Knowledge-Based Systems and Deductive Databases

Christoph Lofi
Institut für Informationssysteme
Technische Universität Braunschweig
http://www.ifis.cs.tu-bs.de
6.1 Implementation of Datalog in DBs

6.2 Top-Down-Evaluation
• The **Datalog semantics** are given by **Herbrand interpretations**

  – A Datalog program $\mathcal{P}$ is a set of **Horn clauses**
  – Any Herbrand interpretation that satisfies $\mathcal{P}$ is a **model**
  – Unfortunately, it is not quite that easy to **compute** an Herbrand model for $\mathcal{P}$
  – Also, **multiple models** exists per program – which conveys the **intended semantic**?
• Datalog\textsuperscript{f}
  – Datalog\textsuperscript{f} is computationally complete
  – The **intended semantic** of a Datalog\textsuperscript{f} program is given by the **least Herbrand** model
    • For the least Herbrand model $\mathcal{M}$, $\mathcal{M} \subseteq \mathcal{M}'$ for any other Herbrand model $\mathcal{M}'$ holds
    • This leads to $\mathcal{M} := \bigcap \mathcal{M}$, whereas $\mathcal{M}$ is the set of all Herbrand models
    • Informally: The least model is a model for $\mathcal{P}$ and does not contain superfluous statements
• **Operational semantics for Datalog**
  
  – To compute the least Herbrand model, a **fixpoint iteration approach** can be employed
  
  • Start with an **empty set of ground atoms**
  • Iteratively refine set (by adding more atoms)
  • Fixpoint iteration is **monotonous** (set is only expanded in each iteration)
  • As soon as the fixpoint is reached, set becomes **stable** (i.e. no changes)
  • The method is **finite** for Datalog
  • The stable result is equivalent to the **least Herbrand model**
• Iterative Transformation step:
  – **Elementary production rule** $T_P$
  – Idea: Apply all given rules with premises contained in the set of the previous step
    • For $I_0 = \emptyset$, this puts all atoms into the result
    • For following steps, everything which can be followed by a single application of any rule is added
• **Datalog**\(^{\text{neg}}\) is more difficult
  
  – **Datalog**\(^{\text{neg}}\) does not provide more expressiveness, but allows for more **natural modeling**
  
  – **Problems:**
    
    • **Datalog**\(^{\text{neg}}\) is potentially **unsafe** (i.e. generates infinite or excessively large models)
    
    • **Datalog**\(^{\text{neg}}\) is potentially **ambiguous** (i.e. multiple distinctive models possible)
      
      – **In general, no least Herbrand model possible**
      
      – Instead, multiple minimal Herbrand Models with
        
        \[ \forall M \text{ which are minimal Model: } \exists M' \text{ such that } M' \subset M \]
      
      – **Intersection** of minimal models **is not a model** itself…
• **Datalog**$^\text{neg}$ problems can be addressed by restricting possible programs

  – **Ambiguity**: Assume **negation as failure**
    • A non-provided fact is assumed to be false

  – **Safety**: Enforce **positive grounding**
    • Each variable appearing in a negative clause needs to appear in a positive clause
    • Variable is positively **grounded**
    • Evaluation can thus be restricted to known facts, examination of the whole (potentially infinite) universe not necessary
– These restrictions allow a deterministic choice of models
  • Negative dependencies of ground instances induce a preference on models
  • “Best” model wrt. that preference is called perfect model and is also a minimal model
  • Perfect model is the intended semantics of Datalog\textsuperscript{neg}

– Operative semantics of Datalog\textsuperscript{neg} is given by iterated fixpoint iteration
  • Take advantage of positive grounding and work along program partitions representing the program strata
– For each strata partition, consider only rules which are **positively grounded** in a previous strata
– On the union of those rules and the previous ground instances, apply **normal fixpoint iteration**
  • i.e. iterate a fixpoint iteration along the program strata

• Both **fixpoint iteration** and **iterative fixpoint iteration** are very inefficient
  – Better algorithms in the next lectures….
In the previous week, we have seen the elementary production operator $T_P$

- But how can we put this operator to use?
- Many deductive DBMS do not choose to implement everything “from the scratch”
  - Especially implementations in Prolog and Lisp are very common
- However, for reliably storing huge amounts of data (e.g. the facts in the extensional DB), there is already a wonderful technology: Relational Databases
  - Also, most applications already use RDBs and SQL
In this section, we will map Datalog\textsuperscript{neg} to \textbf{Relational Algebra}.

