Knowledge-Based Systems and Deductive Databases

Wolf-Tilo Balke
Christoph Lofi
Institut für Informationssysteme
Technische Universität Braunschweig
http://www.ifis.cs.tu-bs.de
• More implementation and optimization techniques
  – Design Space
  – Delta Iteration
  – Logical Rewriting
  – Magic Sets
• **Datalog** can be converted to **Relational Algebra** and vice versa
  – This allows to **merge** Datalog-style reasoning techniques with relational databases
    • e.g. Datalog on RDBs, Recursive SQL, etc.
  – The **elementary production rule** (and thus the fixpoint iteration) has been implemented with relational algebra in the last lecture
• In addition to **bottom-up** approaches (like fix-point iteration), there are also **top-down** evaluation schemes for Datalog

  – Idea: Start with query and try to construct a proof tree down to the facts

  – Simple Bottom Up approach: Construct all possible search trees by their depth

  • **Search tree**: Parameterized **proof tree**

    – Search tree can be transformed to a proof tree by providing a valid substitution
– Search tree are constructed by **backwards-chaining** of rules

– Problem: **When to stop?**
  - A naïve solution: Compute the theoretical maximal chain length and use as limit

– Outlook for today: **Optimization techniques**
  - Evaluation optimization
  - Query rewriting
Exercise 2

• Fixpoint iteration
  – path(X,Y) :- edge(X,Y)
  – path(X,Y) :- edge(X,Y), path(Z,Y)

New facts added by proof tree length!
Exercise 3.2

- **Stratification**
  - `q(1,2)`
  - `q(2,3)`
  - `q(1,3)`
  - `r(X,Y) :- s(X,Y)`
  - `p(X,Y) :- q(X,Y), ¬r(X,Y)`
  - `p(X,Y) :- q(X,Y), ¬s(X,Y)`
  - `p(X,Y) :- p(X,Y), p(X,Y)`

\[
S1 := \{\text{def}(q), \text{def}(s), \text{def}(r)\}
\]
\[
S2 := \{\text{def}(p)\}
\]
Exercise 3.5

- Translate Datalog\textsuperscript{neg} to Datalog. Idea:
  - Use closed world assumption
    - Constants: 1, 2, 3
  - Introduce new predicates
    - ns(X,Y)=false for X=1, Y=3, true otherwise
    - nr(X,Y):-ns(X,Y)

\begin{align*}
q(1,2) \\
q(2,3) \\
s(1,3) \\
r(X,Y):-s(X,Y) \\
p(X,Y):-q(X,Y),\neg r(X,Y) \\
p(X,Y):-q(X,Y),\neg s(X,Y) \\
p(X,Y):-p(X,Y),p(X,Y)
\end{align*}

\begin{align*}
q(1,2) \\
q(2,3) \\
s(1,3) \\
r(X,Y):-s(X,Y) \\
p(X,Y):-q(X,Y),nr(X,Y) \\
p(X,Y):-q(X,Y),ns(X,Y) \\
p(X,Y):-p(X,Y),p(X,Y)
\end{align*}
8.1 Query Optimization

• The computation algorithms introduced in the previous weeks were all far from optimal
  – Usually, a lot of unnecessary deductions were performed
  – Wasted work
  – Termination problems, etc…

• Thus, this week we will focus on optimization methods
8.1 Query Optimization

- Optimization and evaluation methods can be classified along several criterions
  - Search technique
  - Formalism
  - Objective
  - Traversal Order
  - Approach
  - Structure
8.1 Query Optimization

• **Search Technique:**
  
  – **Bottom-Up**
    
    • Start with extensional database and use *forward-chaining* of rules to generate new facts
    • Result is subset of all generated facts
    • **Set oriented-approach** → Very well-suited for databases
  
  – **Top-Down**
    
    • Start with queries and either construct a proof tree or a refutation proof by *backward-chaining* of rules
    • Result is generated **tuple-by-tuple** → More suited for complex languages, but less desirable for use within a database
Furthermore, there are two possible (non-exclusive) formalisms for query optimization

- **Logical**: A Datalog program is treated as *logical rules*
  - The predicates in the rules are connected to the *query predicate*
  - Some of the variables may already be *bound* by the query

- **Algebraic**: The rules in a Datalog program can be translated into *algebraic expressions*
  - Thus, the IDB corresponds to a *system of algebraic equations*
  - Transformations like in normal *database query optimization* may apply
8.1 Query Optimization

• Optimizations can address different objectives

  – **Program Rewriting:**
    
    • Given a specific evaluation algorithm, the Datalog program \( \mathcal{P} \) is rewritten into a semantically equivalent program \( \mathcal{P}' \)
    
    • However, the new program \( \mathcal{P} \) can be executed much faster than \( \mathcal{P} \) using the same evaluation method

  – **Evaluation Optimization:**
    
    • Improve the process of evaluation itself, i.e. program stays as it is but the evaluation algorithm is improved
    
    • Can be combined with program rewriting for even increased effect
8.1 Query Optimization

- Optimizations can focus on different traversal-orders
  - **Depth-First**
    - Order of the literals in the body of a rule may affect performance
      - e.g. consider top-down evaluation with search trees for
        \( P(X,Y) :- P(X,Z), Q(Z,Y) \) vs. \( P(X,Y) :- Q(Z,Y), P(X,Z) \)
      - In more general cases (e.g. Prolog), may even affect decidability
    - It may be possible to quickly produce the first answer
  - **Breadth-First**
    - Whole right hand-side of rules is evaluated at the same time
    - Search trees grow more balanced
    - Due to the restrictions in Datalog, this becomes a set-oriented operation and is thus very suitable for DB’s
When optimizing, two approaches are possible

– **Syntactic**: just focus on the syntax of rules  
  • Easier and thus more popular than semantics  
  • e.g. restrict variables based on the goal structure or use special evaluation if all rules are linear, etc.

