Deductive Databases and Logic Programming

(Winter 2007/2008)

Chapter 6: Bottom-Up Evaluation

- Evaluation of logic programs with DB techniques.
- Predicate dependency graph.
- Seminaive evaluation.
- A compiler for Datalog into C+SQL.

Objectives

After completing this chapter, you should be able to:

- explain how bottom-up evaluation works.
- draw the predicate dependency graph for a given program, determine recursive cliques.
- translate a given Datalog rule into relational algebra and SQL.
- explain seminaive evaluation.
- translate a given Datalog program into C+SQL.

Overview

Bottom-Up Evaluation: Goal

Given:

- Relational DB with relations for EDB-predicates
- Logic program P that defines IDB-predicates

Compute:

- Minimal model, i.e. relations for IDB-predicates.

  This corresponds to the materialization of the views defined by the logic program P. The extension of the derived predicates is computed.

- Only the extension of the special predicate `answer` is important.

  Defined by $\texttt{answer}(X_1, \ldots, X_n) \leftarrow Q$ with query Q with variables X_i.

Basic Method (1)

- Compute the minimal model as least fixpoint of T_P iteratively. The simplest (very naive) algorithm is:

(1) $\mathcal{I} := \emptyset$;
(2) $\mathcal{I}_{new} := T_P(\mathcal{I})$;
(3) **while** $\mathcal{I}_{new} \neq \mathcal{I}$ **do**
(4) $\quad \mathcal{I} := \mathcal{I}_{new}$;
(5) $\quad \mathcal{I}_{new} := T_P(\mathcal{I})$;
(6) **od**;
(7) **print** $\mathcal{I}[\![\texttt{answer}]\!]$;

- The immediate consequence operator T_P has to be implemented with database techniques.

Basic Method (2)

Problems:

- Assignment of whole database states $\mathcal{I}$?

  It would be better to update only single relations.

- The relations for the EDB-predicates are given and do not change during the evaluation.
- Norecursive rules should be applied only once.

  In general, every fact should be derived only once.

- Not all facts in the minimal model are relevant for the given query (→ Magic Sets, Chapter 7).

  After the transformation, all facts in the minimal model are relevant.

Basic Method (3)

Interpretation vs. Compilation:

- Above, a bottom-up interpreter was shown.
- Often, the same logic program is executed several times for different database states.

  I.e. at different times, when updates had occurred in the meantime.

- The time invested in analysis of the input program and query optimization should be amortized over several executions of the program.
- The goal is to translate Datalog into imperative programs (relational algebra + Control structures).

Example: Input

- Let the given program be:

```
parent(X,Y)   ← mother(X,Y).
parent(X,Y)   ← father(X,Y).
ancestor(X,Y) ← parent(X,Y).
ancestor(X,Z) ← parent(X,Y) ∧ ancestor(Y,Z).
answer(X)     ← ancestor(julia,X).
```

- The last rule was automatically added for the query `ancestor(julia,X)`.
- EDB-Predicates (stored in the database):
  - ◇ `father`
  - ◇ `mother`

Example: Output

```
parent := mother ∪ father;
ancestor := ∅;
ancestor_new := parent;
while ancestor_new ≠ ∅ do
    ancestor := ancestor ∪ ancestor_new;
    ancestor_new :=
        π_{$1,$3}(parent ⋈_{$2=$1} ancestor_new);
    ancestor_new := ancestor_new − ancestor;
od;
answer := π_{$2}( σ_{$1='julia'}(ancestor) );
print answer;
```


Example: Remark

- Here, a version of relational algebra is used that refers to columns by position, not by name.

  Some more theoretical database textbooks use this convention because it is easier to define, e.g. there can be no name clashes.

- Usually, the columns of IDB-predicates have no names, but one could of course assign artifical names.

  E.g., if one uses an SQL-database.

- $\$i$ is the i-th column of the input relation.
- For the join $R \underset{\$i=\$j}{\bowtie} S$, the i-th column of R must be equal to the j-th column of S.