- This will allow us an implementation of Datalog concepts within a \textbf{RDB}.
- \textbf{Idea:}
  - Take datalog program
  - Translate to relational algebra
  - Evaluate the algebra statement
  - Return results
- Also, this will allow us to take advantage of \textbf{established features of databases}:
  - Query optimization
    - Indexing!
  - ACID properties
  - Load balancing
  - etc…
• When using the **Relational Model** and **Relational Algebra**, we assume the following:
  
  – Data (i.e. facts) is stored in multiple **relations**
  
  – A **relation** $R$ over some sets $D_1, \ldots, D_n$ is a **subset** of their **Cartesian product**
    
    • $R \subseteq D_1 \times \ldots \times D_n$
    
    • The sets $D_1, \ldots, D_n$ are **finite** and are called **domains**
6.1 Relational Algebra

• Relational algebra operations available
  – Base operations of relational algebra
    - Cartesian Product
    - Selection
    - Projection
    - Set Union
    - Set Minus
  – Derived operations
    - Joins: \( R \bowtie S \equiv \sigma_\theta (R \times S) \)
    - Left & Right Semi Joins: \( R \bowtie\bowtie S \equiv \pi_{\text{att}(R)} (R \bowtie S) \)
In the following, we will use variants of normal relational algebra

- **Attribute are referenced** by their number instead by their name, e.g. #1 or #9
- When using **references to relations** in binary operations, e.g. joins, we may also refer to them as [left] or [right]
  - \((R \times S) \bowtie_{[\text{left}].\#3=[\text{right}].\#1} W\)
- We distinguish two types of relational algebra
  - **RelAlg** excluding the set minus operator
  - **RelAlg** including the set minus operator
### 6.1 Relational Algebra

#### Examples:

- **Name of hero with id=1**
  
  \[ \pi_{\#2} \sigma_{\#1=1} (H) \]

- **All powers of hero with id=2**
  
  \[ \pi_{\#5} ((\sigma_{\#1=2} H) \bowtie (H.\#1=HP.\#1) HP \bowtie ([\text{left}.\#2=[\text{right}.\#1] P)) \]

---

<table>
<thead>
<tr>
<th>id</th>
<th>name</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Phoenix</td>
</tr>
<tr>
<td>2</td>
<td>Professor X</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>hid</th>
<th>pid</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>1</td>
<td>3</td>
</tr>
<tr>
<td>1</td>
<td>4</td>
</tr>
</tbody>
</table>
In the following, we will implement a **simple fixpoint iteration** with relational algebra

- We will only consider **safe Datalog\(^{\text{neg}}\) programs**, i.e. negative literals and head variables are positively grounded

Given is a safe Datalog\(^{\text{neg}}\) program \(\mathcal{P}\) and a relational database

- **Task:**
  - Store extensional DB in tables
  - Encode intensional DB in a customized relational algebra **elementary production operator**
6.1 Implementation

- Each **predicate symbol** \( r_1, \ldots, r_m \) of the **extensional database** is assigned to a relation \( R_1, \ldots, R_m \)
  - i.e. those predicates provide the **facts**, each fact has its own relation

- Each **predicate symbol** \( q_1, \ldots, q_m \) symbol of the **intensional database** is assigned to a relation \( Q_1, \ldots, Q_m \)
  - i.e. those predicates are defined by **rules**

- For ease of use, we restrict each predicate to be defined either in the **intensional** or the **extensional DB**
  - i.e. each predicate which was used to **define facts** is **not allowed** to occur in the **head of a rule**
  - This does not limit the expressiveness of Datalog programs
The predicate symbols $<, >, \leq, \geq, =, \neq$ are assigned to the hypothetical relations $H := \{LT, GT, LTE, GTE, EQ, NEQ\}$

- Those relations are of infinite size and thus, of course, not stored in the RDB.
- We will see later that they can be removed.
6.1 Implementation

• Just a short consideration:
How could we map relational algebra to Datalog?

– $\sigma_{#2=5} R$  $\Rightarrow$  $R(X, 5)$.
– $\pi_{#1} R$  $\Rightarrow$  $R'(X) :- R(X, Y)$.
– $R \times S$  $\Rightarrow$  $RS(W, X, Y, Z) :- R(W, X), S(Y, Z)$.
– $R \bowtie_{[\text{left}. #1=\text{right}. #2]} S$  $\Rightarrow$
  $RS(W, X, Y, Z) :- R(W, X), S(Y, Z), W=Z$.
– $R \bowtie_{[\text{left}. #1=\text{right}. #2]} S$  $\Rightarrow$
  $RS(W, X) :- R(W, X), S(Y, Z), W=Z$.
– $R \cup S$  $\Rightarrow$
  $R'(X, Y) :- R(X,Y)$.
  $R'(X, Y) :- S(X,Y)$.
– $R \setminus S$  $\Rightarrow$
  $R'(X, Y) :- R(X, Y), \neg S(X, Y)$.
6.1 Implementation

