ldots, t_n \rangle \in R$, let $G(\vec{t}) := \langle t_{G[1]}, \ldots, t_{G[\text{len}(G)]} \rangle$, and likewise for A.

Definition 8.6: The aggregation of R w.r.t. G, A, and f is defined through the following steps:

1. Grouping: Let $R_G := \{G(\vec{t}) \mid \vec{t} \in R\}$ and $R_A := \{A(\vec{t}) \mid \vec{t} \in R\}$.

   The function $g : R_G \to 2^{R_A}$ is defined by setting $g(\vec{s}) := \{A(\vec{t}) \mid \vec{t} \in R, G(\vec{t}) = \vec{s}\}$.
2. Apply aggregation: The aggregated relation is $\{\langle \vec{s}, f(g(\vec{s}))\rangle \mid \vec{s} \in R_G\}$.

Datalog Syntax for Aggregation

Users should be able to answer to some questions:
Which relation R is aggregated? What G do we group by? What A do we aggregate?

Syntax in Nemo:

```
        total(?org, #sum(?amount,?year)) :- emission(?org,?country,?year,?amount) .
```

- The aggregate function f is marked by #
- R is the (virtual) relation of all matching instances of the rule head
  (if two distinct body matches coincide on the head, they are treated as one)
- G are the head variables that occur outside of aggregation functions
- A are the variables inside the aggregation function

Datalog Syntax for Aggregation (2)

Syntax in Nemo:

```
countriesPerYear(?year,#count(?country)) :- emission(?org,?country,?year,?amount).
```

Syntax in Clingo:

```
countriesPerYear(YEAR, C) :- emission(_,_,_,YEAR),
                             C = #count{ COUNTRY : emission(_,COUNTRY,YEAR,_)} .
```

⇝ nested notation for aggregation

Syntax in Soufflé:

```
auxYearCountry(year, country) :- emission(_, country, year, _) .
    countriesPerYear(year, c) :- auxYearCountry(year, _),
                                 c = count : { auxYearCountry(year, _) } .
```

⇝ helper rule needed to eliminate duplicates (cannot just count distinct countries)

Stratified Aggregation

Aggregation is typically non-monotonic
(e.g., if we add more facts, previous counts will no longer be inferred)

Possible solution: Extend stratification to aggregates

- Same conditions for stratification of normal rules (with negation)
- For rules with aggregation, require that all predicates that we aggregate over are in a strictly lower stratum than the head predicate

$\leadsto$ aggregtion only applied to “final” sets of facts that will not change later on

Unstratified Aggregation

Stratified negation is most common in systems (e.g., Nemo, RDFox, Soufflé), but it prevents some useful applications

Example: In a directed graph that has a cost for every edge, we might want to compute minimal-cost paths as follows:

```
path(?s,?t,?c) :- edge(?s,?t,?c), ?c>0 .
path(?s,?t,#min(?cp+?c)) :- path(?s,?m,?cp), edge(?m,?t,?c), ?c>0 .
```

But this is not stratified, and therefore not supported. Indeed, it requires somewhat different processing.

Systems that support unstratified aggregation come in three forms:

- "Better safe than sorry" only special program shapes supported; well-defined semantics; guaranteed to be meaningful (example: SocialLite)
- "No strings attached" whatever users write down is "executed"; result may not be well-defined; complete freedom (example: Dynalog)
- "Stable models" ASP semantics; many models (example: Clingo, DLV)

Hands-on: Data integration and analysis

A new, somewhat more complex example: `https://tud.link/r44q79`

This example shows:

- How to load data from a Web-based CSV file
- How to match CSV records with Wikidata (by ID or name)
- How to use negation to implement a fallback handling (match ID before name)
- How to aggregate over partially matched data without overlooking unmatched records

Summary

Stratified negation is a practically important extension of Datalog

Datalog can incorporate datatypes (weak or strong), and functions that are similar to SPARQL

Aggregation (stratified or not) is useful for using Datalog for analytical queries

What's next?

- Complexity and Expressivity of SPARQL and Datalog
- Ontology languages
- Constraint languages for knowledged graphs

References

- Markus Krötzsch: **Modern Datalog: Concepts, Methods, Applications.** In Alessandro Artale, Meghyn Bienvenu, Yazmín Ibáñez García, Filip Murlak, eds., Joint Proceedings of the 20th and 21st Reasoning Web Summer Schools (RW 2024 & RW 2025), volume 138 of OASIcs. Dagstuhl Publishing