Example: Exercise

- Please execute the above program for the following database state:

mother

Child	Mother
emil	birgit
frida	doris
julia	frida

father

Child	Father
emil	arno
frida	chris
julia	emil

EDB- vs. IDB-Predicates (1)

- The distinction between EDB- and IDB-predicates is not important for the semantics (minimal model), and also not for SLD-resolution, but it is fundamental for bottom-up query evaluation.
- Above (in the chapter about Pure Prolog), it was assumed that the EDB-predicates are defined in the logic program (as facts). This is usual in Prolog.
- However, in database applications there are often thousands or millions of facts, but only a few rules.

EDB- vs. IDB-Predicates (2)

- A fact can be seen as special case of a rule (a rule with empty body), but since this special case is so common, it deserves a special treatment.

  It is a general principle of efficiency improvement to treat simple and very common special cases separately.

- Furthermore,
  - ◇ facts are changed by updates, whereas
  - ◇ a query might be executed several times, and
  - ◇ view definitions are stable.

EDB- vs. IDB-Predicates (3)

Definition:

- Given a logic program P, a predicate p that occurs in P is an IDB-predicate (of P), if P contains at least one rule $A \leftarrow B_1 \wedge \cdots \wedge B_m$ with $pred(A) = p$ and $m \geq 1$, or an EDB-predicate (of P) otherwise.
- Let $\mathcal{P}_{\mathcal{IDB}}(P)$ be the set of IDB-predicates of P, and $\mathcal{P}_{\mathcal{EDB}}(P)$ be its set of EDB-predicates.
- $IDB(P) := \{A \leftarrow B_1 \wedge \cdots \wedge B_m \in P \mid pred(A) \in \mathcal{P}_{\mathcal{IDB}}(P),\ m \geq 0\}.$
  $EDB(P) := \{A \mid A \leftarrow true \in P,\ p(A) \in \mathcal{P}_{\mathcal{EDB}}(P)\}.$

EDB- vs. IDB-Predicates (4)

Interpreter with Distinction EDB-IDB:

(1) $\mathcal{I}_{db} := EDB(P)$;
(2) $P' := IDB(P)$;
(3) $\mathcal{I} := \emptyset$;
(4) $\mathcal{I}_{new} := T_{P'}(\mathcal{I} \cup \mathcal{I}_{db})$;
(5) **while** $\mathcal{I}_{new} \neq \mathcal{I}$ **do**
(6) $\quad \mathcal{I} := \mathcal{I}_{new}$;
(7) $\quad \mathcal{I}_{new} := T_{P'}(\mathcal{I} \cup \mathcal{I}_{db})$;
(8) **od**;
(9) **print** $\mathcal{I}(\texttt{answer})$;

Overview

1. Basic Approach of Bottom-Up Evaluation
2. Predicate Dependency Graph
3. Translation of Rules
4. First Compiler for Datalog
5. Seminaive Evaluation
6. Further Remarks: Optimizations, Problems

Predicate Dependencies (1)

Definition:

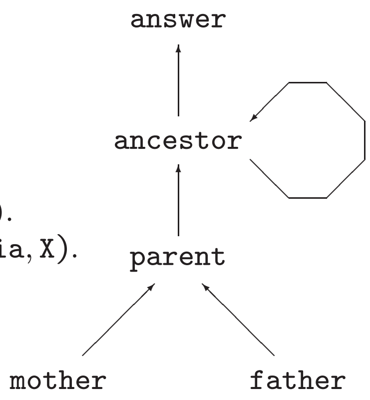
- The predicate-dependency graph of a logic program P is the directed graph $\mathcal{DG}(P) = (V, E)$ with
  - ◇ the set V of nodes is the set of all predicates that occur in P,
  - ◇ there is an edge from p to q, i.e. $(p, q) \in E \subseteq V \times V$, if and only if there is a rule $A \leftarrow B_1 \wedge \cdots \wedge B_m$ in P and $i \in \{1, \ldots, m\}$, such that $pred(B_i) = p$ and $pred(A) = q$.

    Some authors use the other direction of the arrows.

Predicate Dependencies (2)

Example:

```
parent(X,Y)   ← mother(X,Y).
parent(X,Y)   ← father(X,Y).
ancestor(X,Y) ← parent(X,Y).
ancestor(X,Z) ← parent(X,Y) ∧
                ancestor(Y,Z).
answer(X)     ← ancestor(julia,X).
```



Predicate Dependencies (3)

Definition:

- Let a logic program P be given.
- A predicate q depends on a predicate p iff there is a path (consisting of one or more edges) from p to q in $\mathcal{DG}(P)$.
- A predicate p is recursive iff p depends on itself.
- The nodes $\{p_1, \ldots, p_k\}$ of a strongly connected component (SCC) of $\mathcal{DG}(P)$ are called recursive clique.
- Two predicates p and q that belong to the same recursive clique are mutually recursive.