• Now, how can we translate from Datalog to relational algebra
  – Some pre-processing is necessary!
• Transform all rules of the intensional DB such that the head contains only variables
  – This can be achieved by replacing any head constant with a new variable and adding a literal binding that variable to the old value
  – e.g. \( q(X, a, b) :- L_1, ..., L_n \)
    \[ \Rightarrow q(X, Y, Z) :- L_1, ..., L_n, Y=a, Z=b \]
• Change the order of the variables such that their safety is ensured by previous body literals

  – A literal is **unsafe**, if it is potentially **infinite**
  
  – e.g., $R(X,Y) :\rightarrow X = Y, p(X), q(Y)$ is not in correct order as the safety $X = Y$ is not ensured by previous literals
    
    • There are infinite possibilities for $X$ being equal to $Y$
  
  – $\Rightarrow R(X,Y) :\rightarrow p(X), q(Y), X = Y$
    
    • is in correct order as $p(X)$ and $q(Y)$ limit the possible values of $X$ and $Y$
  
  – We also sort positive literals before negative ones
    
    • …for positive grounding
6.1 Implementation

• Each rule $R ::= L_1, ..., L_n$ is now transformed to relational algebra as follows
  
  – For each literal $L_1, ..., L_n$, the respective atomic component $A_i \equiv p_i(t_1, ..., t_m)$ is transformed into an relational expression $E_i$
    
    • $E_i \equiv \sigma_{\theta}(P_i)$ with $P_i$ being the relation corresponding to $p_i$
    
    • The selection criterion $\theta$ is a conjunction of conditions defined as follows:
      For each $t_i$, a condition is added
      
      – $#j = t_j$ if $t_j$ is a constant symbol
      
      – $#j = #k$ if $t_j$ and $t_k$ are the same variables
6.1 Implementation

– Example:

\[\begin{align*}
\text{p}(X, 2) & : q(X, X, Y, 2), r(X, 1) \implies \text{(Replace constants)} \\
\text{p}(X, Z) & : q(X, X, Y, 2), r(X, 1), Z = 2 \implies \text{(Translate to R-Alg)} \\
E_1 & := \sigma(#1 = #2 \land #4 = 2) Q \\
E_2 & := \sigma(#2 = 1) R \\
E_3 & := \sigma(#2 = 2) EQ
\end{align*}\]

– After treating the single literals, we will compose the **body expression** \( F \) from left to right

– Initialize the temporary expression \( F_1 := E_1 \)
• Depending on the variables in the literals, the following expressions \( F_2 - F_k \) are generated differently:

- \( F_i := F_{i-1} \times E_i \) iff \( L_i \) does not contain any variables of the previous body literals, i.e. \( \text{vars}(L_i) \cap \text{vars}({L_1, ..., L_{i-1}}) = \emptyset \)

• \( R(X, Y, Z) :- q(X, 2), r(Y), Z=3 \)  \( \Rightarrow \)
  \[ E_1 := F_1 = \sigma_{(#2=2)} Q ; \ E_2 = R ; \ E_3 = \sigma_{(#1=3)} EQ \]  \( \Rightarrow \)
  \[ F_2 := (\sigma_{(#2=2)} Q) \times R ; \ F_3 := (\sigma_{(#2=2)} Q) \times R \times \sigma_{(#1=3)} EQ \]

• In short: Conjunctions of unrelated literals result to computing the **Cartesian Product**
– \( F_i := F_{i-1} \bowtie_{\theta} E_i \) iff \( L_i \) is **positive** and shares variables with previous body literals

- \( \theta \) forces the columns representing the shared variables to be equal

- \( R(X, Y) :- q(3, X), r(Y), X < Y \Rightarrow \)
  
  \( E_1 := F_1 = \sigma_{(#1=3)} Q; E_2 = R; E_3 = LT; \Rightarrow \)
  
  \( F_2 := \sigma_{(#1=3)} Q \times R; \)
  
  \( F_3 := (\sigma_{(#1=3)} Q \times R) \bowtie ([\text{left}].#2 = [\text{right}].#1 \land [\text{left}].#3=[\text{right}].#2 ) LT; \)

- In short: Conjunctions of **related positive literals** result in generating a **join**, using the related variables as join condition
6.1 Implementation

- \( F_i := F_{i-1} \setminus (F_{i-1} \bowtie_\theta E_i) \) iff \( L_i \) is negative and shares variables with previous body literals.

- \( \theta \) forces the columns representing the shared variables to be equal

- \( R(X) :- q(X), \neg r(X) \Rightarrow E_1 := F_1 = Q, \quad E_2 = R \Rightarrow F_2 := Q \setminus (Q \bowtie_{(Q.#1 = R.#1)} R) \)

- In short: Conjunctions of related negative literals result to generating a set-minus, removing those tuples which are related to the negative literal
6.1 Implementation

- Now, we still have the **infinite hypothetical relations** $\mathcal{H} := \{\text{LT, GT, LTE, GTE, EQ, NEQ}\}$ in our expressions

- Each join $E \bowtie_\theta H_i$ or **Cartesian product** $E \times H_i$ for any “normal” expression $E$ and $H_i \in \mathcal{H}$ is replaced by a suitable expression of the form $\pi(\sigma(E))$, e.g.