– **Semantic**: utilize external knowledge during evaluation  
  • E.g., integrity constraints  
  • External constraints: “Lufthansa flights arrive at Terminal 1” 
    Query: “Where does the flight LH1243 arrive?”
8.1 Query Optimization

- **Summary** of optimization classification with their (not necessarily exclusive) alternatives

<table>
<thead>
<tr>
<th>Criterion</th>
<th>Alternatives</th>
</tr>
</thead>
<tbody>
<tr>
<td>Search technique</td>
<td>bottom-up</td>
</tr>
<tr>
<td></td>
<td>top-down</td>
</tr>
<tr>
<td>Formalism</td>
<td>logic</td>
</tr>
<tr>
<td></td>
<td>relational algebra</td>
</tr>
<tr>
<td>Objective</td>
<td>rewriting</td>
</tr>
<tr>
<td></td>
<td>pure evaluation</td>
</tr>
<tr>
<td>Traversal order</td>
<td>depth-first</td>
</tr>
<tr>
<td></td>
<td>breadth-first</td>
</tr>
<tr>
<td>Approach</td>
<td>syntactic</td>
</tr>
<tr>
<td></td>
<td>semantic</td>
</tr>
<tr>
<td>Structure</td>
<td>rule structure</td>
</tr>
<tr>
<td></td>
<td>goal structure</td>
</tr>
</tbody>
</table>
8.1 Query Optimization

- Not all combinations are feasible or sensible
  - We will focus on following combinations

<table>
<thead>
<tr>
<th>Evaluation Methods</th>
<th>BOTTOM-UP</th>
<th>TOP-DOWN</th>
</tr>
</thead>
<tbody>
<tr>
<td>Naïve (Jacobi, Gauss-Seidel)</td>
<td>Semi-naïve (Delta Iteration)</td>
<td>Naïve Top-Down with Search trees</td>
</tr>
<tr>
<td></td>
<td>Henschen-Naqvi</td>
<td>Query-Subquery</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Rewriting Methods</th>
<th>Logic</th>
<th>Algebraic</th>
</tr>
</thead>
<tbody>
<tr>
<td>Magic Sets</td>
<td>Variable reduction</td>
<td>Static Filtering</td>
</tr>
<tr>
<td>Counting</td>
<td>Constant reduction</td>
<td>Variable reduction</td>
</tr>
<tr>
<td>Static Filtering</td>
<td></td>
<td>Constant reduction</td>
</tr>
</tbody>
</table>
8.1 Query Optimization

- Optimization techniques may be combined
  - Thus, mixed execution of rewriting and evaluation techniques based on logical and algebraic optimization is possible
- Start with logic program \(LP\)
8.1 Query Optimization

Transformation into Relational Algebra

Datalog program $\mathcal{P}$

Datalog program $\mathcal{P}'$

Relational algebra equations

Relational algebra equations

Logical query evaluation methods

Algebraic query evaluation methods

Query result
8.2. Evaluation Methods

• Evaluation methods actually compute the result of an (optimized or un-optimized) program $\mathcal{P}$

<table>
<thead>
<tr>
<th>BOTTOM-UP</th>
<th>TOP-DOWN</th>
</tr>
</thead>
</table>
| Evaluation Method | Naïve (Jacobi, Gauss-Seidel)  
Semi-naïve (Delta Iteration)  
Henschen-Naqvi |
| Naïve Top-Down with  
Search trees  
Query-Subquery |

– Better evaluation methods skip unnecessary evaluation steps and/or terminate earlier
8.2 Bottom-Up Evaluation

• Datalog programs can easily be evaluated in a bottom-up fashion, but this should also be efficient
  – The naïve algorithm derives everything that is possible from the facts
  – But naively answering queries wastes valuable work...
  – For dealing with recursion we have to evaluate fixpoints
    • For stratified Datalog\textsuperscript{f,neg} programs we apply the fixpoint algorithms to every stratum
8.2 Bottom-Up Evaluation

- **Bottom-up evaluation** techniques are usually based on the **fixpoint iteration**

- Remember: Fixpoint iteration itself is a **general concept** within all fields of mathematics
  
