Overview

1. Introduction | Relational data model
2. First-order queries
3. Complexity of query answering
4. Complexity of FO query answering
5. Conjunctive queries
6. Tree-like conjunctive queries
7. Query optimisation
8. Conjunctive Query Optimisation / First-Order Expressiveness
9. First-Order Expressiveness / Introduction to Datalog
10. Expressive Power and Complexity of Datalog
11. Optimisation and Evaluation of Datalog
12. Evaluation of Datalog (2)
13. Path queries
14. Outlook: database theory in practice

See course homepage [⇒ link] for more information and materials

Review: Datalog Expressivity and Complexity

A rule-based recursive query language

\[
\begin{align*}
&\text{father}(alice, bob) \\
&\text{mother}(alice, carla) \\
&\quad \text{Parent}(x, y) \leftarrow \text{father}(x, y) \\
&\quad \text{Parent}(x, y) \leftarrow \text{mother}(x, y) \\
&\text{SameGeneration}(x, x) \\
&\text{SameGeneration}(x, y) \leftarrow \text{Parent}(x, v) \land \text{Parent}(y, w) \land \text{SameGeneration}(v, w)
\end{align*}
\]

Datalog Implementation and Optimisation

How can Datalog query answering be implemented?
How can Datalog queries be optimised?

Recall: static query optimisation

- Query equivalence
- Query emptiness
- Query containment

⇒ all undecidable for FO queries, but decidable for (U)CQs

Datalog is more complex than FO query answering:

- \(\text{EXPTIME}\)-complete for query and combined complexity
- \(\text{P}\)-complete for data complexity

Datalog cannot express all query mappings in \(\text{P}\)
but semipositive Datalog with a successor ordering can
Learning from CQ Containment?

How did we manage to decide the question $Q_1 \sqsubseteq Q_2$ for conjunctive queries $Q_1$ and $Q_2$?

Key ideas were:
- We want to know if all situations where $Q_1$ matches are also matched by $Q_2$.
- We can simply view $Q_1$ as a database $I_{Q_1}$: the most general database that $Q_1$ can match to.
- Containment $Q_1 \sqsubseteq Q_2$ holds if $Q_2$ matches the database $I_{Q_1}$.

$\Rightarrow$ decidable in NP

A CQ $Q[x_1, \ldots, x_n]$ can be expressed as a Datalog query with a single rule $Ans(x_1, \ldots, x_n) \leftarrow Q$

$\Rightarrow$ Could we apply a similar technique to Datalog?

Example: Rule Entailment

Let $P$ be the program

\[
\begin{align*}
\text{Ancestor}(x, y) & \leftarrow \text{parent}(x, y) \\
\text{Ancestor}(x, z) & \leftarrow \text{parent}(x, y) \land \text{Ancestor}(y, z)
\end{align*}
\]

and consider the rule $\text{Ancestor}(x, z) \leftarrow \text{parent}(x, y) \land \text{parent}(y, z)$.

Then $I_{\text{parent}(x,y)\land\text{parent}(y,z)} = \{\text{parent}(c_x, c_y), \text{parent}(c_y, c_z)\}$ (abbr. as $I$).

We can compute $T^p(I)$:

\[
\begin{align*}
T^p_0(I) &= I \\
T^p_2(I) &= \{\text{Ancestor}(c_x, c_y), \text{Ancestor}(c_y, c_z)\} \cup I \\
T^p_3(I) &= \{\text{Ancestor}(c_x, c_z) \cup T^p_1(I) \\
T^p_4(I) &= T^p_3(I) = T^p_5(I)
\end{align*}
\]

Therefore, $\text{Ancestor}(x, z) \Rightarrow \text{Ancestor}(c_x, c_z) \in T^p_{\infty}(I)$, so $P$ entails $\text{Ancestor}(x, z) \leftarrow \text{parent}(x, y) \land \text{parent}(y, z)$.

Deciding Datalog Containment?

Idea for two Datalog programs $P_1$ and $P_2$:
- If $P_2 \models P_1$, then every entailment of $P_1$ is also entailed by $P_2$
- In particular, this means that $P_1$ is contained in $P_2$
- We have $P_2 \models P_1$ if $P_2 \models H \leftarrow B_1 \land \ldots \land B_n$ for every rule $H \leftarrow B_1 \land \ldots \land B_n \in P_1$
- We can decide $P_2 \models H \leftarrow B_1 \land \ldots \land B_n$.