  - $E \bowtie_{E.\#1=\text{LT}.\#1 \land E.\#2=\text{LT}.\#2} \text{LT} \Rightarrow \sigma_{\#1<\#2}(E)$
    - This expression was created by, e.g.: $E(X, Y, ...), X < Y$

  - $E \bowtie_{E.\#1=\text{EQ}.\#1} \text{EQ} \Rightarrow \pi_{\text{attributesOf}(E), \text{EQ}.\#1}(E)$
    - This expression was created by, e.g.: $E(X, ..., X = Y)$
6.1 Implementation

• $E \times (\sigma_{\#2=c} \ E \ Q) \ \Rightarrow \ \pi_{\text{attributesOf}(E), \ c}(E)$
  - This expression was created by, e.g.: $E(\ldots), \ X=c$

• Examples:
  - $R(X, Y) :\neg q(3, X), \ r(Y), \ X<Y \ \Rightarrow$
  - $F := (\sigma_{(#1=3)} Q \times R) \bowtie ([\text{left}].\#2=[\text{right}].\#1 \land [\text{left}].\#3=[\text{right}].\#2)$
  - $F = \sigma_{\#2<\#3} (\sigma_{(#1=3)} Q \times R)$
  - By algebraic optimization, this will later result to
    • $F = (\sigma_{(#1=3)} Q) \bowtie \#2<\#3 \ R$
6.1 Implementation

- Finally, the whole rule \( C \equiv R : L_1, \ldots, L_n \) is now transformed to the expression \( \text{eval}(C) := \pi_{\text{head}(R)}(F) \)
  - i.e. to evaluate the rule \( C \), we project all variables appearing in its head from its body expression \( F \)

- For evaluating one iteration step for given intensional predicate \( q_i \), all related rules have to be united
  - \( \text{eval}(q_i) := \bigcup_{C \in \text{def}(q_i)} (C) \)
6.1 Implementation

• Now, the elementary production rule $T_p$ corresponds to evaluating all $\text{eval}(q_i)$

• Queries $Q \equiv p(t_1, ..., t_n)$ can be transformed to relational algebra likewise

• Also note that Datalog can be translated to RelAlg$^+$ while Datalog$^{\text{neg}}$ has to be translated to full RelAlg
  – **Negation** requires the highly inefficient $\setminus$ operator
6.1 Implementation

• For actually performing the **fixpoint iteration**, the following is performed
  
  1. Create tables for each intensional predicate $q_i$
  2. **Execute the elementary production** $T_P$ (i.e. run $\text{eval}(q_i)$ for each intensional predicate) and **store results temporarily**
     
     a. If result tables are of the **same size** as the predicate tables, the **fixpoint** has been reached and we can continue with step 3
     
     b. **Replace** content of intensional predicate tables with respective temporary tables
     
     c. Continue with step 2
  3. Run the actual **query** on the tables to obtain **final result**
6.1 Implementation

• Example
  – edge(1, 2). edge(1, 3). edge(2, 4).
    edge(3, 4). edge(4, 5).
  – path(X, Y) :- edge(X, Y).
  – path(X, Y) :- edge(X, Z), path(Z, Y).
  – path(2, X)?
  – The facts all go into the extensional table Edge, an intensional table Path is created.
6.1 Implementation

- \text{path}(X, Y) :- \text{edge}(X, Y).
  
  \begin{itemize}
  \item F := \pi_{\#1, \#2} \sigma_{\text{true}} \text{Edge} \\
  \hspace{1cm} = \text{Edge}
  \end{itemize}

- \text{path}(X, Y) :- \text{edge}(X, Z), \text{path}(Z, Y).
  
  \begin{itemize}
  \item F := \pi_{\#1, \#2} (\sigma_{\text{true}} \text{Edge} \bowtie_{[\text{left}].\#2=[\text{right}].\#1} \sigma_{\text{true}} \text{Path}) \\
  \hspace{1cm} = \text{Edge} \bowtie_{[\text{left}].\#2=[\text{right}].\#1} \text{Path}
  \end{itemize}

- \text{path}(2, X) = \text{path}(Y, X), Y=2
  
  \begin{itemize}
  \item F := \sigma_{\#1=2} \text{Path}
  \end{itemize}

- \text{eval(path)} := \text{Edge} \cup \text{Edge} \bowtie_{[\text{left}].\#2=[\text{right}].\#1} \text{Path}
6.1 Implementation