Predicate Dependencies (4)

Exercise:

Compute the predicate dependency graph and the (mutually) recursive predicates for the following program:

`rule` and `rhs` contain a context free grammar. The program computes the nonterminal symbols from which the empty string can be derived.

```
empty_until(Rule, 0)     ← rule(Rule, _, _).
empty_until(Rule, Next)  ← empty_until(Rule, Pos) ∧
                           succ(Pos, Next) ∧
                           rhs(Rule, Next, NonTerm) ∧
                           empty(NonTerm).
empty(Left)              ← rule(Rule, Left, RightSize) ∧
                           empty_until(Rule, RightSize).
```

Predicate Dependencies (5)

Definition:

- A rule $A \leftarrow B_1 \wedge \cdots \wedge B_m$ is a recursive rule iff there is $i \in \{1, \ldots, m\}$, such that $pred(B_i)$ depends on $pred(A)$.

  Note that not all rules about a recursive predicate are recursive rules.

- Let P be a logic program and $C \subseteq \mathcal{P}_{\mathcal{IDB}}(P)$. Then
  - ◇ $rec(P, C)$ is the set of recursive rules from P about predicates in C.
  - ◇ $nrec(P, C)$ is the set of nonrecursive rules from P about predicates in C.

Predicate Dependencies (6)

Definition:

The reduced predicate-dependency graph of a program P is the directed acyclic graph $\mathcal{RG}(P) = (\hat{V}, \hat{E})$ with

- $\hat{V} := \{C \subseteq \mathcal{P}_{\mathcal{IDB}}(P) \mid C \text{ is recursive clique}\}$
  $\cup \{p \in \mathcal{P}_{\mathcal{IDB}}(P) \mid p \text{ is non-recursive}\}$.
- $(C_1, C_2) \in \hat{E}$ if and only if $C_1 \neq C_2$ and there are $p_1, p_2 \in \mathcal{P}_{\mathcal{IDB}}(P)$ with
  - ◇ $p_1 \in C_1$ or $p_1 = C_1$,
  - ◇ $p_2 \in C_2$ or $p_2 = C_2$,
  - ◇ there is an edge $p_1 \longrightarrow p_2$ in $\mathcal{DG}(P)$.

Predicate Dependencies (7)

Exercise:

- Compute the reduced predicate dependency graph:

$$
\begin{array}{lll@{\qquad\qquad}lll}
p_1 & \leftarrow & q_1 \wedge q_2. & p_6 & \leftarrow & p_5. \\
p_1 & \leftarrow & q_1 \wedge q_3. & p_4 & \leftarrow & p_6. \\
p_2 & \leftarrow & p_1. & p_7 & \leftarrow & p_5 \wedge p_3. \\
p_3 & \leftarrow & q_3. & p_7 & \leftarrow & q_4 \wedge p_7. \\
p_3 & \leftarrow & p_1. & p_8 & \leftarrow & p_2. \\
p_3 & \leftarrow & p_2. & p_9 & \leftarrow & p_8 \wedge p_7. \\
p_4 & \leftarrow & p_2 \wedge p_3. & & & \\
p_5 & \leftarrow & p_4 \wedge q_2. & & &
\end{array}
$$

Predicate Dependencies (8)

Definition:

- Let P be a logic program and $\mathcal{RG}(P) = (\hat{V}, \hat{E})$ be its reduced predicate-dependency graph.
- A predicate evaluation sequence for P is a sequence $C_1, \ldots, C_k$ of the nodes of this graph such that

  for all $i, j \in \{1, \ldots, k\}$: $(C_i, C_j) \in \hat{E} \implies i < j$

  (i.e. there are only forward edges).
- One gets such a sequence by topologically sorting $\hat{V}$ with respect to the order relation $\hat{E}$.

Predicate Dependencies (9)

- A predicate evaluation sequence always exists.

  However, it may be not unique.