Can we decide Datalog containment this way?

$\Rightarrow$ No! In fact, Datalog containment is undecidable. What’s wrong?
Implication Entailment vs. Datalog Entailment

Consider the Datalog queries \( \langle A, P_1 \rangle \) and \( \langle B, P_2 \rangle \):
- Clearly, \( \langle A, P_1 \rangle \) and \( \langle B, P_2 \rangle \) are equivalent (and mutually contained in each other).
- However, \( P_2 \) entails no rule of \( P_1 \) and \( P_1 \) entails no rule of \( P_2 \).

\( \leadsto \) IDB predicates do not matter in Datalog, but predicate names matter in first-order implications.

First-Order vs. Second-Order Logic

A Datalog program looks like a set of first-order implications, but it has a second-order semantics.

We have already seen that Datalog can express things that are impossible to express in FO queries – that’s why we introduced it!

Consequences for query optimisation:
- Entailment between sets of first-order implications is decidable (shown above).
- Containment between Datalog queries is not decidable (shown next).

Undecidability of Datalog Query Containment

A classical undecidable problem: Post Correspondence Problem
- Input: two lists of words \( \alpha_1, \ldots, \alpha_n \) and \( \beta_1, \ldots, \beta_n \)
- Output: “yes” if there is a sequence of indices \( i_1, i_2, i_3, \ldots, i_m \) such that \( \alpha_{i_1} \alpha_{i_2} \alpha_{i_3} \cdots \alpha_{i_m} = \beta_{i_1} \beta_{i_2} \beta_{i_3} \cdots \beta_{i_m} \).

\( \leadsto \) we will reduce PCP to Datalog containment.

We need to define Datalog programs that work on databases that encode words:
- We represent words by chains of binary predicates
- Binary EDB predicates represent a letters
- For each letter \( \sigma \), we use a binary EDB predicate \( \text{letter}[\sigma] \)
- We assume that the words \( \alpha_i \) have the form \( a_{i_1}^1 \cdots a_{i_{|\alpha_i|}}^1 \) and that the words \( \beta_i \) have the form \( b_{i_1}^1 \cdots b_{i_{|\beta_i|}}^1 \)
Solving PCP with Datalog Containment

A program $P_1$ to recognise potential PCP solutions.

Rules to recognise words $\alpha_i$ and $\beta_i$ for every $i \in \{1, \ldots, m\}$:

$A_i(x_0, x_{i0}) \leftarrow \text{letter}[\alpha_i](x_0, x_1) \land \ldots \land \text{letter}[\alpha_i](x_{i0}, x_{i0})$

$B_i(x_0, x_{i1}) \leftarrow \text{letter}[\beta_i](x_0, x_1) \land \ldots \land \text{letter}[\beta_i](x_{i1}, x_{i1})$

Rules to check for synchronised chairs (for all $i \in \{1, \ldots, m\}$):

$\text{PCP}(x, y_1, y_2) \leftarrow A_i(x, y_1) \land B_i(x, y_2)$

$\text{PCP}(x, z_1, z_2) \leftarrow \text{PCP}(x, y_1, y_2) \land A_i(y_1, z_1) \land B_i(y_2, z_2)$

Accept() $\leftarrow \text{PCP}(x, z, z)$

Additional IDB facts that are derived (among others):

$\text{PCP}(1, 3, 2) \quad \text{PCP}(1, 5, 3) \quad \text{PCP}(1, 6, 6) \quad \text{Accept()}$

Solving PCP with Datalog Containment (3)

Example: $\alpha_1 = aaaaa, \beta_1 = bbb$

Problem: $P_1$ also accepts some unintended cases

Additional IDB facts that are derived:

$\text{PCP}(1, 6, 6) \quad \text{Accept()}$

Solving PCP with Datalog Containment (4)

Solution: specify a program $P_2$ that recognises all unwanted cases

$P_2$ consists of the following rules (for all letters $\sigma, \sigma'$):

$\text{EP}(x, x) \leftarrow$

$\text{EP}(y_1, y_2) \leftarrow \text{EP}(x_1, x_2) \land \text{letter}[\sigma](x_1, y_1) \land \text{letter}[\sigma](x_2, y_2)$

Accept() $\leftarrow \text{EP}(x_1, x_2) \land \text{letter}[\sigma](x_1, y_1) \land \text{letter}[\sigma'](x_2, y_2)$