- Execute elementary production on current tables
  
  \[
  \text{eval}(\text{path}) := \text{Edge} \cup \text{Edge} \ni [\text{left}.#2=[\text{right}.#1] \cdot \text{Path}
  \]

<table>
<thead>
<tr>
<th>edge</th>
<th>#1</th>
<th>#2</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>3</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>4</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>4</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>5</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>path</th>
<th>#1</th>
<th>#2</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>temp_{path}</th>
<th>#1</th>
<th>#2</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>3</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>4</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>4</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>5</td>
<td></td>
</tr>
</tbody>
</table>
6.1 Implementation

- Replace path table and repeat

\[ \text{eval}(\text{path}) := \text{Edge} \cup \text{Edge} \bowtie_{[\text{left}].#2 = [\text{right}].#1} \text{Path} \]

<table>
<thead>
<tr>
<th>edge</th>
<th>#1</th>
<th>#2</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>3</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>4</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>4</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>5</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>path</th>
<th>#1</th>
<th>#2</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>3</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>4</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>4</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>5</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>temp path</th>
<th>#1</th>
<th>#2</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>3</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>4</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>4</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>5</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>4</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>5</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>5</td>
<td></td>
</tr>
</tbody>
</table>
6.1 Implementation

- Replace path table and repeat

\[ \text{eval}(\text{path}) := \text{Edge} \cup \text{Edge} \bowtie [\text{left}.#2=\text{right}.#1] \text{Path} \]
6.1 Implementation

- Replace path table and repeat
  - No change – fixpoint is reached

<table>
<thead>
<tr>
<th>edge</th>
<th>#1</th>
<th>#2</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>3</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>4</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>4</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>5</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>path</th>
<th>#1</th>
<th>#2</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>3</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>4</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>4</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>5</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>4</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>5</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>5</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>5</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>temp</th>
<th>#1</th>
<th>#2</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>3</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>4</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>4</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>5</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>4</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>5</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>5</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>5</td>
<td></td>
</tr>
</tbody>
</table>
6.1 Implementation

• Run query to obtain final result

\[ \sigma_{#1=2} \text{Path} \]

<table>
<thead>
<tr>
<th>edge</th>
<th>#1</th>
<th>#2</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>3</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>4</td>
</tr>
<tr>
<td>3</td>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td>4</td>
<td>4</td>
<td>5</td>
</tr>
</tbody>
</table>

path

<table>
<thead>
<tr>
<th>path</th>
<th>#1</th>
<th>#2</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>3</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>4</td>
</tr>
<tr>
<td>3</td>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td>4</td>
<td>4</td>
<td>5</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>4</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>5</td>
</tr>
<tr>
<td>3</td>
<td>3</td>
<td>5</td>
</tr>
</tbody>
</table>

result

<table>
<thead>
<tr>
<th>#1</th>
<th>#2</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>4</td>
</tr>
<tr>
<td>2</td>
<td>5</td>
</tr>
</tbody>
</table>
6.2 Strategies

• Given an extensional database and a query, there are two general strategies for evaluation
  – **Bottom-Up**: Start with given facts in the EDB and generate all new facts. Then discard those which don’t match the query
    • e.g. fixpoint iteration
    • Performs well in restricted and smaller scenarios
    • “forward-chaining”
6.2 Strategies

- **Top-Down**: Start with query and generate proofs down to the EDB facts
  - Most logical programming environments choose this approach
    - e.g. **SDL resolution**
  - Performs well in more complex scenarios where bottom-up becomes prohibitive
  - “backward-chaining”
6.2 Strategies

• **Scenario**
  - All **facts** are contained in extensional database EDB
  - All **rules** are contained in the Datalog program $\mathcal{P}$
    - No facts in $\mathcal{P}$
  - Given is a **goal query** $Q \equiv p(t_1, ..., t_n)$?

• **Bottom-up problems**
  - Generate all deducible facts of $\mathcal{P} \cup \text{EDB}$
  - When finished, **throw away** all facts not matching the query pattern. Especially:
    - All those facts whose **predicate is not** $p$
    - All those facts whose predicate is $p$, but are **more general than** the query
6.2 Strategies

- Example with constants:
  - $Q \equiv p(a, X, b)$?
  - Why should we generate all facts of $p$ and later discard those which are not subsumed by $Q$?

