TECHNISCHE UNIVERSITÄT DRESDEN

DATABASE THEORY

Lecture 14: Datalog Evaluation

Markus Krötzsch

Knowledge-Based Systems

TU Dresden, 16 June 2025

Review: Datalog

A rule-based recursive query language

$$
\begin{aligned}
\mathsf{father}(\mathsf{alice}, \mathsf{bob}) & \\
\mathsf{mother}(\mathsf{alice}, \mathsf{carla}) & \\
\mathsf{Parent}(x, y) & \leftarrow \mathsf{father}(x, y) \\
\mathsf{Parent}(x, y) & \leftarrow \mathsf{mother}(x, y) \\
\mathsf{SameGeneration}(x, x) & \\
\mathsf{SameGeneration}(x, y) & \leftarrow \mathsf{Parent}(x, v) \wedge \mathsf{Parent}(y, w) \wedge \mathsf{SameGeneration}(v, w)
\end{aligned}
$$

- Datalog is more complex than FO query answering
- Datalog is more expressive than FO query answering
- Semipositive Datalog with a successor ordering captures P
- Datalog containment is undecidable

Remaining question: How can Datalog query answering be implemented?

Implementing Datalog

FO queries (and thus also CQs and UCQs) are supported by almost all DBMS
$\leadsto$ many specific implementation and optimisation techniques

How can Datalog queries be answered in practice?
$\leadsto$ techniques for dealing with recursion in DBMS query answering

There are two major paradigms for answering recursive queries:

- Bottom-up: derive conclusions by applying rules to given facts
- Top-down: search for proofs to infer results given query

Computing Datalog Query Answers Bottom-Up

We already saw a way to compute Datalog answers bottom-up:
the step-wise computation of the consequence operator T_P

Bottom-up computation is known under many names:

- Forward-chaining since rules are "chained" from premise to conclusion (common in logic programming)
- Materialisation since inferred facts are stored ("materialised") (common in databases)
- Saturation since the input database is "saturated" with inferences (common in theorem proving)
- Deductive closure since we "close" the input under entailments (common in formal logic)

Naive Evaluation of Datalog Queries

A direct approach for computing T_P^∞

01 $T_P^0 := \emptyset$
02 $i := 0$
03 **repeat** :
04 $\quad T_P^{i+1} := \emptyset$
05 $\quad$ **for** $H \leftarrow B_1 \wedge \ldots \wedge B_\ell \in P$:
06 $\quad\quad$ **for** $\theta \in B_1 \wedge \ldots \wedge B_\ell(T_P^i)$:
07 $\quad\quad\quad T_P^{i+1} := T_P^{i+1} \cup \{H\theta\}$
08 $\quad i := i + 1$
09 **until** $T_P^{i-1} = T_P^i$
10 **return** T_P^i

Notation for line 06/07:

- a substitution θ is a mapping from variables to database elements
- for a formula F, we write $F\theta$ for the formula obtained by replacing each free variable x in F by $\theta(x)$
- for a CQ Q and database $\mathcal{I}$, we write $\theta \in Q(\mathcal{I})$ if $\mathcal{I} \models Q\theta$

What's Wrong with Naive Evaluation?

An example Datalog program:

$$\mathsf{e}(1,2) \quad \mathsf{e}(2,3) \quad \mathsf{e}(3,4) \quad \mathsf{e}(4,5)$$

$$(R1) \qquad \mathsf{T}(x,y) \leftarrow \mathsf{e}(x,y)$$

$$(R2) \qquad \mathsf{T}(x,z) \leftarrow \mathsf{T}(x,y) \wedge \mathsf{T}(y,z)$$

How many body matches do we need to iterate over?

