Technische Universität Dresden

International Center for Computational Logic

Knowledge Graphs

Lecture 7: Datalog for Knowledge Graphs

Markus Krötzsch

Knowledge-Based Systems

TU Dresden, 25 Nov 2025

Review

Semantics of each feature is defined by specific algebra operators

- Join(M_1, M_2): join compatible mappings from M_1 and M_2
- Filter$_G(\varphi, M)$: remove from multiset M all mappings for which φ does not evaluate to EBV "true"
- Union(M_1, M_2): compute the union of mappings from multisets M_1 and M_2
- Minus(M_1, M_2): remove from multiset M_1 all mappings compatible with a non-empty mapping in M_2
- LeftJoin$_G(M_1, M_2, \varphi)$: extend mappings from M_1 by compatible mappings from M_2 when filter condition is satisfied; keep remaining mappings from M_1 unchanged
- Extend(M, v, φ): extend all mappings from M by assigning v the value of φ.
- OrderBy$(L, \text{condition})$: sort list by a condition
- Slice$(L, \text{start}, \text{length})$: apply limit and offset modifiers

Further operators exist, e.g., Distinct(L).

Translating SPARQL to nested algebra expressions is mostly straightforward (we saw an algorithm for a subset of features).

Introduction to Datalog

The Simplest Rule Language

$$\text{hasUncle}(x, z) \leftarrow \text{hasParent}(x, y), \text{hasBrother}(y, z)$$

Some terminology:

- terms can be variables (e.g., x, y, z) or constants
- predicates denote relations; they have an arity (e.g., hasUncle has arity 2)
- atoms are constructed from predicates and terms (e.g., $\text{hasUncle}(x, z)$)

Definition 7.1: A Datalog rule is an expression of the form:

$$H \leftarrow B_1, \ldots, B_m$$

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 rule with empty body ($m = 0$) is called a fact. A ground rule is one without variables (i.e., all terms are constants).

Datalog Syntax in the Wild

In contrast to SPARQL and RDF, Datalog is not a standard . . .

```
           hasUncle(X, Z) :- hasFather(X, Y), hasBrother(Y, Z) .
           hasUncle(x, z) :- hasFather(x, y), hasBrother(y, z) .
         hasUncle(?x, ?z) :- hasFather(?x, ?y), hasBrother(?y, ?z) .
     [?x, :hasUncle, ?z] :- [?x, :hasFather, ?y], [?y, :hasBrother, ?z] .
 hasUncle(x, z) distinct :- hasFather(x, y), hasBrother(y, z) ;
      [(has-uncle ?x ?z)    [?x :has-father ?y] [?y :hasBrother ?z]]
hasUncle(p: x, uncle: z) :- hasFather(child: x, father: y),
                            hasBrother(p: y, brother: z) .
{ ?X :hasFather ?Y . ?Y :hasBrother ?Z .} => { ?X :hasUncle ?Z } .
```

Top to bottom: Prolog/ASP, Soufflé, Nemo, RDFox, Logica, Datomic, Percial, N3

From Rules to Programs

Example:

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

Trying it out

We will use Nemo for hands-on exercises.

Nemo
Graph Rule Engine

Web app: `https://tools.iccl.inf.tu-dresden.de/nemo/`

Open previous example online: `https://tud.link/ctdxps`

Task: Define an auncle (aunt or uncle) relation.

Hint: You might need inequality `!=`.

Nemo is free and open source. Issue reports and feature requests are welcome.

Datalog Programs as Functions

Idea:
Datalog programs represent functions from sets of input facts and to sets of output facts.

Definition 7.2: A Datalog program is a triple $\langle P, \mathbf{P}_{\text{in}}, \mathbf{P}_{\text{out}} \rangle$, where

- P is a finite set of rules,
- $\mathbf{P}_{\text{in}}$ is a non-empty set of input predicates that do not occur in rule heads of P, and
- $\mathbf{P}_{\text{out}}$ is a non-empty set of output predicates.

We require all predicates in $\mathbf{P}_{\text{in}}$ and $\mathbf{P}_{\text{out}}$ to occur in P.
Input predicates are also called EDB predicates, and all other (non-input) predicates are also called IDB predicates.

