Review: Datalog

A rule-based recursive query language


definitions:

- father:
  - father(alice, bob)
- mother:
  - mother(alice, carla)
- Parent:
  - Parent(x, y) ← father(x, y)
  - Parent(x, y) ← mother(x, y)
- SameGeneration:
  - SameGeneration(x, y)

- 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 $T_p^{\infty}$

01 $T_p^0 := \emptyset$
02 $i := 0$
03 repeat:
04 $T_p^{i+1} := \emptyset$
05 for $H \leftarrow B_1 \land \ldots \land B_l \in P$:
06 for $\theta \in B_1 \land \ldots \land B_l (T_p^i)$:
07 $T_p^{i+1} := T_p^{i+1} \cup [H\theta]$
08 $i := i + 1$
09 until $T_p^{i+1} = T_p^i$
10 return $T_p^i$

Notation for line 06/07:
• a substitution $\theta$ 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 $I$, we write $\theta \in Q(I)$ if $I \models Q\theta$

What’s Wrong with Naive Evaluation?

An example Datalog program:

\[
\begin{align*}
&\text{(R1)} \quad T(x, y) \leftarrow \theta(x, y) \\
&\text{(R2)} \quad T(x, z) \leftarrow T(x, y) \land T(y, z)
\end{align*}
\]

How many body matches do we need to iterate over?

\[
\begin{align*}
T_p^0 &= 0 \quad \text{initialisation} \\
T_p^1 &= |T(1, 2), T(2, 3), T(3, 4), T(4, 5)| \quad 4 \text{ matches for } (R1) \\
T_p^2 &= T_p^1 \cup [T(1, 3), T(2, 4), T(3, 5)] \quad 4 \times (R1) + 3 \times (R2) \\
T_p^3 &= T_p^2 \cup [T(1, 4), T(2, 5), T(1, 5)] \quad 4 \times (R1) + 8 \times (R2) \\
T_p^4 &= T_p^3 = T_p^{\infty} \quad 4 \times (R1) + 10 \times (R2)
\end{align*}
\]

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!

~ 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
~ 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 $R^\prime$ be the “predicate” that contains exactly the $R$-facts in $T_p^i$
• For $i \leq 1$, let $\Delta_R^i$ be the collection of facts $R^\prime \setminus R^{i-1}$

We can restrict rules to use only some computations.

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

\[
\begin{align*}
T(x, z) &\leftarrow T'(x, y) \land T(y, z) \quad \text{same as original rule} \\
T(x, z) &\leftarrow \Delta_R^i(x, y) \land \Delta_R^i(y, z) \quad \text{restrict to new facts} \\
T(x, z) &\leftarrow \Delta_R^i(x, y) \land T'(y, z) \quad \text{partially restrict to new facts} \\
T(x, z) &\leftarrow T'(x, y) \land \Delta_R^i(y, z) \quad \text{partially restrict to new facts}
\end{align*}
\]

What to choose?
Semi-Naive Evaluation: Example

Inferences that involve new and old facts are necessary:

\[ e(1, 2) \quad e(2, 3) \quad e(3, 4) \quad e(4, 5) \]

(1) \( T(x, y) \leftarrow e(x, y) \)

(2) \( T(x, z) \leftarrow \Delta_i \)

To derive \( T(x, z) \) we need to combine

\[ \Delta_1 = \{ T(1, 2), T(2, 3), T(3, 4), T(4, 5) \} \]

\[ \Delta_2 = \{ T(1, 3), T(2, 4), T(3, 5) \} \]

\[ \Delta_3 = \{ T(1, 4), T(2, 5), T(1, 5) \} \]

\[ \Delta_4 = \emptyset \]

We consider only rule application that use at least one newly derived

To derive \( T(x, z) \) we need to combine \( \Delta_i \) with \( T(3, 4) \in \Delta_1 \) or \( T(1, 2) \in \Delta_1 \) with \( T(2, 4) \in \Delta_2 \)

\[ T_0^p = \emptyset \]

\[ T_1^p = T_1^p \cup \Delta_1 \]

\[ T_2^p = T_2^p \cup \Delta_2 \]

\[ T_3^p = T_3^p \cup \Delta_3 \]

\[ T_4^p = \emptyset \]

Semi-Naive Evaluation: Full Definition

In general, a rule of the form

\[ H(\vec{x}) \leftarrow e_1(\vec{y}_1) \land \ldots \land e_n(\vec{y}_n) \land I_1(\vec{z}_1) \land I_2(\vec{z}_2) \ldots \land I_m(\vec{z}_m) \]

is transformed into \( m \) rules

\[ H(\vec{x}) \leftarrow e_1(\vec{y}_1) \land \ldots \land e_n(\vec{y}_n) \land I_1(\vec{z}_1) \land I_2(\vec{z}_2) \ldots \land I_m(\vec{z}_m) \]

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?