  - Start with an **empty initial solution** $X_0$
  
  - Compute a new $X_{n+1}$ from a given $X_n$ by using a **production rule**
    
    - $X_{n+1} := T(X_{n+1})$
  
  - As soon as $X_{n+1} = X_n$, the algorithm stops
    
    - **Fixpoint reached**
• Up to now we have stated the elementary production rule declaratively
  
  \[ T_P : I \mapsto \{ B \in B_L \mid \text{there exists a ground instance } B : - A_1, A_2, \ldots, A_n \text{ of a program clause such that } \{A_1, A_2, \ldots, A_n\} \subseteq I \} \]

• However, we need an operative implementation
  
  – The set \( I_{i+1} \) is computed from \( I_i \) as follows:
    
    • Enumerate all ground instances \( GI \)
      
      – Each ground instance is given by some substitution (out of a finite set)
    
    • Iterate over the ground instances, i.e. try all different substitutions
      
      – For each \( B : - A_1, A_2, \ldots, A_n \in GI, \text{ if } \{A_1, A_2, \ldots, A_n\} \subseteq I_i, \text{ add } B \text{ to } I_{i+1} \)
a) **Full Enumeration:** Consecutively generate and test all instances by enumeration

- Loop over all rules
  - Apply each possible substitution on each rule

**Constant symbols:** \{1,2,3\}

**Rules:** \{p(X,Y) :- e(X,Y). p(X,Y) :- e(X,Z), p(Z,Y).\}

**Enumeration of instances:**

**Rule 1:**
- p(1,1) :- e(1,1).
- p(1,2) :- e(1,2).
- p(1,3) :- e(1,3).
- p(2,1) :- e(2,1).
- p(2,2) :- e(2,2).
- p(2,2) :- e(2,2).
- p(3,1) :- e(3,1).
- p(3,2) :- e(3,2).
- p(3,2) :- e(3,2).

**Rule 2:**
- p(1,1) :- e(1,1), p(1,1).
- p(1,1) :- e(1,2), p(2,1).
- ...
8.2 Bottom-Up Evaluation

b) Restricted enumeration

- Loop over all rules
  - For each rule, generate all instances possible when trying to unify the rules right hand side with the facts in $I$
  - Only instances which will trigger a rule in the current iteration will be generated

**Constant symbols:** $\{1,2,3\}$
**Rules:**

\[
\begin{align*}
p(X,Y) & : - e(X,Y). \\
p(X,Y) & : - e(X,Z), p(Z,Y). \\
p(Z,Y) & : - \text{can not be unified with any fact in I}
\end{align*}
\]

$I$: $\{e(1,2), e(2,3)\}$

**Enumeration of instances:**

Rule 1:

\[
p(1,2) : - e(1,2). \\
p(2,3) : - e(2,3).
\]

Rule 2: Nothing. $p(Z,Y)$ can not be unified with any fact in $I$
8.2 Jacobi Iteration

• The most naïve fixpoint algorithm class are the so-called **Jacobi-Iterations**
  
  – Developed by Carl Gustav Jacob Jacobi for solving **linear equitation systems** $Ax=b$, early 18$^{th}$ century

  – Characteristics:
    
    • Each intermediate result $X_{n+1}$ is **wholly computed** by utilizing **all data** in $X_n$
    
    • **No reuse** between both results
    
    • Thus, the memory complexity for a given iteration step is roughly $|X_{n+1}|*|X_n|$
Both fixpoint iterations introduced *previously* in the lecture are *Jacobi iterations*

- i.e. fixpoint iteration and iterated fixpoint iteration

- i.e. $I_{n+1} := T_p(I_n)$

  - "Apply production rule to all elements in $I_n$ and write results to $I_{n+1}$. Repeat"
• **Please note**
  
  – **Within each iteration, all already deduced facts of previous iteration are deduced again**
    
    • Yes, they were… We just used the union notation for convenience
      
      – \( I_1 := I_0 \cup \{e(1,2), e(1,3)\} \)
      
      – \( I_2 := I_1 \cup \{p(1,2), p(1,3)\} \) was actually not reflecting this correctly
    
    – \( I_1 := \{e(1,2), e(1,3)\} \)
    
    – \( I_2 := \{e(1,2), e(1,3), p(1,2), p(1,3)\} \) matches algorithm better…

  – **Furthermore, both sets** \( I_{n+1} \) **and** \( I_n \) **involved in the iteration are treated strictly separately**
    
    • Elementary production checks which rules are true within \( I_i \) and puts result into \( I_{i+1} \)
8.2 Gauss-Seidel Iteration

• Idea:
  – The convergence speed of the Jacobi iteration can be improved by also respecting **intermediate results of current iteration**

• This leads to the class of **Gauss-Seidel-Iterations**
  – Historically, an improvement of the Jacoby equitation solver algorithm
    • Devised by **Carl Friedrich Gauss** and **Philipp Ludwig von Seidel**
  – Base property:
    • If new information is produced by current iteration, it should also possible to use it the moment it is created (and not starting next iteration)
• A Gauss-Seidel fixpoint iteration is obtained by modifying the elementary production

\[ T_p : I \mapsto \{ B \in B_L \mid \text{there exists a ground instance which has not been tested before in this iteration} \}
\]

\[ B :: A_1, A_2, \ldots, A_n \text{ of a program clause such that} \]

\[ \{A_1, A_2, \ldots, A_n\} \subseteq \{I \cup \text{new}_B\text{'s}\} \]