- In the next lecture, we will explore bottom-up approaches which avoid generating unnecessary facts
  - Magic Sets
  - Counting techniques

- Today, we start with a simple top-down approach
6.2 Top-Down Evaluation

• Basic Idea:
  – Start with the query $Q \equiv p(t_1, ..., t_n)$?
  – Iteratively generate all proof trees ending with a ground instance of $Q$ and starting with known facts
    • Iterate over tree depth
    • As a helper data structure create all possible search trees of current depth
    • Transform search trees to all possible proof trees
    • Stop if no additional search trees / proof trees can be constructed
– A **search tree** is a **generic proof tree** which is still parameterized to some extent

- Proof trees can be **generated** from search trees
- Leaf nodes are called **subgoal nodes**
- Root node is called **goal node**
Example:

- $e(1, 2). e(1, 3). e(2, 4). e(3, 4). e(4, 5). e(5, 6). e(5,7)$
- $p(X, Y) :- e(X, Y)$.  
- $p(X, Y) :- e(X, Z), p(Z, Y)$.  
- $Q \equiv p(2, X)$

Rule 1

Rule 2
6.1 Top-Down Evaluation

• **Proof Trees of depth 0**
  – Which facts are *ground instances* of \( Q \)?
  – In our example, this is not the case for any fact…

• **Search Trees of depth 1**
  – Find all rules \( R \equiv B : - A_1, \ldots, A_k \) such that \( Q \) and \( B \) are *unifiable*
    • **Unifiable**: There are substitutions such that \( B \) matches \( Q \)
  – For each rule \( R \), construct a *search tree* with \( Q \) as root
    • Attach a *rule node* to \( Q \) containing \( R \)
    • Attach \( k \) *subgoal nodes* representing \( A_1, \ldots, A_k \) in its unified form
• **Search Trees of depth 1**
  – Rule 1: \( p(X, Y) :- e(X, Y) \).
    
    \[
    T1\quad Q \equiv p(2, X)
    \]
    
    \[
    p(Y, X) :- e(Y, X). \quad e(2, X)
    \]
  – Rule 2: \( p(X, Y) :- e(X, Z), p(Z, Y) \).
    
    \[
    T2\quad Q \equiv p(2, X)
    \]
    
    \[
    p(Y, X) :- e(Y, Z), p(Z, X). \quad e(2, Z) \quad p(Z, X)
    \]
To generate proof trees from a given search tree, we have to find a substitution $\rho$ such that for each goal node with clause $C$, $\rho(C) \in \mathcal{P} \cup \text{EDB}$

- By applying this substitution to the whole tree, we obtain a proof tree
- The root node is a result of the query

Example:
- Find a substitution for $T_1$ ($T_2$ does not have one)
6.2 Top-Down Evaluation

• For any $n > 1$, all existing search trees of depth $n-1$ are expanded by treating any subgoal node as a goal node
  – Thus, new rule nodes and subgoals are appended

• Example: Expanding $T_2$ to $T_{2,2}$ and $T_{2,1}$

![Diagram showing the expansion of $T_2$ to $T_{2,2}$ and $T_{2,1}$]
6.2 Top-Down Evaluation

- \textbf{T2,1} and some substitutions \( \rho \)

\[ Q \equiv p(2, X) \]

\[ p(Y, X) :- e(Y, Z), p(Z, X). \]

\[ e(2, Z) \]

\[ p(Z, X) :- e(Z, X). \]

\[ e(Z, X) \]

\[ p(2, 5) \]

\[ p(Y, X) :- e(Y, Z), p(Z, X). \]

\[ e(2, 4) \]

\[ p(4, 5) \]

\[ p(Z, X) :- e(Z, X). \]

\[ e(4, 5) \]

\[ \rho := \{ Z = 4 \ X = 5 \} \]
6.2 Top-Down Evaluation

- T2,2,1 and substitutions $\rho_1$ and $\rho_2$

\[ T_{2,2,1} \]

\[ Q \equiv p(2, X) \]

\[ p(Y, X) : e(Y, Z), p(Z, X). \]

\[ e(2, Z) \quad p(Z, X) \]

\[ p(Z, X) : e(Z, W), p(W, X). \]

\[ e(Z, W) \quad p(W, X) \]

\[ p(W, X) : e(W, X). \]

\[ e(W, X) \]

$\rho_1 := \{Z = 4, W=5, X = 6\}$

\[ P_{2,2,1}(1) \]

\[ p(2, 6) \]

\[ \ldots \]

$\rho_2 := \{Z = 4, W=5, X = 7\}$

\[ P_{2,2,1}(2) \]

\[ p(2, 7) \]

\[ \ldots \]
6.2 Top-Down Evaluation

• Please note:
  – By applying this type of backward-chaining, not all possible proof trees for the query can be generated
  – Only proof trees having **ground facts in all leaf nodes** are possible
    • Those trees are called **full proof trees**
    • However, for each proof tree matching the query, there is also a respective **full proof tree**
We can see that the backward chaining proof trees can reach arbitrary depth

- The backward chaining method is sound and complete
- But consider the iterated use of rule 2