$T_P^0 = \emptyset$	initialisation
$T_P^1 = \{\mathsf{T}(1,2), \mathsf{T}(2,3), \mathsf{T}(3,4), \mathsf{T}(4,5)\}$	4 matches for $(R1)$
$T_P^2 = T_P^1 \cup \{\mathsf{T}(1,3), \mathsf{T}(2,4), \mathsf{T}(3,5)\}$	$4 \times (R1) + 3 \times (R2)$
$T_P^3 = T_P^2 \cup \{\mathsf{T}(1,4), \mathsf{T}(2,5), \mathsf{T}(1,5)\}$	$4 \times (R1) + 8 \times (R2)$
$T_P^4 = T_P^3 = T_P^\infty$	$4 \times (R1) + 10 \times (R2)$

In total, we considered 37 matches to derive 11 facts

Less Naive Evaluation Strategies

Does it really matter how often we consider a rule match?
After all, each fact is added only once . . .

In practice, finding applicable rules takes significant time, even if the conclusion does not need to be added – iteration takes time!
$\leadsto$ huge potential for optimisation

Observation:
we derive the same conclusions over and over again in each step

Idea: apply rules only to newly derived facts
$\leadsto$ semi-naive evaluation

Semi-Naive Evaluation

The computation yields sets $T_P^0 \subseteq T_P^1 \subseteq T_P^2 \subseteq \ldots \subseteq T_P^\infty$

- For an IDB predicate R, let R^i be the “predicate” that contains exactly the R-facts in T_P^i
- For $i \leq 1$, let Δ_{R}^i be the collection of facts $\mathsf{R}^i \setminus \mathsf{R}^{i-1}$

We can restrict rules to use only some computations.

Some options for the computation in step $i+1$:

$\mathsf{T}(x,z) \leftarrow \mathsf{T}^i(x,y) \wedge \mathsf{T}^i(y,z)$	same as original rule
$\mathsf{T}(x,z) \leftarrow \Delta_{\mathsf{T}}^i(x,y) \wedge \Delta_{\mathsf{T}}^i(y,z)$	restrict to new facts
$\mathsf{T}(x,z) \leftarrow \Delta_{\mathsf{T}}^i(x,y) \wedge \mathsf{T}^i(y,z)$	partially restrict to new facts
$\mathsf{T}(x,z) \leftarrow \mathsf{T}^i(x,y) \wedge \Delta_{\mathsf{T}}^i(y,z)$	partially restrict to new facts

What to choose?

Semi-Naive Evaluation (2)

Inferences that involve new and old facts are necessary:

$$\mathsf{e}(1,2) \quad \mathsf{e}(2,3) \quad \mathsf{e}(3,4) \quad \mathsf{e}(4,5)$$

$$(R1) \qquad \mathsf{T}(x,y) \leftarrow \mathsf{e}(x,y)$$

$$(R2) \qquad \mathsf{T}(x,z) \leftarrow \mathsf{T}(x,y) \wedge \mathsf{T}(y,z)$$

$$\begin{array}{ll}
 & T_P^0 = \emptyset \\
\Delta_\mathsf{T}^1 = \{\mathsf{T}(1,2), \mathsf{T}(2,3), \mathsf{T}(3,4), \mathsf{T}(4,5)\} & T_P^1 = \Delta_\mathsf{T}^1 \\
\Delta_\mathsf{T}^2 = \{\mathsf{T}(1,3), \mathsf{T}(2,4), \mathsf{T}(3,5)\} & T_P^2 = T_P^1 \cup \Delta_\mathsf{T}^2 \\
\Delta_\mathsf{T}^3 = \{\mathsf{T}(1,4), \mathsf{T}(2,5), \mathsf{T}(1,5)\} & T_P^3 = T_P^2 \cup \Delta_\mathsf{T}^3 \\
\Delta_\mathsf{T}^4 = \emptyset & T_P^4 = T_P^3 = T_P^\infty
\end{array}$$