– new_B’s refers to all heads of the ground instances of rules considered in the current iteration which had their body literals in I

• Some of these are already in I, but others are new and would usually only be available starting next iteration → improved convergence speed
8.2 Gauss-Seidel Iteration

- Example program $\mathcal{P}$

  \[
  \begin{align*}
  \text{edge}(1, 2). \\
  \text{edge}(1, 3). \\
  \text{edge}(2, 4). \\
  \text{edge}(3, 4). \\
  \text{edge}(4, 5). \\
  \text{path}(X, Y) :&= \text{edge}(X, Y). \\
  \text{path}(X, Y) :&= \text{edge}(X, Z), \text{path}(Z, Y).
  \end{align*}
  \]

  \[
  \begin{align*}
  I_0 &= \{\} \\
  I_1 &= \{\text{edge}(1, 2). \text{edge}(1, 3). \text{edge}(2, 4). \text{edge}(3, 4). \text{edge}(4, 5). \\
  &\quad \text{path}(1, 2). \text{path}(1, 3). \text{path}(2, 4). \text{path}(3, 4). \text{path}(4, 5). \\
  &\quad \text{path}(1, 4). \text{path}(2, 5). \text{path}(3, 5) \} \\
  I_2 &= \{\text{path}(1, 5)\}
  \end{align*}
  \]
8.2 Gauss-Seidel Iteration

• Please note:
  – The **effectiveness** of **Gauss-Seidel** iteration for increasing convergence speed varies highly with respect to the chosen **order of instance enumeration**
    • e.g. “Instance K tested - generates the new fact $B_1$ from I”, “Instance L tested – generates the new fact $B_2$ from $I \cup B_1$”
      – Good luck – improvement vs. Jacobi
    • v.s. “Instance L tested – does not fire because it needs fact $B_1$”, “Instance K tested – generates the new fact $B_1$ from I”
      – Bad luck – no improvement
  – Each single iteration which can be saved improves performance dramatically as each iteration recomputes all known facts!
8.2 Semi-Naïve Evaluation

• For both Gauss-Seidel and Jacobi, a lot of wasted work is performed
  – Everything is recomputed times and again

• But it can be shown that the elementary production rule is strictly monotonic
  – Thus, each result is a subset of the next result
    • i.e. \( I_i \subseteq I_{i+1} \)

• This leads to the semi-naïve evaluation for linear Datalog
• The main operator for the fixpoint iteration is the elementary production $T_P$
  – Naïve Fixpoint Iteration
    • $I_{n+1} := T_P(I_n)$
  – Is there a better algorithm?
    • Idea: avoid re-computing known facts, but make sure that at least one of the facts in the body of a rule is new, if a new fact is computed!
    • Really new facts, always involve new facts of the last iteration step, otherwise they could already have been computed before…
8.2 Semi-Naïve Evaluation

- Semi-naïve linear evaluation algorithms for Datalog are generally known as **Delta-Iteration**
  - In each iteration step, compute just the **difference** between successive results \( \Delta I_i := I_i \setminus I_{i-1} \)
  - i.e. \( \Delta I_1 := I_1 \setminus I_0 = T_P(\emptyset) \)
    \( \Delta I_{i+1} := I_{i+1} \setminus I_i = T_P(I_i) \setminus I_i \)
    \( = T_P(I_{i-1} \cup \Delta I_i) \setminus I_i \)
8.2 Semi-Naïve Evaluation

- It is important to **efficiently** calculate \( \Delta I_{i+1} := T_P (I_{i-1} \cup \Delta I_i) \setminus I_i \)
  - Especially the \( T_P \) operator is often inefficient, because it simply applies **all rules** in the EDB
  - More efficient is the use of **auxiliary functions**
    - Define an auxiliary function of \( T_P \) \( \text{aux}_P : 2^{B_P} \times 2^{B_P} \to 2^{B_P} \)
      such that \( T_P (I_{i-1} \cup \Delta I_i) \setminus I_i = \text{aux}_P (I_{i-1}, \Delta I_i) \setminus I_i \)
    - Auxiliary functions can be chosen intelligently by just taking **recursive parts** of rules into account
    - A classic method of deriving auxiliary functions is **symbolic differentiation**
8.2 Semi-Naïve Evaluation

- The **symbolic differentiation operator** $dF$ can be used on the respective relational algebra expressions $E$ for Datalog programs

- $dF(E) := \Delta R$, if $E$ is an IDB or EDB relation $R$

- $dF(\sigma_\vartheta(E)) = \sigma_\vartheta(dF(E))$ and

- $dF(\pi_\vartheta(E)) = \pi_\vartheta(dF(E))$

- $dF(E_1 \cup E_2) = dF(E_1) \cup dF(E_2)$

This is interesting, especially since delta sets of extensional predicates are **empty**

Not affected by selections, projections, and unions
8.2 Semi-Naïve Evaluation

- \( dF(E_1 \times E_2) = E_1 \times dF(E_2) \)
  \( \cup dF(E_1) \times E_2 \)
  \( \cup dF(E_1) \times dF(E_2) \)