- Example: A predicate evaluation sequence for

$$
\begin{array}{ll}
\texttt{parent(X,Y)} & \leftarrow \texttt{mother(X,Y)}.\\
\texttt{parent(X,Y)} & \leftarrow \texttt{father(X,Y)}.\\
\texttt{ancestor(X,Y)} & \leftarrow \texttt{parent(X,Y)}.\\
\texttt{ancestor(X,Z)} & \leftarrow \texttt{parent(X,Y)} \land \texttt{ancestor(Y,Z)}.\\
\texttt{answer(X)} & \leftarrow \texttt{ancestor(julia,X)}.
\end{array}
$$

  is $\texttt{parent}, \{\texttt{ancestor}\}, \texttt{answer}$.

- Exercise: Compute a predicate evaluation sequence for the program on Slide 6-23.

Predicate Dependencies (10)

Interpreter with evaluation sequence

(1) $\mathcal{I}_{db} := EDB(P)$;
(2) Compute evaluation sequence $C_1, \ldots, C_k$;
(3) $\mathcal{I} := \emptyset$;
(4) **for** $i := 1$ **to** k **do**
(5)   /∗ Extend $\mathcal{I}$ by evaluating C_i: ∗/
(6)   ... see next slide ...
(16) **od**;
(17) **print** $\mathcal{I}(\texttt{answer})$;

Predicate Dependencies (11)

(6) **if** C_i is nonrecursive predicate p **then**
(7) $\mathcal{I} := \mathcal{I} \cup T_{nrec(P,\{p\})}(\mathcal{I} \cup \mathcal{I}_{db})$;
(8) **else** /∗ C_i is recursive clique ∗/
(9) $\mathcal{I}_{new} := T_{nrec(P,C_i)}(\mathcal{I} \cup \mathcal{I}_{db})$;
(10)   **while** $\mathcal{I}_{new} \neq \emptyset$ **do**
(11) $\mathcal{I} := \mathcal{I} \cup \mathcal{I}_{new}$;
(12) $\mathcal{I}_{new} := T_{rec(P,C_i)}(\mathcal{I} \cup \mathcal{I}_{db})$;
(13) $\mathcal{I}_{new} := \mathcal{I}_{new} - \mathcal{I}$;
(14)   **od**;
(15) **fi**;

Predicate Dependencies (12)

Remark (Rule Dependency Graph):

- One can also define a rule-dependency graph in which the rules are represented as nodes, and there is an edge from rule $\mathcal{R}_1$ to rule $\mathcal{R}_2$ if the body of $\mathcal{R}_2$ contains a predicate that appears in the head of $\mathcal{R}_1$.
- This graph contains so many edges that it is never used (but see predicate-rule graph below).
- An advantage might be that the recursive rules are exactly the rules occurring in cycles.

Predicate Dependencies (13)

Remark (Predicate-Rule Graph):

- An improvement is the predicate-rule graph: It is a bipartite graph with rules and predicates as nodes.
  - ◇ There is an edge from predicate p to rule $\mathcal{R}$ if p appears in the body of $\mathcal{R}$.
  - ◇ There is an edge from rule $\mathcal{R}$ to predicate p if p appears in the head of $\mathcal{R}$.
- But the predicate dependency graph is the smallest of the three, and together with the rules, the information in the other graphs can easily derived.

Overview

Translation of Rules (1)

- Statements like the following must still be further refined:
$$\mathcal{I} := \mathcal{I} \cup T_{nrec(P,C_i)}(\mathcal{I} \cup \mathcal{I}_{db});$$
- For nonrecursive components (and seminaive evaluation of recursive ones, see below), rules can be executed one after the other.
- I.e., if head and body predicates are disjoint,
$$\mathcal{I} := \mathcal{I} \cup T_{\{\mathcal{R}_1,\ldots,\mathcal{R}_n\}}(\mathcal{I})$$
is equivalent to
$$\mathcal{I} := \mathcal{I} \cup T_{\{\mathcal{R}_1\}}(\mathcal{I});\ \ldots\ \mathcal{I} := \mathcal{I} \cup T_{\{\mathcal{R}_n\}}(\mathcal{I});$$

Translation of Rules (2)

Example:

- Input rule: $p(X, a) \leftarrow q(X, Y) \land r(Y, b).$
- The principles are:
  - ◇ Do a selection for constants in the body,