To derive $\mathsf{T}(1,4)$ in Δ_T^3, we need to combine
$\mathsf{T}(1,3) \in \Delta_\mathsf{T}^2$ with $\mathsf{T}(3,4) \in \Delta_\mathsf{T}^1$ or $\mathsf{T}(1,2) \in \Delta_\mathsf{T}^1$ with $\mathsf{T}(2,4) \in \Delta_\mathsf{T}^2$
$\leadsto$ rule $\mathsf{T}(x,z) \leftarrow \Delta_\mathsf{T}^i(x,y) \wedge \Delta_\mathsf{T}^i(y,z)$ is not enough

Semi-Naive Evaluation (3)

Correct approach: consider only rule application that use at least one newly derived IDB atom

For example program:

$$
\begin{array}{rl}
 & \mathsf{e}(1,2) \quad \mathsf{e}(2,3) \quad \mathsf{e}(3,4) \quad \mathsf{e}(4,5) \\
(R1) & \mathsf{T}(x,y) \leftarrow \mathsf{e}(x,y) \\
(R2.1) & \mathsf{T}(x,z) \leftarrow \Delta^i_{\mathsf{T}}(x,y) \wedge \mathsf{T}^i(y,z) \\
(R2.2) & \mathsf{T}(x,z) \leftarrow \mathsf{T}^i(x,y) \wedge \Delta^i_{\mathsf{T}}(y,z)
\end{array}
$$

There is still redundancy here: the matches for $\mathsf{T}(x,z) \leftarrow \Delta^i_{\mathsf{T}}(x,y) \wedge \Delta^i_{\mathsf{T}}(y,z)$ are covered by both $(R2.1)$ and $(R2.2)$

$\leadsto$ replace $(R2.2)$ by the following rule:

$$
(R2.2') \qquad \mathsf{T}(x,z) \leftarrow \mathsf{T}^{i-1}(x,y) \wedge \Delta^i_{\mathsf{T}}(y,z)
$$

EDB atoms do not change, so their Δ would be $\emptyset$

$\leadsto$ ignore such rules after the first iteration

Semi-Naive Evaluation: Example

$$\mathsf{e}(1,2) \quad \mathsf{e}(2,3) \quad \mathsf{e}(3,4) \quad \mathsf{e}(4,5)$$

$$(R1) \qquad \mathsf{T}(x,y) \leftarrow \mathsf{e}(x,y)$$

$$(R2.1) \qquad \mathsf{T}(x,z) \leftarrow \Delta^i_{\mathsf{T}}(x,y) \wedge \mathsf{T}^i(y,z)$$

$$(R2.2') \qquad \mathsf{T}(x,z) \leftarrow \mathsf{T}^{i-1}(x,y) \wedge \Delta^i_{\mathsf{T}}(y,z)$$

How many body matches do we need to iterate over?

$T_P^0 = \emptyset$	initialisation
$T_P^1 = \{\mathsf{T}(1,2), \mathsf{T}(2,3), \mathsf{T}(3,4), \mathsf{T}(4,5)\}$	$4 \times (R1)$
$T_P^2 = T_P^1 \cup \{\mathsf{T}(1,3), \mathsf{T}(2,4), \mathsf{T}(3,5)\}$	$3 \times (R2.1)$
$T_P^3 = T_P^2 \cup \{\mathsf{T}(1,4), \mathsf{T}(2,5), \mathsf{T}(1,5)\}$	$3 \times (R2.1), 2 \times (R2.2')$
$T_P^4 = T_P^3 = T_P^\infty$	$1 \times (R2.1), 1 \times (R2.2')$

In total, we considered 14 matches to derive 11 facts

Semi-Naive Evaluation: Full Definition

In general, a rule of the form

$$\mathsf{H}(\vec{x}) \leftarrow \mathsf{e}_1(\vec{y}_1) \wedge \ldots \wedge \mathsf{e}_n(\vec{y}_n) \wedge \mathsf{I}_1(\vec{z}_1) \wedge \mathsf{I}_2(\vec{z}_2) \wedge \ldots \wedge \mathsf{I}_m(\vec{z}_m)$$