Note 1: Normally we want input facts to change, not to be a static part of programs.
Note 2: Finite sets of facts are often called databases, so Datalog maps input databases to output databases.

A Datalog Program

$$
\begin{aligned}
\mathrm{father}(\mathrm{alice}, \mathrm{bob}) \\
\mathrm{mother}(\mathrm{alice}, \mathrm{cho}) \\
\mathrm{father}(\mathrm{cho}, \mathrm{daniel}) \\
\mathrm{mother}(\mathrm{cho}, \mathrm{eiko}) \\
\mathrm{mother}(\mathrm{finley}, \mathrm{eiko}) \\
\mathrm{parent}(x, y) &\leftarrow \mathrm{father}(x, y) \\
\mathrm{parent}(x, y) &\leftarrow \mathrm{mother}(x, y) \\
\mathrm{ancestor}(x, y) &\leftarrow \mathrm{parent}(x, y) \\
\mathrm{ancestor}(x, z) &\leftarrow \mathrm{ancestor}(x, y), \mathrm{parent}(y, z) \\
\mathrm{commonAnc}(x) &\leftarrow \mathrm{ancestor}(\mathrm{alice}, x), \mathrm{ancestor}(\mathrm{finley}, x)
\end{aligned}
$$

Input (=EDB) predicates: father, mother

Output predicates: commonAnc

IDB predicates: parent, ancestor, commonAnc

An Equivalent Datalog Program

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

Input (=EDB) predicates: father, mother

Output predicates: commonAnc

IDB predicates: ancestor, commonAnc

Why Datalog?

Why Datalog?

The semantics of Datalog is defined in mathematical logic. What are the potential advantages?

- Declarativity: Programs describe high-level logical relationships, not detailed computational process
- System and Platform Independence: Portable to new computing environments
- Optimisability and Performance: Reasoners can optimise computation
- Explainability: Results are justifiable and traceable
- Verifiability and Certifiability: Correctness can be proven independently
- Intuitive Understandability: Logic capturing natural human thinking (questionable claim)
- Conciseness and Fast Development: Programs can focus on essentials of task at hand; program logic independent of underlying data structures

$\leadsto$ Could mostly be said about SPARQL as well . . .
but Datalog use cases are often hard or impossible to address with SPARQL

What is it good for? (1)

Application Area 1: Rule-Based Knowledge Representation and Reasoning

Datalog-like rules occur in many applications of formal logic:

- Direct use of rules in many reasoning approaches (e.g., qualitative spatial reasoning)
- Many logics admit Datalog-based reasoning mechanisms (e.g., reasoning for OWL EL ontologies)
- Sometimes reasoning subtasks can be outsourced to Datalog (e.g., grounding in ASP, unit propagation in theorem proving)

Typical system requirements:

- Fast, reactive systems; typically main-memory based
- Pure Datalog, limited need for extensions

What is it good for? (2)

Application Area 2: Analysis of Structured Data

Datalog is great for analysing structured data, especially with nested/recursive structures:

- Program analysis (e.g., CodeQuest)
- Data flow and control flow analysis (e.g., DOOP/Soufflé)
- HTML analsyis and data extraction (e.g., DIADEM)
- Analytical processing of structured data bases (e.g., legal conformance checks in health care)

Typical system requirements:

- Interactive or batch processing, depending on use case
- Data-related extensions (datatypes, built-ins, aggregation)
- Possibly domain-specific, structured datatypes

What is it good for? (3)

Application Area 3: Data Extraction and Transformation

Datalog allows us to define recursive views over large data collections:

- Rule-based relational DB data access (e.g., Yedalog, Logica)
- Rule-based graph DB data access (e.g., RDFox, Nemo)

Typical system requirements:

- Mostly interactive query answering
- Database bindings and matching datatypes
- Efficient update handling

What is it good for? (4)

Application Area 4: Graph and Network Analysis

Graph-like structures suggest iterative, declarative processing:

- Network analysis, e.g., PageRank centrality (e.g., EmptyHeaded, SocialLite)
- Graph algorithms, e.g., shortest path (e.g., SocialLite, Dynalog)

Typical system requirements:

- Mostly main-memory based, may or may not be batch processed
- Custom control for termination of approxiation algorithms
- Support for recursive use of aggretation (at least min and max)

Semantics of Datalog

From Intuiton to Formal Semantics

Intuition:
Given an input database, a Datalog program derives the output database that consists of all facts that are necessarily entailed by the given input facts and rules.