$\sigma \neq \sigma'$

$\text{NEP}(x_1, y_2) \leftarrow \text{EP}(x_1, x_2) \land \text{letter}[\sigma](x_2, y_2)$

$\text{NEP}(x_1, y_2) \leftarrow \text{NEP}(x_1, x_2) \land \text{letter}[\sigma](x_2, y_2)$

Accept() $\leftarrow \text{NEP}(x, x)$

Intuition:

- $\text{EP}$ defines equal paths (forwards, from one starting point)
- $\text{NEP}$ defines paths of different length (from one starting point to the same end point)

~ $P_2$ accepts all databases with distinct parallel paths
Solving PCP with Datalog Containment (5)

What does it mean if \( \langle \text{Accept}, P_1 \rangle \) is contained in \( \langle \text{Accept}, P_2 \rangle \)?

The following are equivalent:

- All databases with potential PCP solutions also have distinct parallel paths.
- Databases without distinct parallel paths have no PCP solutions.
- Linear databases (words) have no PCP solutions.
- The answer to the PCP is “no”.

\[ \implies \text{If we could decide Datalog containment, we could decide PCP} \]

Theorem

Containment and equivalence of Datalog queries are undecidable.

(Note that emptiness of Datalog queries is trivial)

Implementing Datalog

FO queries (and thus also CQs and UCQs) are supported by almost all DMBS
\[ \implies \text{many specific implementation and optimisation techniques} \]

How can Datalog queries be answered in practice?
\[ \implies \text{techniques for dealing with recursion in DBMS query answering} \]

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)
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! \(\sim\) 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
\(\sim\) semi-naive evaluation

Naive Evaluation of Datalog Queries

A direct approach for computing \(T_p^{\infty}\)

<table>
<thead>
<tr>
<th>Line</th>
<th>Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>01</td>
<td>(T_p^0 := \emptyset)</td>
</tr>
<tr>
<td>02</td>
<td>(i := 0)</td>
</tr>
<tr>
<td>03</td>
<td>repeat:</td>
</tr>
<tr>
<td>04</td>
<td>(T_p^{i+1} := \emptyset)</td>
</tr>
<tr>
<td>05</td>
<td>for (H \leftarrow B_1 \land \ldots \land B_k \in P: )</td>
</tr>
<tr>
<td>06</td>
<td>(\text{for } \theta \in B_1 \land \ldots \land B_k(T_p) : )</td>
</tr>
<tr>
<td>07</td>
<td>(T_p^{i+1} := T_p^{i+1} \cup {H\theta})</td>
</tr>
<tr>
<td>08</td>
<td>(i := i + 1)</td>
</tr>
<tr>
<td>09</td>
<td>until (T_p^{i+1} = T_p)</td>
</tr>
<tr>
<td>10</td>
<td>return (T_p)</td>
</tr>
</tbody>
</table>

Notaion 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:

\[
\begin{align*}
&\text{e}(1, 2) \quad \text{e}(2, 3) \quad \text{e}(3, 4) \quad \text{e}(4, 5) \\
&(R1) \quad \text{T}(x, y) \leftarrow \text{e}(x, y) \\
&(R2) \quad \text{T}(x, z) \leftarrow \text{T}(x, y) \land \text{T}(y, z)
\end{align*}
\]

How many body matches do we need to iterate over?

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

In total, we considered 37 matches to derive 11 facts

Semi-Naive Evaluation

The computation yields sets \(T_p^0 \subseteq T_p^1 \subseteq T_p^2 \subseteq \ldots \subseteq T_p^o\)
- For an IDB predicate \(R\), let \(R'\) 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 \(R' \setminus R'^{i-1}\)

We can restrict rules to use only some computations.

Some options for the computation in step \(i + 1:\)

\[
\begin{align*}
\text{T}(x, z) &\leftarrow T(x, y) \land T(y, z) \quad \text{same as original rule} \\
\text{T}(x, z) &\leftarrow \Delta_{R}(x, y) \land \Delta_{R}(y, z) \quad \text{restrict to new facts} \\
\text{T}(x, z) &\leftarrow \Delta_{R}(x, y) \land T(y, z) \quad \text{partially restrict to new facts} \\
\text{T}(x, z) &\leftarrow T(x, y) \land \Delta_{R}(y, z) \quad \text{partially restrict to new facts}
\end{align*}
\]