- The tree is of infinite depth
• When do we stop building trees?
  – A-priory, we have no idea which recursion depth we will need
    • ?path\( (a,X) \)
    • Obviously, the more nodes we have, the deeper the recursion depth will be
  – Still the number of sensible combinations of EDB facts and predicates in \( \mathcal{P} \) is limited since
    • Both the database and the datalog program are finite
    • We can only substitute any constant symbol from some fact in any predicate symbol at any position of a variable
• Theorem: **Backwards chaining remains complete**, if the search depth is limited to 

\[#\text{predicates} \times \#\text{constants}^{\text{max(args)}}\]

– #predicates is the number of predicate symbols used 
– #constants is the number of constant symbols used 
– max(args) is the maximum number of arguments, i.e. the arity, of all predicate symbols 

– With this theorem, we can stop the backward chaining process after the **last sensible production**
6.2 Top-Down Evaluation

• Proof sketch:
  – \#\text{predicates} \times \#\text{constants}^{\max(\text{args})} \text{ is an upper limit for the number of distinct ground facts derived from } P \text{ and EDB (purely syntactical)}
  – We can limit the production process to full proof trees, where at least one new fact is added in each depth level (otherwise the new level is useless…)
  – Since we only have a limited number of ground facts, also the number of levels has to be limited…
• Consider an example: a finite number of facts \{\text{path}(a,b), \text{path}(b,c),..., \text{path}(m,n)\} and a rule \text{path}(X,Y) :- \text{path}(X,Z), \text{path}(Z,Y).

– Worst case

• Longest possible deduction chain is \text{path}(a,n) of length \(n-1\)

– The least determined query is \(?\text{path}(X,Y)\), i.e. all paths

• There are \(n\) constant symbols and a single predicate symbol
• The constants can occur in two places, i.e. \(\max(\text{args}) = 2\)
• That means the maximum number of deducible facts is \(n^2\)
Many backward-chaining algorithms rely on the concepts of search trees and proof trees.

However, the generation strategy may differ.

In the previous example, the search trees have been generated one by one according to their depth:

- Depth 0, depth 1, depth 2, …
- This is called level saturation strategy and resembles an breadth-first approach.

Alternatively, depth-first approaches are possible:

- Rule saturation strategy
6.2 Resolution

• The previously presented top-down algorithm is extremely naïve
  – It generates all possible search and proof trees up to the worst-case depth which are somehow related to the query
    • Performance is far from optimal
  – In case of less restricted scenarios (e.g. not only Horn clauses or infinite universes), this approach is inevitably doomed to failure
6.2 Resolution

• From the field of “real” logics, we can borrow the concept of resolution
  – A technique for refutation theorem proofing
  • “Reductio ad absurdum”

– Mainly explored around 1965 by J.A. Robinson

– Established itself as THE standard technique for logical symbolic computation
There are several variants of resolution

– Best known in the field of logical programming is the class of **SDL resolution** algorithms
  
  • “Linear Resolution with **Selection Function for Definite Clauses**”
  
  • Most popular among these are the general algorithms employed in languages like **Prolog** or **Lisp**
  
  • However, in the next lecture we shall study a simplified SDL resolution algorithm suitable for **Datalog**
    
    – Be curious – that will be fun!
6.3. Recursive SQL

• The research and developments in the area of deductive databases successfully provided the ability to perform recursive queries
  – And with these, some limited reasoning capabilities

• However, most applications have been tailored to work with traditional SQL based databases
  – When using SQL2 (SQL-92), recursive queries cannot be facilitated without external control and huge performance penalties
  – SQL2 is still the default for most today’s databases
6.3. Recursive SQL

- **SQL3 (SQL-99)** is a later SQL standard which mainly aims at widening the scope of SQL
  - Contains many features which extend beyond the scope of traditional RDBs
    - Binary Large Objects
    - Limited support for soft constraints
    - Updatable views
    - Active databases
    - Object orientation
    - UDF / UDT / UDM
    - References
    - **Recursive Temporary Tables**
6.3. Recursive SQL

• Recursive temporary tables adopt many concepts of deductive databases into the SQL world
  – Most vendors developed proprietary implementations of recursive tables
    • Nobody cared for the standard…
    • Syntax may thus differ
  – In DB2 known as Common Table Expressions
6.3. Recursive SQL

• Main idea:
  – Predicates are represented by temporary tables
  – Usually, definition of temporary table consists of two parts which are united via the union operator
    • Base case: Represents the extensional part of the predicates (i.e. known facts which are read from the database)
    • Recursive step: The intentional part encoding the rules
6.3. Recursive SQL