is transformed into m rules

$$\begin{aligned}
\mathsf{H}(\vec{x}) &\leftarrow \mathsf{e}_1(\vec{y}_1) \wedge \ldots \wedge \mathsf{e}_n(\vec{y}_n) \wedge \Delta^i_{\mathsf{I}_1}(\vec{z}_1) \wedge \mathsf{I}^i_2(\vec{z}_2) \wedge \ldots \wedge \mathsf{I}^i_m(\vec{z}_m) \\
\mathsf{H}(\vec{x}) &\leftarrow \mathsf{e}_1(\vec{y}_1) \wedge \ldots \wedge \mathsf{e}_n(\vec{y}_n) \wedge \mathsf{I}^{i-1}_1(\vec{z}_1) \wedge \Delta^i_{\mathsf{I}_2}(\vec{z}_2) \wedge \ldots \wedge \mathsf{I}^i_m(\vec{z}_m) \\
&\ldots \\
\mathsf{H}(\vec{x}) &\leftarrow \mathsf{e}_1(\vec{y}_1) \wedge \ldots \wedge \mathsf{e}_n(\vec{y}_n) \wedge \mathsf{I}^{i-1}_1(\vec{z}_1) \wedge \mathsf{I}^{i-1}_2(\vec{z}_2) \wedge \ldots \wedge \Delta^i_{\mathsf{I}_m}(\vec{z}_m)
\end{aligned}$$

Advantages and disadvantages:

- Huge improvement over naive evaluation
- Some redundant computations remain (see example)
- Some overhead for implementation (store level of entailments)

Goal-Directed Datalog Evaluation

Top-Down Evaluation

Idea: we may not need to compute all derivations to answer a particular query

Example 14.1:

$$\mathsf{e}(1,2) \quad \mathsf{e}(2,3) \quad \mathsf{e}(3,4) \quad \mathsf{e}(4,5)$$

$$(R1) \qquad \mathsf{T}(x,y) \leftarrow \mathsf{e}(x,y)$$

$$(R2) \qquad \mathsf{T}(x,z) \leftarrow \mathsf{T}(x,y) \wedge \mathsf{T}(y,z)$$

$$\mathsf{Query}(z) \leftarrow \mathsf{T}(2,z)$$

The answers to Query are the T-successors of 2.

However, bottom-up computation would also produce facts like $\mathsf{T}(1,4)$, which are neither directly nor indirectly relevant for computing the query result.

Assumption

Assumption: For all techniques presented in this lecture, we assume that the given Datalog program is safe.

- This is without loss of generality (as shown in exercise).
- One can avoid this by adding more cases to algorithms.

Query-Subquery (QSQ)

QSQ is a technique for organising top-down Datalog query evaluation

Main principles:

- Apply backward chaining/resolution: start with query, find rules that can derive query, evaluate body atoms of those rules (subqueries) recursively
- Evaluate intermediate results "set-at-a-time" (using relational algebra on tables)
- Evaluate queries in a "data-driven" way, where operations are applied only to newly computed intermediate results (similar to idea in semi-naive evaluation)
- "Push" variable bindings (constants) from heads (queries) into bodies (subqueries)
- "Pass" variable bindings (constants) "sideways" from one body atom to the next

Details can be realised in several ways.

Adornments

To guide evaluation, we distinguish free and bound parameters in a predicate.

Example 14.2: If we want to derive atom $\mathsf{T}(2, z)$ from the rule $\mathsf{T}(x, z) \leftarrow \mathsf{T}(x, y) \wedge \mathsf{T}(y, z)$, then x will be bound to 2, while z is free.

We use adornments to denote the free/bound parameters in predicates.