- \( dF(E_1 \bowtie E_2) = E_1 \bowtie dF(E_2) \)
  \( \cup dF(E_1) \bowtie E_2 \)
  \( \cup dF(E_1) \bowtie dF(E_2) \)

For Cartesian products and joins mixed terms need to be considered.
Consider the program

- `ancestor(X,Y) :- parent(X,Y).`
- `ancestor(X,Y) :- parent(X,Z), ancestor(Z,Y).`

The respective expression in relational algebra for `ancestor` is

\[
\text{parent} \bigcup \pi_{#1, #2}(\text{parent} \bowtie_{#2=#1} \text{ancestor})
\]

– Symbolic differentiation

\[
\begin{align*}
\text{dF}(\text{parent} \bigcup \pi_{#1, #2}(\text{parent} \bowtie_{#2=#1} \text{ancestor})) \\
= \text{dF}(\text{parent}) \bigcup \pi_{#1, #2}(\text{dF}(\text{parent} \bowtie_{#2=#1} \text{ancestor})) \\
= \emptyset \bigcup \pi_{#1, #2}(\text{dF}(\text{parent}) \bowtie_{#2=#1} \text{ancestor} \bowtie_{#2=#1} \text{parent} \bowtie_{#2=#1} \text{dF}(\text{ancestor})) \\
= \pi_{#1, #2}(\emptyset \bigcup \text{dF}(\text{ancestor}) \bowtie_{#2=#1} \text{parent} \bowtie_{#2=#1} \text{dF}(\text{ancestor})) \\
= \pi_{#1, #2}(\text{parent} \bowtie_{#2=#1} \Delta \text{ancestor})
\end{align*}
\]
8.2 Semi-Naïve Evaluation

- Having found a suitable auxiliary function the delta iteration works as follows

  - **Initialization**
    - $I_0 := \emptyset$
    - $\Delta I_1 := TP(\emptyset)$

  - **Iteration until $\Delta I_{i+1} = \emptyset$**
    - $I_i := I_{i-1} \cup \Delta I_i$
    - $\Delta I_{i+1} := aux_P(I_{i-1}, \Delta I_i) \setminus I_i$

- Again, for stratified Datalog$^{f, neg}$ programs the iteration has to be applied to every stratum
8.2 Semi-Naïve Evaluation

Let’s consider our ancestor program again

- `parent(Thomas, John).`  
  `parent(Mary, John).`  
  `parent(George, Thomas).`  
  `parent(Sonja, Thomas).`  
  `parent(Peter, Mary).`  
  `parent(Karen, Mary).`

- `ancestor(X,Y) :- parent(X,Y).`  
  `ancestor(X,Y) :- parent(X,Z), ancestor(Z,Y).`

- `Aux_ancestor(ancestor, Δancestor) := π_{#1, #2}(parent \bowtie_{#2=#1} Δancestor)}`
8.2 Semi-Naïve Evaluation

- \( \text{ancestor}_0 := \emptyset \)

- \( \Delta \text{ancestor}_1 := T_\mathcal{P}(\emptyset) \)
  \[= \{(T, J), (M, J), (G, T), (S, T), (P, M), (K, M)\} \]

- \( \text{ancestor}_1 := \text{ancestor}_0 \cup \Delta \text{ancestor}_1 \)
  \[= \Delta \text{ancestor}_1 \]

- \( \Delta \text{ancestor}_2 := \text{aux}_\text{ancestor}(\text{ancestor}_0, \Delta \text{ancestor}_1) \setminus \text{ancestor}_1 \)
  \[:= \pi_{#1, #2}(\text{parent} \bowtie_{#2=#1} \Delta \text{ancestor}_1) \setminus \text{ancestor}_1 \]
  \[= \{(G, J), (S, J), (P, J), (K, J)\} \]
8.2 Semi-Naïve Evaluation

\[ \text{ancestor}_2 := \text{ancestor}_1 \cup \Delta \text{ancestor}_2 \]
\[ = \{(T, J), (M, J), (G, T), (S, T), (P, M), (K, M), (G, J), (S, J), (P, J), (K, J)\} \]

\[ \Delta \text{ancestor}_3 := \text{aux}_{\text{ancestor}}(\text{ancestor}_1, \Delta \text{ancestor}_2) \setminus \text{ancestor}_2 \]
\[ := \pi_{\#_1, \#_2}(\text{parent} \bowtie_{\#_2=\#_1} \Delta \text{ancestor}_2) \setminus \text{ancestor}_2 \]
\[ = \emptyset \]

– Thus, the least fixpoint is \( \text{ancestor}_2 \cup \text{parent} \)
8.2 Push Selection

• Transforming a Datalog program into relational algebra also offers other optimizations
  – Typical relational algebra equivalences can be used for heuristically constructing better query plans
    • Usually an operator tree is built and transformed
  – Example: push selection
    • If a query involves a join or Cartesian product, pushing all selections down to the input relations avoids large intermediate results
  – But now we have a new operator in our query plan: the least fixpoint iteration (denoted as LFP)
8.2 Push Selection

• Consider an example
  – edge(1, 2).
  edge(4, 2).
  edge(2, 3).
  edge(3, 5).
  edge(5, 6).
  – path(X,Y) :- edge(X,Y).
    path(X,Y) :- edge(X,Z), path(Z,Y).
 – Relational algebra: edge $\cup \pi_{#1, #2}(\text{edge} \bowtie_{#2=#1} \text{path})$
• Now consider the query \texttt{?path(X, 3)}
  
  \begin{align*}
  &\pi_{#1} \sigma_{#2=3}(\text{LFP (edge } \bigcup \pi_{#1,#2}(\text{edge } \bowtie_{#2=#1} \text{path}))) \\
  \end{align*}

  • From which nodes there is a path to node 3?