    Or if a variable appears more than once in the same body literal.
  - ◇ a join for common variables of different body literals, and
  - ◇ a projection for the head.
- Translation into relational algebra:
$$p := p \cup \pi_{\$1,a}\Big(q \underset{\$2=\$1}{\bowtie} \sigma_{\$2=b}(r)\Big).$$

Translation of Rules (3)

Example, continued:

- The same principles are applied for a translation to SQL.
- Input rule (again): $p(X, a) \leftarrow q(X, Y) \wedge r(Y, b)$.
- Execution in an SQL database:

```
INSERT INTO p (SELECT DISTINCT B1.$1, 'a'
               FROM   q B1, r B2
               WHERE  B1.$2=B2.$1 AND B2.$2='b')
```

- Of course, the $$i$ could be replaced by other column names.

Translation of Rules (4)

- Input Rule (again): $p(X, a) \leftarrow q(X, Y) \wedge r(Y, b)$.
- Direct implementation with "Nested Loop Join":

(1) **foreach** $B_1 \in q$ **do**
(2) $\quad X := B_1[1]$; $Y := B_1[2]$;
(3) $\quad$ **foreach** $B_2 \in r$ **do**
(4) $\quad\quad$ **if** $B_2[1] = Y$ **and** $B_2[2] =$ 'b' **then**
(5) $\quad\quad\quad$ /∗ Insert tuple into p ∗/
(6) $\quad\quad\quad$ ... see next slide ...
(16) $\quad\quad$ **fi**;
(17) $\quad$ **od**;
(18) **od**;

Translation of Rules (5)

- Insertion with duplicate test:

(6) $Duplicate := false$;
(7) **foreach** $A \in p$ **do**
(8) $\quad$ **if** $A[1] = X$ **and** $A[2] =$ 'a' **then**
(9) $\quad\quad Duplicate := true$; **break**;
(10) $\quad$ **fi**;
(11) **od**;
(12) **if not** $Duplicate$ **then**
(13) $\quad$ **new** A'; $A'[1] = X$; $A'[2] =$ 'a';
(14) $\quad$ **append** A' **to** p;
(15) **fi**;

Translation of Rules (6)

Exercises:

- Name some alternatives for evaluating this rule.

  E.g. another join method, using indexes. Are there other possibilities for eliminating duplicates?

- If this is the only/first rule about p, is it possible to work without the duplicate test?

  Or is further knowledge necessary (key constraints)?

- For nonrecursive programs, it is possible to eliminate duplicates only once at the end. What are the advantages and disadvantages?

Translation of Rules (7)

Exercises, continued:

- Why might this method (deferred duplicate elimination) be dangerous in recursive programs?
- Name some advantages and disadvantages of using a commercial DBMS (translation into SQL) compared with the direct translation to Pascal.

  At this state, it might be difficult to find convincing disadvantages. When discussing seminaive evaluation below, it will become clear that some kind of versioned relations are needed that are missing in todays commercial databases.

Translation of Rules (8)

Remark (General Approach):

- It is not difficult to write a program for translating Datalog rules into relational algebra or SQL.

  However, a good translation into program code (Pascal, C, abstract machine) that contains a query optimizer and efficiently accesses external memory is a big project (basically, one reimplements a DBMS).
- When a variable appears for the first time, one has to remember its value (or a reference to the value), and for every future occurrence of the same variable, one must do a comparison.

  E.g., when translating to SQL, one will use a table that maps X to `B1.$1` after the first body literal is processed.

Nonrecursive Programs (1)

- If there are several rules about a predicate, one can combine the respective algebra expressions / SQL queries with $\cup$/`UNION`.
- For non-recursive programs, one can successively replace the IDB-predicates in the algebra expression for `answer` by their definitions (algebra expressions).

  Like a macro expansion. This process is called "unfolding".
- In this way, one gets a single algebra expression that contains only EDB-predicates and computes the query result.

Nonrecursive Programs (2)

- If one works with SQL, one can also translate an entire nonrecursive Datalog program into a single SQL query.

  This is obvious, since SQL is at least as powerful as relational algebra: If a translation to relational algebra is possible, the resulting algebra expression can be translated to SQL.
- One can also define views for the IDB predicates.
- Many DBMS will internally do a view expansion.