Example 14.3:

$$\mathsf{T}^{bf}(x, z) \leftarrow \mathsf{T}^{bf}(x, y) \wedge \mathsf{T}^{bf}(y, z)$$

- since x is bound in the head, it is also bound in the first atom
- any match for the first atom binds y, so y is bound when evaluating the second atom (in left-to-right evaluation)

Adornments: Examples

The adornment of the head of a rule determines the adornments of the body atoms:

$$\mathsf{R}^{bbb}(x, y, z) \leftarrow \mathsf{R}^{bbf}(x, y, v) \wedge \mathsf{R}^{bbb}(x, v, z)$$
$$\mathsf{R}^{fbf}(x, y, z) \leftarrow \mathsf{R}^{fbf}(x, y, v) \wedge \mathsf{R}^{bbf}(x, v, z)$$

The order of body predicates affects the adornment:

$$\mathsf{S}^{fff}(x, y, z) \leftarrow \mathsf{T}^{ff}(x, v) \wedge \mathsf{T}^{ff}(y, w) \wedge \mathsf{R}^{bbf}(v, w, z)$$
$$\mathsf{S}^{fff}(x, y, z) \leftarrow \mathsf{R}^{fff}(v, w, z) \wedge \mathsf{T}^{fb}(x, v) \wedge \mathsf{T}^{fb}(y, w)$$

$\leadsto$ For optimisation, some orders might be better than others

Auxiliary Relations for QSQ

To control evaluation, we store intermediate results in auxiliary relations.

When we "call" a rule with a head where some variables are bound, we need to provide the bindings as input

$\leadsto$ for adorned relation R^{α}, we use an auxiliary relation $\mathsf{input}_{\mathsf{R}}^{\alpha}$

$\leadsto$ arity of $\mathsf{input}_{\mathsf{R}}^{\alpha}$ = number of b in α

The result of calling a rule should be the "completed" input, with values for the unbound variables added

$\leadsto$ for adorned relation R^{α}, we use an auxiliary relation $\mathsf{output}_{\mathsf{R}}^{\alpha}$

$\leadsto$ arity of $\mathsf{output}_{\mathsf{R}}^{\alpha}$ = arity of R (= length of α)

Auxiliary Relations for QSQ (2)

When evaluating body atoms from left to right, we use supplementary relations sup_i

$\leadsto$ bindings required to evaluate rest of rule after the ith body atom

$\leadsto$ the first set of bindings sup_0 comes from $\mathsf{input}^{\alpha}_{\mathsf{R}}$

$\leadsto$ the last set of bindings sup_n go to $\mathsf{output}^{\alpha}_{\mathsf{R}}$

Example 14.4:

$$\begin{array}{ccccccc}
\mathsf{T}^{bf}(x,z) \leftarrow & \mathsf{T}^{bf}(x,y) & \wedge & \mathsf{T}^{bf}(y,z) & & & \\
& \Uparrow & \searrow & \Uparrow & \searrow & & \\
\mathsf{input}^{bf}_{\mathsf{T}} \Rightarrow & \mathsf{sup}_0[x] & & \mathsf{sup}_1[x,y] & & \mathsf{sup}_2[x,z] \Rightarrow & \mathsf{output}^{bf}_{\mathsf{T}}
\end{array}$$

- $\mathsf{sup}_0[x]$ is copied from $\mathsf{input}^{bf}_{\mathsf{T}}[x]$ (with some exceptions, see exercise)
- $\mathsf{sup}_1[x,y]$ is obtained by joining tables $\mathsf{sup}_0[x]$ and $\mathsf{output}^{bf}_{\mathsf{T}}[x,y]$
- $\mathsf{sup}_2[x,z]$ is obtained by joining tables $\mathsf{sup}_1[x,y]$ and $\mathsf{output}^{bf}_{\mathsf{T}}[y,z]$
- $\mathsf{output}^{bf}_{\mathsf{T}}[x,z]$ is copied from $\mathsf{sup}_2[x,z]$

(we use "named" notation like $[x,y]$ to suggest what to join on; the relations are the same)

QSQ Evaluation

The set of all auxiliary relations is called a QSQ template (for the given set of adorned rules)

General evaluation:

- add new tuples to auxiliary relations until reaching a fixed point
- evaluation of a rule can proceed as sketched on previous slide
- in addition, whenever new tuples are added to a sup relation that feeds into an IDB atom, the input relation of this atom is extended to include all binding given by sup (may trigger subquery evaluation)

$\leadsto$ there are many strategies for implementing this general scheme

Notation:

- for an EDB atom A, we write $A^{\mathcal{I}}$ for table that consists of all matches for A in the database

Recursive QSQ

Recursive QSQ (QSQR) takes a “depth-first” approach to QSQ

Evaluation of single rule in QSQR:

Given: adorned rule r with head predicate R^{α}; current values of all QSQ relations

(1) Copy tuples $\mathsf{input}_{\mathsf{R}}^{\alpha}$ (that unify with rule head) to sup_0^r

(2) For each body atom $A_1, \ldots, A_n$, do:
   - If A_i is an EDB atom, compute sup_i^r as projection of $\mathsf{sup}_{i-1}^r \bowtie A_i^{\mathcal{I}}$
   - If A_i is an IDB atom with adorned predicate S^{β}:
     (a) Add new bindings from sup_{i-1}^r, combined with constants in A_i, to $\mathsf{input}_{\mathsf{S}}^{\beta}$
     (b) If $\mathsf{input}_{\mathsf{S}}^{\beta}$ changed, recursively evaluate all rules with head predicate S^{β}
     (c) Compute sup_i^r as projection of $\mathsf{sup}_{i-1}^r \bowtie \mathsf{output}_{\mathsf{S}}^{\beta}$

(3) Add tuples in sup_n^r to $\mathsf{output}_{\mathsf{R}}^{\alpha}$

QSQR Algorithm

Evaluation of query in QSQR:

Given: a Datalog program P and a conjunctive query $q[\vec{x}]$ (possibly with constants)

(1) Create an adorned program P^a:
   - Turn the query $q[\vec{x}]$ into an adorned rule $\mathrm{Query}^{ff\ldots f}(\vec{x}) \leftarrow q[\vec{x}]$
   - Recursively create adorned rules from rules in P for all adorned predicates in P^a.

(2) Initialise all auxiliary relations to empty sets.

(3) Evaluate the rule $\mathrm{Query}^{ff\ldots f}(\vec{x}) \leftarrow q[\vec{x}]$.
Repeat until no new tuples are added to any QSQ relation.

(4) Return $\mathrm{output}^{ff\ldots f}_{\mathrm{Query}}$.

QSQR Transformation: Example

Predicates S (same generation), p (parent), h (human)

$$\mathsf{S}(x,x) \leftarrow \mathsf{h}(x)$$
$$\mathsf{S}(x,y) \leftarrow \mathsf{p}(x,w) \wedge \mathsf{S}(v,w) \wedge \mathsf{p}(y,v)$$

with query $\mathsf{S}(1,x)$.
$\leadsto$ Query rule: $\mathsf{Query}(x) \leftarrow \mathsf{S}(1,x)$

Transformed rules:

$$\mathsf{Query}^{f}(x) \leftarrow \mathsf{S}^{bf}(1,x)$$
$$\mathsf{S}^{bf}(x,x) \leftarrow \mathsf{h}(x)$$
$$\mathsf{S}^{bf}(x,y) \leftarrow \mathsf{p}(x,w) \wedge \mathsf{S}^{fb}(v,w) \wedge \mathsf{p}(y,v)$$
$$\mathsf{S}^{fb}(x,x) \leftarrow \mathsf{h}(x)$$
$$\mathsf{S}^{fb}(x,y) \leftarrow \mathsf{p}(x,w) \wedge \mathsf{S}^{fb}(v,w) \wedge \mathsf{p}(y,v)$$

Summary and Outlook

Datalog queries can be evaluated bottom-up or top-down

Simplest practical bottom-up technique: semi-naive evaluation

Top-down: Query-Subquery (QSQ) approach (goal-directed)

Next question:

- Can bottom-up evaluations be goal directed?
- What about practical implementations?
- Graph databases