  • The above query binds the second argument of \texttt{path}

  • \texttt{path(X,Y)} ::= \texttt{edge(X,Y)}.

  \texttt{path(X,Y)} ::= \texttt{edge(X,Z), path(Z,Y)}.

  • Thus the selection could be pushed down to the \texttt{edge} and \texttt{path} relations.
8.2 Push Selection

• To answer the query we now only have to consider the facts and rules having the correct second argument:

- `edge(2, 3)`. (fact)
- `path(2, 3)`. (R1)
- `path(1, 3)`. (R2)
- `path(4, 3)`.

– Result: \{2, 1, 4\}
Now let's try a different query \(?path(3,Y)\):

- \(\pi_{#1} \sigma_{#1=3}(\text{LFP (edge } \bigcup \pi_{#1,#2}(\text{edge } \bowtie_{#2=#1} \text{path})))\)

  - To which nodes there is a path from node 3?

  - The above query binds the first argument of \(\text{path}\)

    - \(\text{path}(X,Y) :\text{- edge}(X,Y)\).
    - \(\text{path}(X,Y) :\text{- edge}(X,Z), \text{path}(Z,Y)\).
8.2 Push Selection

- To answer the query we now only have to consider the facts and rules having the correct second argument

  - edge(3,5).
  - path(3,5).
  - Ø

- Result: {5}
- Obviously this is wrong
8.2 Push Selection

• More general: when can the least fixpoint iteration and selections be exchanged?
  
  – Let \( p \) be a predicate in a linear recursive Datalog program and assume a query \( \textcolor{red}{?}\ p(..., c, ...) \), binding some variable \( X \) at the \( i \)-th position to constant \( c \)
  
  – The selection \( \sigma_{\#i=c} \) and the least fixpoint iteration \( \text{LFP} \) can be safely exchanged, if

\[ X \text{ occurs in all literals with predicate } p \text{ exactly in the } i\text{-th position} \]
8.3. Logical Rewriting

• In the following, we deal with **rewriting** methods

<table>
<thead>
<tr>
<th>Logic</th>
<th>Algebraic</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Rewriting Method</strong></td>
<td><strong>Magic Sets</strong></td>
</tr>
<tr>
<td></td>
<td>Counting</td>
</tr>
<tr>
<td></td>
<td>Static Filtering</td>
</tr>
</tbody>
</table>

• **Basic Idea:**
  
  – Transform program $P$ to a semantically equivalent program $P'$ which can be evaluated faster using the same evaluation technique
  
  • e.g. same result, but faster when applying Jacobi iteration
8.3. Logical Rewriting

- **Clever** rewriting could work like this:

\[ \mathcal{P}: \]

\[
\begin{align*}
\text{ancestor}(X, Y) & :\ - \text{parent}(X, Y). \\
\text{ancestor}(X, Y) & :\ - \text{ancestor}(X, Z), \text{parent}(Z, Y). \\
\text{ancestor}(\text{Wolfi}, Y) & ?: 
\end{align*}
\]

- All valid proof trees for result tuples need a substitution for rule 1 and rule 2 such that \( X \) is substituted by \( \text{Wolfi} \).
8.3. Logical Rewriting

• Thus, an equivalent program \( P' \) for the query looks like this

\[
P'::
\begin{align*}
\text{ancestor}(\text{Wolfi}, Y) & : \text{parent}(\text{Wolfi}, Y). \\
\text{ancestor}(\text{Wolfi}, Y) & : \text{ancestor}(\text{Wolfi}, Z), \text{parent}(Z, Y). \\
\text{ancestor}(\text{Wolfi}, Y) & ?
\end{align*}
\]

– This simple transformation will skip the deduction of many (or in this case all) useless facts

– Actually, this transformation was straightforward and simple, but there are also unintuitive but effective translations…
  • Magic sets!
8.3. Magic Sets

• Magic Sets
  – Magic sets are a **rewriting** method exploiting the **syntactic** form of the **query**
  – The base idea is to capture some of the **binding patterns** of top-down evaluation approaches into rewriting
    • If there is a subgoal with a **bound argument**, solving this subgoal may lead to new instantiations of other arguments in the original rule
    • Only **potentially useful** deductions should be performed
• Who are the ancestors of Wolfi?
8.3. Magic Sets

• A typical **top-down search tree** for the goal \( \text{ancestor}(\text{Wolfi}, X) \) looks like this
  – Possible substitutions already restricted

\[
\begin{align*}
Q & \equiv \text{ancestor}(\text{Wolfi}, X) \\
\text{anc.}(\text{Wolfi}, X) & :- \text{anc.}(\text{Wolfi}, Z), \text{par.}(Z, X). \\
\text{anc.}(\text{Wolfi}, Z) & \\
\text{par.}(Z, X) & \\
\text{anc.}(\text{Wolfi}, X) & :- \text{par.}(\text{Wolfi}, Z). \\
\text{par.}(\text{Wolfi}, Z) &
\end{align*}
\]