Nonrecursive Programs (3)

- When the IDB predicates are eliminated, the optimizer has more possibilities (it is not bound by the partitioning of the program into different predicate definitions).

  The optimizer than can globally work on the entire query instead of optimizing only locally every single rule.

- Thus, this single big query will often be executed more efficiently than the sequential execution of all rules.

Nonrecursive Programs (4)

- Another important advantage of the single big query is that the IDB predicates do not have to be “materialized” in this way (explicitly stored).
- Database systems often use internally iterators for subexpressions that compute the next tuple only when it is actually needed.
- Since main memory is restricted, it is important to store intermediate results only when really necessary, e.g. when sorting.

Nonrecursive Programs (5)

- An exception occurs if the same predicate is used several times: The DBMS might not detect the common subexpression and recompute the same intermediate tuples.
- The effect is not so clear and deserves are careful analysis: If the work for recomputation is cheap, that might still be the better alternative.
- Also, if different parts of the extension of the predicates are used, treating the two subexpressions separately might be much better.

Nonrecursive Programs (6)

Exercise:

- Translate the following program into a single algebra expression (and a single SQL query):

$$\begin{array}{lll}
\mathtt{parent}(\mathtt{X},\mathtt{Y}) & \leftarrow & \mathtt{mother}(\mathtt{X},\mathtt{Y}). \\
\mathtt{parent}(\mathtt{X},\mathtt{Y}) & \leftarrow & \mathtt{father}(\mathtt{X},\mathtt{Y}). \\
\mathtt{grandfather}(\mathtt{X},\mathtt{Z}) & \leftarrow & \mathtt{parent}(\mathtt{X},\mathtt{Y}) \wedge \mathtt{father}(\mathtt{Y},\mathtt{Z}). \\
\mathtt{answer}(\mathtt{X}) & \leftarrow & \mathtt{grandfather}(\mathtt{julia},\mathtt{X}).
\end{array}$$

- Do an algebraic optimization and discuss different query evaluation plans.

  If you took Database Systems IIB”.

Recursive Programs

- In this case, there are two interpretations ($\mathcal{I}$, $\mathcal{I}_{new}$):

$$\mathcal{I}_{new} := T_{rec(P,C_i)}(\mathcal{I} \cup \mathcal{I}_{db});$$

Actually three, but $\mathcal{I}$ and $\mathcal{I}_{db}$ define disjoint predicates.

- Solution: One uses only a single interpretation (database), but introduces different variants of the predicates. E.g.

$$\texttt{ancestor(X,Z)} \leftarrow \texttt{parent(X,Y)} \wedge \texttt{ancestor(Y,Z)}$$

is translated like

$$\texttt{ancestor\_new(X,Z)} \leftarrow \texttt{parent(X,Y)} \wedge \texttt{ancestor(Y,Z)}.$$

Overview

Compiler for Datalog (1)

- One can get a compiler from an interpreter by partial evaluation.
- The interpreter is executed as far as possible.

  The logic program is already known (at compile time), but the input relations (values for the EDB predicates) will only be known at runtime. If an EDB predicate is known to be small and very stable (lookup tables), one could think about using it in the compilation.

- Whenever a statement depends on the actual data, one prints the statement instead of executing it.

  Of course, it might be possible to specialize/simplify the statement.

Compiler for Datalog (2)

Notation/Auxillary Procedures:

- $\mathcal{P}^{new}_{\mathcal{IDB}}(P) := \{p\_new \mid p \in \mathcal{P}_{\mathcal{IDB}}(P),\ p$ is recursive$\}$.
- $translate(C)$ translates rule sets as explained in the last section.
- $make\_head\_new(C)$ changes every predicate p in a rule head to p_new.

