TECHNISCHE UNIVERSITÄT DRESDEN

DATABASE THEORY

Lecture 14: Datalog Implementation

David Carral

Knowledge-Based Systems

TU Dresden, May 26, 2020

Review: Datalog

A rule-based recursive query language

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

- 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^i_{R} 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
$\mathsf{T}(x, z) \leftarrow \Delta^i_{\mathsf{T}}(x, y) \wedge \Delta^i_{\mathsf{T}}(y, z)$	restrict to new facts
$\mathsf{T}(x, z) \leftarrow \Delta^i_{\mathsf{T}}(x, y) \wedge \mathsf{T}^i(y, z)$	partially restrict to new facts
$\mathsf{T}(x, z) \leftarrow \mathsf{T}^i(x, y) \wedge \Delta^i_{\mathsf{T}}(y, z)$	partially restrict to new facts

What to choose?

Semi-Naive Evaluation (2)

Inferences that involve new and old facts are necessary:

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

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

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

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

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

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

$$\begin{array}{rl}
(R1) & \mathsf{T}(x,y) \leftarrow \mathsf{e}(x,y) \\
(R2.1) & \mathsf{T}(x,z) \leftarrow \Delta^i_{\mathsf{T}}(x,y) \wedge \mathsf{T}^i(y,z) \\
(R2.2') & \mathsf{T}(x,z) \leftarrow \mathsf{T}^{i-1}(x,y) \wedge \Delta^i_{\mathsf{T}}(y,z)
\end{array}$$

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)

Summary and Outlook

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

Simplest practical bottom-up technique: semi-naive evaluation

Next question:

- Can we improve Datalog evaluation further?
- What about practical implementations?