What to chose?
Semi-Naive Evaluation (2)

Inferences that involve new and old facts are necessary:

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

\[(R1) \quad T(x, y) \leftarrow e(x, y) \]

\[(R2) \quad T(x, z) \leftarrow \Delta_i \quad T(x, y) \]

\[ T_p^1 = \emptyset \]

\[ \Delta^1_T = \{T(1, 2), T(2, 3), T(3, 4), T(4, 5)\} \]

\[ \Delta^2_T = \{T(1, 3), T(2, 4), T(3, 5)\} \]

\[ \Delta^3_T = \{T(1, 4), T(2, 5), T(1, 5)\} \]

\[ \Delta^4_T = \emptyset \]

\[ T_p^1 = T_p^0 \]

\[ T_p^2 = T_p^1 \cup \Delta^2_T \]

\[ T_p^3 = T_p^2 \cup \Delta^3_T \]

\[ T_p^4 = T_p^3 = T_p^\infty \]

To derive \( T(1, 4) \) in \( \Delta^3_T \), we need to combine \( T(1, 3) \in \Delta^2_T \) with \( T(3, 4) = \Delta^1_T \) or \( T(1, 2) \in \Delta^1_T \) with \( T(2, 4) \in \Delta^2_T \)

\( \leadsto \) rule \( T(x, z) \leftarrow \Delta^1_T(y, x) \land \Delta^2_T(y, z) \) is not enough

Semi-Naive Evaluation: Example

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

\[(R1) \quad T(x, y) \leftarrow e(x, y) \]

\[(R2.1) \quad T(x, z) \leftarrow \Delta_i \quad T(x, y) \land T(y, z) \]

\[(R2.2') \quad T(x, z) \leftarrow T^{-1}(x, y) \land \Delta^1_T(y, z) \]

How many body matches do we need to iterate over?

\[ T_p^0 = \emptyset \]

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

\[ T_p^2 = T_p^1 \cup \{T(1, 3), T(2, 4), T(3, 5)\} \]

\[ T_p^3 = T_p^2 \cup \{T(1, 4), T(2, 5), T(1, 5)\} \]

\[ T_p^4 = T_p^3 = T_p^\infty \]

In total, we considered 14 matches to derive 11 facts

Semi-Naive Evaluation (3)

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

For example program:

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

\[(R1) \quad T(x, y) \leftarrow e(x, y) \]

\[(R2.1) \quad T(x, z) \leftarrow \Delta_i \quad T(x, y) \land T(y, z) \]

\[(R2.2) \quad T(x, z) \leftarrow T^{-1}(x, y) \land \Delta^1_T(y, z) \]

There is still redundancy here: the matches for \( T(x, z) \leftarrow \Delta^1_T(x, y) \land \Delta^2_T(y, z) \) are covered by both \( (R2.1) \) and \( (R2.2) \)

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

\[(R2.2') \quad T(x, z) \leftarrow T^{-1}(x, y) \land \Delta^1_T(y, z) \]

EDB atoms do not change, so their \( \Delta \) would be \( \emptyset \)

\( \leadsto \) ignore such rules after the first iteration

Semi-Naive: Evaluation: Full Definition

In general, a rule of the form

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

is transformed into \( m \) rules

\[ H(\vec{x}) \leftarrow e_1(\vec{y}_1) \land \ldots \land e_n(\vec{y}_n) \land \Delta^1_I(\vec{z}_1) \land I_1(\vec{z}_2) \land \ldots \land I_{m-1}(\vec{z}_m) \]

\[ H(\vec{x}) \leftarrow e_1(\vec{y}_1) \land \ldots \land e_n(\vec{y}_n) \land I_1^{-1}(\vec{z}_1) \land \Delta^1_I(\vec{z}_2) \land \ldots \land I_{m-1}(\vec{z}_m) \]

\[ \vdots \]

\[ H(\vec{x}) \leftarrow e_1(\vec{y}_1) \land \ldots \land e_n(\vec{y}_n) \land I_1^{-i}(\vec{z}_1) \land \Delta^1_I(\vec{z}_{i+1}) \land \ldots \land I_{m-1}(\vec{z}_m) \]

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

Perfect Datalog optimisation is impossible
  • same situation as for FO queries
  • but for somewhat different reasons

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

Simplest practical bottom-up technique: semi-naive evaluation

Next topics:
  • More on Datalog implementation
  • Further query languages
  • Applications