- Variables in rules represent arbitrary constants

Definition 7.3: A substitution σ is a partial mapping from variables to terms. A substitution is ground if it maps to constants only.

- A rule can be applied under specific substitutions of its variables

Definition 7.4: Consider a database $\mathcal{D}$ and a Datalog rule ρ of the form $H \leftarrow B_1, \ldots, B_n$. A ground substitution σ is a match for ρ on $\mathcal{D}$ if (1) σ is defined exactly on the variables in ρ, and (2) $B_1\sigma, \ldots, B_n\sigma \in \mathcal{D}$.
In this case, applying ρ to $\mathcal{D}$ under σ yields the inference $H\sigma$.

- The output of a program are the facts that follow by applying rules exhaustively.

The Consequence Operator

Definition 7.5: The immediate consequence operator T_P maps sets of ground facts $\mathcal{D}$ to sets of ground facts $T_P(\mathcal{D})$:

$$T_P(\mathcal{D}) = \{H\sigma \mid \text{there is some } H \leftarrow B_1, \ldots, B_n \in P \text{ with match } \sigma \text{ on } \mathcal{D}\}$$

Given a database $\mathcal{D}$, we can define a sequence of databases $\mathcal{D}_P^i$ as follows:

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

Observations:

- We obtain an increasing sequence $\mathcal{D}_P^0 \subseteq \mathcal{D}_P^1 \subseteq \mathcal{D}_P^2 \subseteq \ldots \subseteq \mathcal{D}_P^\infty$ (why?)
- Only a finite number of ground facts can ever be derived from $\mathcal{D} \cup P$ (why?).
- Hence the sequence $\mathcal{D}_P^0, \mathcal{D}_P^1, \ldots$ is finite and there is some $k \geq 1$ with $\mathcal{D}_P^k = \mathcal{D}_P^\infty$.

Definition 7.6: The output database of P over $\mathcal{D}$ is the restriction of $\mathcal{D}_P^\infty$ to output predicates, i.e., the set $\{p(c_1, \ldots, c_n) \mid p(c_1, \ldots, c_n) \in \mathcal{D}_P^\infty, p \in \mathbf{P}_{\text{out}}\}$.

The consequence operator: Example

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 $\mathcal{D}$:

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

$$\begin{aligned}
\mathcal{D}_P^0 &= \{\text{father}(\text{alice},\text{bob}), \text{mother}(\text{alice},\text{cho}), \text{mother}(\text{cho},\text{eiko}), \text{mother}(\text{finley},\text{eiko})\}\\
\mathcal{D}_P^1 &= \mathcal{D}_P^0 \cup \{\text{parent}(\text{alice},\text{bob}), \text{parent}(\text{alice},\text{cho}), \text{parent}(\text{cho},\text{eiko}), \text{parent}(\text{finley},\text{eiko})\}\\
\mathcal{D}_P^2 &= \mathcal{D}_P^1 \cup \{\text{ancestor}(\text{alice},\text{bob}), \text{ancestor}(\text{alice},\text{cho}), \text{ancestor}(\text{cho},\text{eiko}), \text{ancestor}(\text{finley},\text{eiko})\}\\
\mathcal{D}_P^3 &= \mathcal{D}_P^2 \cup \{\text{ancestor}(\text{alice},\text{eiko})\}\\
\mathcal{D}_P^4 &= \mathcal{D}_P^3 \cup \{\text{commonAnc}(\text{eiko})\}\\
\mathcal{D}_P^5 &= \mathcal{D}_P^4 = \mathcal{D}_P^\infty
\end{aligned}$$

Models of Datalog

Definition 7.7: An Herbrand model of P and $\mathcal{D}$ is a database $\mathcal{H}$ such that

1. $\mathcal{D} \subseteq \mathcal{H}$, and
2. for every rule $\rho \in P$ of the form $H \leftarrow B_1, \ldots, B_n$, and every match σ for ρ over $\mathcal{H}$, we also have $H\sigma \in \mathcal{H}$.