• Common table expressions begin with the `WITH` keyword
  – Two variants:
    • Just `WITH`: Only base definition *without recursion*. Resembles more a less a normal temporary view.
    • `WITH RECURSION`: Additionally allows a *recursive* definition
      – At least the standard defines it this way, most DB vendors don’t care…
  – **Multiple** temporary recursive tables may be defined in one `WITH` statement
  – You can also use the `WITH` statement for *view definitions* or within `INSERT`, `DELETE` or `UPDATE` statements
• Example: Paths in a graph
  – Prepare the edges (facts)
  – Datalog

edge(1,2). edge(1,3). edge(2,4). edge(3,4).
  edge(4,5). edge(5,6). edge(5,7)

– SQL3 equivalent

CREATE TABLE edge (x int, y int);
INSERT INTO edge VALUES (1,2),(1,3),
  (2,4), (3,4), (4,5), (5,6), (5,7);
6.3. Recursive SQL

• Create a non-recursive view
  – and query all paths from 5
    • … which is quite boring

  – **Datalog & SQL3**

```sql
WITH path (x, y) AS ( SELECT x, y FROM edge )
SELECT x, y FROM path WHERE x=5
```

• In this case, the **WITH** statement just creates a named, **temporal view** which can be used by the directly following select query

```prolog
path(X, Y) :- edge(X, Y).
path(5, Y)?
```
6.3. Recursive SQL

– This could also **easily be done in SQL2**
  
  • `SELECT x, y FROM (SELECT x, y FROM edge) WHERE x=5`

– However, CTE allow for a more flexible **reuse of temporal views**
  
  • `SELECT x, y FROM (SELECT x, y FROM edge) WHERE y=2`  
    `UNION`  
    `SELECT x, y FROM (SELECT x, y FROM edge) WHERE y=3`

– v.s.
  
  • `WITH path (x, y) AS (SELECT x, y FROM edge)`  
    `SELECT x, y FROM path WHERE y=3`  
    `UNION`  
    `SELECT x, y FROM path WHERE y=2`

– However, nothing overly exciting yet…
6.3. Recursive SQL

- Create a recursive view
  - and query it
- **Datalog & SQL3**

\[
\text{path}(X, Y) :\text{-} \text{edge}(X, Y).
\]

\[
\text{path}(X, Y) :\text{-} \text{edge}(X, Z), \text{path}(Z, Y)
\]

\[
\text{path}(4, Y)?
\]

**WITH** \( \text{path}(x, y) \) **AS** (  
  **SELECT** \( x, y \) **FROM** \( \text{edge} \) **UNION ALL**  
  **SELECT** \( e.x, p.y \) **FROM** \( \text{edge} e, \text{path} p \) **WHERE** \( e.y=p.x \)  
  **SELECT** \( x, y \) **FROM** \( \text{path} \) **WHERE** \( x=4 \);  
)

DB2 Syntax!
• **Linear & Non-Linear Recursion**

  – The SQL3 standard only specifies **linear recursion**
    
    • i.e. a recursive step definition may refer to its own recursive table only **once**
    
    • e.g. `WITH path (x, y) AS (...
    UNION ALL
    SELECT e.x, p.y FROM edge e, path p WHERE e.y=p.x)`

  – However, a DB vendor may decide to additionally support **non-linear recursion**
    
    • Fixpoint during evaluation **may be reached faster**
    
    • Evaluation **more complex** in general
    
    • e.g. `WITH path (x, y) AS (...
    UNION ALL
    SELECT p1.x, p2.y FROM path p1, path p2 WHERE p1.y=p2.x)`
6.3. Recursive SQL

• Common table expressions also support negation
  – However, restrictions similar to Datalog apply
    • Statement must be stratified
    • Negative references to tables must be positively grounded

\[ \text{toll}(1, 2). \]

\[
\text{CREATE TABLE} \ \text{toll} \ (x \ \text{int}, \ y \ \text{int}): \\
\text{INSERT INTO} \ \text{toll} \ \text{VALUES} \ (1, 2);
\]
• Example of negation

```prolog
....
goodpath(X, Y) :- edge(X, Y), ~toll(X).
goodpath(X, Z) :- goodpath(X, Y), goodpath(Y, Z).

goodpath(1, X)?
```
6.3. Recursive SQL

WITH

path (x, y) AS (  
SELECT x,y FROM edge  
UNION ALL  
SELECT e.x, p.y FROM edge e, path p WHERE  
e.y=p.x),  
goodpath (x, y) AS (  
SELECT x, y FROM edge e WHERE  
NOT EXISTS  
(SELECT t.x, t.y FROM toll t WHERE  
t.x=e.x  
AND t.y=e.y)  
UNION ALL  
SELECT p1.x, p2.y FROM goodpath p1, goodpath p2  
WHERE p1.y=p2.x)  
SELECT 1, y FROM goodpath

Careful: This is not linear (e.g. won’t work in DB2)
Next lecture

• More implementation and optimization techniques
  – Magic Sets
  – SDL resolution
  – Further optimization