– How can such a restriction be incorporated into rewriting methods?
8.3. Magic Sets

• For rewriting, propagating binding is more difficult than using top-down approaches

• **Magic Set** strategy is based on augmenting rules with additional **constraints** (collected in the magic predicate)
  – This is facilitated by "*adorning*" predicates
  – **Sideways information passing** (SIP) is used to propagate binding information
8.3. Magic Sets

• Before being able to perform the magic set transformation, we need some auxiliary definitions and considerations
  – Every query (goal) can also be seen as a rule and thus be added to the program
    • e.g. ancestor(Wolfi, X)? $\Rightarrow$ q(X) :- ancestor(Wolfi, X)
• Arguments of predicates can be **distinguished**
  
  – **Distinguished arguments** have their **range restricted** by either **constants** within the same predicate or **variables** which are already restricted themselves

  – i.e.: The argument is **distinguished** if
    • it is a **constant**
    • OR it is **bound by an adornment**
    • OR it appears in an **EDB fact** that has a distinguished argument
• **Predicates occurrences** are distinguished if all its arguments are distinguished
  – In case of EDB facts, either all or none of the arguments are distinguished
• **Predicate occurrences** are then adorned (i.e. annotated) to express which arguments are distinguished
  – Adornments are added to the predicate, e.g. \( p^{fb}(X, Y) \) vs. \( p^{bb}(X, Y) \)
– For each argument, there are two possible adornments
  - \( b \) for **bound**, i.e. distinguished variables
  - \( f \) for **free**, i.e. non-distinguished variables

– Thus, for a **predicate** with \( n \) arguments, there are \( 2^n \) possible **adorned occurrences**
  - e.g., \( p^{bb}(X, Y) \), \( p^{fb}(X, Y) \), \( p^{bf}(X, Y) \), \( p^{ff}(X, Y) \)
  - Those adorned occurrences are treated as if they were different predicates, each being defined by its own set of rules
8.3. Magic Sets

- Example output of magic set algorithm

\( \mathcal{P} \):

\[
\text{ancestor}(Wolfi, X) \ ? \\
\text{ancestor}(X, Y) : - \text{parent}(X, Y). \\
\text{ancestor}(X, Y) : - \text{ancestor}(X, Z), \text{parent}(Z, Y).
\]

\( \mathcal{P}' \):

\[
\text{magic}(Wolfi). \\
\text{magic}(Z) : - \text{magic}(Y), \text{parent}(Z, Y). \\
qf(X) : - \text{ancestor}^{bf}(Wolfi, X). \\
\text{ancestor}^{bf}(X, Y) : - \text{magic}(X), \text{parent}(X, Y). \\
\text{ancestor}^{bf}(X, Y) : - \text{magic}(X), \text{ancestor}^{bf}(X, Z), \text{parent}(Z, Y).
\]
8.3. Magic Sets

• The idea of the magic set method is that the **magic set contains all possibly interesting constant values**
  
  – The magic set is **recursively** computed by the **magic rules**

• Each **adorned predicate occurrence** has its own **defining rules**
  
  – In those rules, the attributes are restricted according to the adornment pattern to the magic set
Now, following problems remain

- How is the **magic set** computed?
- How are the **rules for adorned predicate occurrences** actually defined?

Before solving these problems, we have to find out which adorned occurrences are needed.

Thus, the **reachable adorned system** has to be found

- i.e. incorporate the query as rule and replace all predicate by its respective adornments
8.3. Magic Sets

- **Incorporate goal query**

  \[
  \text{ancestor}(X, \text{Wolfi})? \\
  \text{ancestor}(X, Y) :- \text{parent}(X, Y). \\
  \text{ancestor}(X, Y) :- \text{ancestor}(X, Z), \text{parent}(Z, Y).
  \]

  \[
  q(X) :- \text{ancestor}(X, \text{Wolfi}) \\
  \text{ancestor}(X, Y) :- \text{parent}(X, Y). \\
  \text{ancestor}(X, Y) :- \text{ancestor}(X, Z), \text{parent}(Z, Y).
  \]

- **Adorn predicate occurrences**

  \[
  q^f(X) :- \text{ancestor}^{fb}(X, \text{Wolfi}). \\
  \text{ancestor}^{fb}(X, Y) :- \text{parent}(X, Y). \\
  \text{ancestor}^{fb}(X, Y) :- \text{ancestor}^{fb}(X, Z), \text{parent}(Z, Y).
  \]

  reachable adorned system
For defining the magic set, we create **magic rules**

- For each adorned predicate occurrence in a rule of an intensional DB predicate, a magic rule corresponding to the right hand side of that rule is created

- Predicate occurrences is replaced by **magic predicate**, bound arguments are used in rule head, free ones are dropped