Notes:

- Herbrand models interpret constants "as themselves", hence can be defined as sets of facts.
- Among all Herbrand models of P and $\mathcal{D}$, there is actually a least one (w.r.t. $\subseteq$), which coincides with $\mathcal{D}_P^\infty$.

Theorem 7.8: The output database of P over $\mathcal{D}$ is equal to:

- the set of all output facts that are true in $\mathcal{D}_P^\infty$,
- the set of all output facts that are true in all Herbrand models of P and $\mathcal{D}$,
- the set of all output facts that are true in all first-order models of P and $\mathcal{D}$.

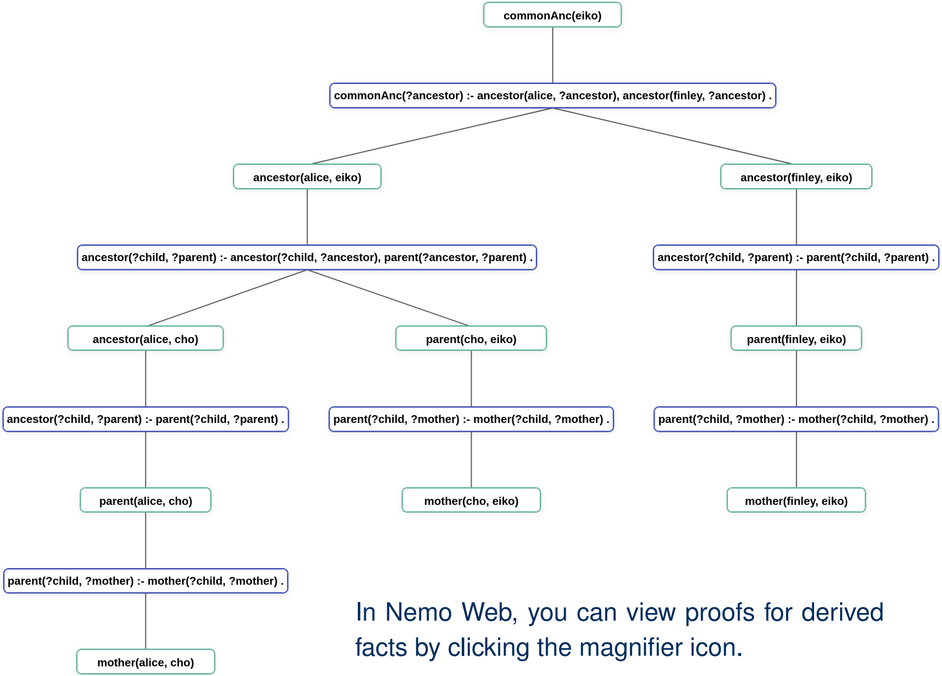
Proof trees

Definition 7.9: Consider a Datalog program P with input database $\mathcal{D}$. A proof tree with respect to P and $\mathcal{D}$ is a tree structure T that satisfies the following conditions:

1. every node n of T is labelled by a fact $\lambda(n)$,
2. if n is a leaf node, then $\lambda(n) \in \mathcal{D}$,
3. if n is an inner node, then n is additionally labelled by a rule $H \leftarrow B_1, \ldots, B_k \in P$ and a substitution σ, such that (1) $\lambda(n) = H\sigma$ and (2) n has exactly k child nodes $c_1, \ldots, c_k$ with $\lambda(c_i) = B_i\sigma$ for all $1 \leq i \leq k$.

If a proof tree T has root node r, we say that T is a proof for $\lambda(r)$ with respect to P and $\mathcal{D}$.

Theorem 7.10: The output database of P over $\mathcal{D}$ is equal to the set of all ground facts for which there is a proof with respect to P and $\mathcal{D}$.



In Nemo Web, you can view proofs for derived facts by clicking the magnifier icon.

Datalog as Second-Order Logic

Example 7.11: The following two programs are equivalent:

$$\begin{aligned}
\text{reach}(x,y) &\leftarrow \text{edge}(x,y) \\
\text{reach}(x,z) &\leftarrow \text{reach}(x,y), \text{reach}(y,z) \\
\text{output}(z) &\leftarrow \text{reach}(c,z)
\end{aligned}
\qquad\qquad
\begin{aligned}
\text{output}(y) &\leftarrow \text{edge}(c,y) \\
\text{output}(z) &\leftarrow \text{output}(y), \text{edge}(y,z)
\end{aligned}$$

Yet they do not have the same T_P operator, Herbrand models, or proof trees.

The following second-order logic formula captures the semantics more acurately:

$$\forall\, \text{Reach}, \text{Output}. \left(\left(\begin{array}{l} (\forall x, y.\ \text{edge}(x,y)) \to \text{Reach}(x,y)) \wedge \\ (\forall x, y, z.\ \text{Reach}(x,y) \wedge \text{Reach}(y,z) \to \text{Reach}(x,z)) \wedge \\ (\forall z.\ \text{Reach}(c,z) \to \text{Output}(z)) \end{array} \right) \to \text{Output}(v) \right)$$

Datalog Semantics: Summary

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

$\leadsto$ pleasing and reassuring agreement of various ideas, witnessing the underlying mathematical elegance

Note: Datalog is generally considered a fragment of second-order logic, but for most uses, we don't need to worry.

Working with Real 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 $\mathsf{triple}(s, p, o)$
- triple is the only predicate needed to represent arbitrary databases
- IRIs on any triple position can be related in Datalog

Where can input data come from?

We often want to use input databases that are not given as facts in the program:

- Scalability: Other formats are more suitable for large datasets
- Practicality: We don't want to edit our programs to change data
- Access: Some data sources cannot be converted to facts (size, access restrictions, legal constraints)

$\leadsto$ many systems support data imports

Commonly supported data sources include:

- CSV/TSV/DSV files: simple relational text format
- RDF: knowledge graphs in triples and quads
- SQL bindings: loading directly from relational DBMS
- SPARQL bindings: loading directly from RDF database

Data inputs in Nemo

Nemo supports CSV/TSV/DSV, RDF, and SPARQL

- RDF imports: Nemo can import triple data from all standard formats; imported triples are stored in a ternary predicate; a file path or URL needs to be specified
- SPARQL imports: Nemo can import query results of arbitrary SPARQL queries into predicates of suitable arity (depending on number of selected variables); a query service (endpoint) needs to be specified with the query

A Worked Example: https://tud.link/wkrajt

```
% Prefixes help to abbreviate long identifiers or URLs:
@prefix wdqs: <https://query.wikidata.org/> .
@prefix wd: <http://www.wikidata.org/entity/> .
% Parameters can be used for fixed terms throughout the program:
@parameter $personId1 = wd:Q7259 . % Ada Lovelace
@parameter $personId2 = wd:Q14045 . % Moby

% Import predicate "wdParent" (mother or father) through SPARQL:
@import wdParent :- sparql{
  endpoint=wdqs:sparql,
  query="""PREFIX wdt: <http://www.wikidata.org/prop/direct/>
    SELECT ?child ?parent WHERE { ?child (wdt:P22|wdt:P25) ?parent }"""
} .
% Import predicate "wdLabel" (English label) through SPARQL:
@import wdLabel :- sparql{
  endpoint=wdqs:sparql,
  query="""PREFIX wikibase: <http://wikiba.se/ontology#>
    SELECT ?qid ?qidLabel WHERE {
      SERVICE wikibase:label {
        <http://www.bigdata.com/rdf#serviceParam> wikibase:language "mul,en" } }"""
} .

% Find relevant ancestors, starting from selected persons:
ancestor($personId1, ?parent) :- wdParent($personId1, ?parent) .
ancestor($personId2, ?parent) :- wdParent($personId2, ?parent) .
 ancestor(?person, ?ancestor) :- ancestor(?person, ?x), wdParent(?x, ?ancestor) .
% Find common ancestors, and determine their names:
commonAnc(?qid, ?name) :- ancestor($personId1, ?qid), ancestor($personId2, ?qid),
                          wdLabel(?qid,?name) .
% Select one output predicate:
@output commonAnc .
```

Summary

Pure Datalog is an elegant and simple rule language

Datalog smoothly works with diverse data formats, including knowledge graphs

Datalog has a declarative, logic-based semantics, and this has important practical benefits

What's next?

- More features of Datalog
- Complexity and Expressivity of SPARQL and Datalog
- Ontology languages

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. Available online.