- Magic predicates in the head are **annotated** with its origin (rule & predicate), those on the right hand side just with the predicate

\[
q^f(X) :- \text{ancestor}^\text{fb}(X, \text{Wolfi}). \\
\Rightarrow \text{magic}_r0\text{\_ancestor}^\text{fb}(\text{Wolfi}).
\]

\[
\text{ancestor}^\text{fb}(X, Y) :- \text{ancestor}^\text{fb}(X, Z), \text{parent}(Z, Y). \\
\Rightarrow \text{magic}_r2\text{\_ancestor}^\text{fb}(Z) :- \text{magic}_\text{ancestor}^\text{fb}(Z), \text{parent}(Z, Y).
\]
8.3. Magic Sets

• Thus, we obtain multiple magic predicates for a single adorned predicate occurrence

  – Depending on the creating rule
    • e.g. magic_r0_ancestor\textsuperscript{fb}, magic_r2_ancestor\textsuperscript{fb} both using magic_ancestor\textsuperscript{fb}

  – Now we need complementary rules connecting the magic predicates
    • Adorned magic predicate follows from special rule magic predicate with same adornment
      • magic_ancestor\textsuperscript{fb} (X):- magic_r0_ancestor\textsuperscript{fb}(X).
      • magic_ancestor\textsuperscript{fb} (X):- magic_r2_ancestor\textsuperscript{fb}(X).
Finally, we have a complete definition of magic predicates with different adornments

- In our case, we have only the fb-adornment
  - magic_r0_ancestor^{fb}(Wolfi).
  - magic_r2_ancestor^{fb}(Z) :- magic_ancestor^{fb}(Z), parent (Z, Y).
  - magic_ancestor^{fb}(X) :- magic_r0_ancestor^{fb}(X).
  - magic_ancestor^{fb}(X) :- magic_r2_ancestor^{fb}(X).

- The magic magic_ancestor^{fb} set thus contains all possibly useful constants which should considered when evaluating an ancestor subgoal with the second argument bound for the current program
  - Like, e.g. our query…
8.3. Magic Sets

• As all magic sets are defined, the original rules of the reachable adorned system have to restricted to respect the sets

  – Every rule using an adorned IDB predicate in its body is augmented with an additional literal containing the respective magic set

  – e.g.

    • \( \text{ancestor}^{\text{fb}}(X, Y) :\neg \text{ancestor}^{\text{fb}}(X, Z), \text{parent}(Z, Y). \)

      \( \Rightarrow \text{ancestor}^{\text{fb}}(X, Y) :\neg\text{magic_ancestor}^{\text{fb}}(X), \text{ancestor}^{\text{fb}}(X, Z), \text{parent}(Z, Y). \)
Finally, the following program is created

\[
\text{ancestor}(X, Y) :- \text{parent}(X, Y).
\]
\[
\text{ancestor}(X, Y) :- \text{ancestor}(X, Z), \text{parent}(Z, Y).
\]
\[
\text{ancestor}(X, \text{Wolfi})? 
\]

\[
\text{magic\textsubscript{r0\_ancestor}}^{\text{fb}}(\text{Wolfi}).
\]
\[
\text{magic\textsubscript{r2\_ancestor}}^{\text{fb}}(Z) :- \text{magic\_ancestor}^{\text{fb}}(Z), \text{parent}(Z, Y).
\]
\[
\text{magic\_ancestor}^{\text{fb}}(X) :- \text{magic\textsubscript{r0\_ancestor}}^{\text{fb}}(X).
\]
\[
\text{magic\_ancestor}^{\text{fb}}(X) :- \text{magic\textsubscript{r2\_ancestor}}^{\text{fb}}(X).
\]
\[
\text{ancestor}^{\text{fb}}(X, Y) :- \text{parent}(X, Y).
\]
\[
\text{ancestor}^{\text{fb}}(X, Y) :- \text{magic\_ancestor}^{\text{fb}}(X), \text{ancestor}^{\text{fb}}(X, Z), \text{parent}(Z, Y).
\]
\[
q^{f}(X) :- \text{ancestor}^{\text{fb}}(X, \text{Wolfi}).
\]
8.3. Magic Sets

• In this example, following further optimizations are possible
  – In this case, it is not necessary to separate the two occurrences of magic_r0_ancestor\(_{fb}\) and magic_r2_ancestor\(_{fb}\)
    • No dependencies between both
    • We can unify and rename them
  – We have only one adornment pattern (\(_{fb}\)) and can thus drop it
  – This final program can be evaluated using any evaluation technique with increased performance

```
magic(\(Wolfi\)).
magic(Z) :- magic(Z), parent(Z, Y).
ancestor(X, Y) :- parent(X, Y).
ancestor(X, Y) :- magic(X), ancestor(X, Z), parent(Z, Y).
```
• **Magic Sets in short form**
  – Query is part of the program
  – Determine *reachable adorned system*
    • i.e. observe which terms are distinguished and propagate the resulting adornments
    • Reachable adorned system contains separated *adorned predicate occurrences*
  – Determine the **magic set** for each adorned predicate occurrence
    • Use **magic rules** and **magic predicates**
  – **Restricts rules** using adorned predicates to using inly the constant in the respective magic set
• Uncertain Reasoning!