Compiler for Datalog (3)

foreach $p \in \mathcal{P}_{\mathcal{IDB}}(P) \cup \mathcal{P}_{\mathcal{IDB}}^{new}(P)$ **do**
    print `"CREATE TABLE` p `(...)"`; **od**;

Compute evaluation sequence $C_1, \ldots, C_k$;
for $i := 1$ **to** k **do** /* Evaluate C_i: */
    . . . see next slide . . .
od;

print `"SELECT * FROM answer"`;
foreach $p \in \mathcal{P}_{\mathcal{IDB}}(P) \cup \mathcal{P}_{\mathcal{IDB}}^{new}(P)$ **do**
    print `"DROP TABLE` p`"`; **od**;

Compiler for Datalog (4)

if C_i is nonrecursive predicate p **then**
    $translate\big(\,nrec(P, C_i)\,\big)$;
else /* C_i is recursive clique */
    $translate\big(\,make\_head\_new(nrec(P, C_i))\,\big)$;
    print `"WHILE "`;
    $print\_while\_cond(C_i)$;
    print `" DO "`;
    $print\_while\_body(C_i, P)$;
    print `" OD;"`;
fi;

Compiler for Datalog (5)

procedure $print\_while\_cond(C_i)$:
    $First := true$;
    foreach $p \in C_i$ **do**
        if not $First$ **then**
            print `" OR "`;
        else
            $First := false$;
        fi;
        print "$p\_new \neq \emptyset$";
    od;

Compiler for Datalog (6)

procedure $print\_while\_body(C_i, P)$:
    foreach $p \in C_i$ **do**
        print "$p := p \cup p\_new$;";
    od;
    $translate\big(\,make\_head\_new(rec(P, C_i))\,\big)$;
    foreach $p \in C_i$ **do**
        print "$p\_new := p\_new - p$;";
    od;

Overview

Seminaive Evaluation (1)

- The first algorithm, which simply iterates to T_P-operator until a fixpoint is reached, is called "naive bottom-up evaluation".
- The main disadvantage is that in every iteration, it recomputes the facts already known from previous iterations.
- The goal now is that every applicable rule instance is applied only once.

  Actually, one would like to derive only fact only once. But if a fact can be derived with different rule instances, this is at least difficult.

Seminaive Evaluation (2)

- With the improved algorithm above that uses the predicate dependency graph, the goal (single application of every applicable rule instance) was reached for nonrecursive programs.
- But for recursive programs, nothing has changed.

Seminaive Evaluation (3)

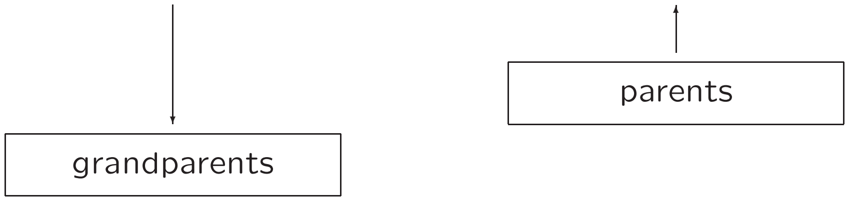
Translation Result so far:

(1) `ancestor_new` := `parent`;
(2) **while** `ancestor_new` $\neq \emptyset$ **do**
(3) `ancestor` := `ancestor` $\cup$ `ancestor_new`;
(4) `ancestor_new` :=
(5) $\pi_{\$1,\$3}(\texttt{parent} \underset{\$2=\$1}{\bowtie} \texttt{ancestor})$;
(6) `ancestor_new` := `ancestor_new` $-$ `ancestor`;
(7) **od**;

Seminaive Evaluation (4)

Naive Evaluation, Iteration 1:

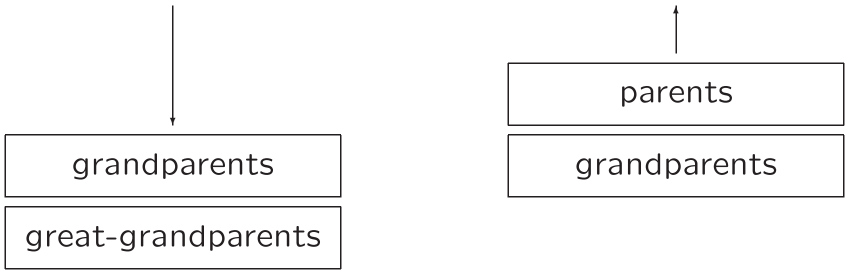
$$\texttt{ancestor\_new(X, Z)} \leftarrow \texttt{parent(X, Y)} \wedge \texttt{ancestor(Y, Z)}$$



Seminaive Evaluation (5)

Naive Evaluation, Iteration 2:

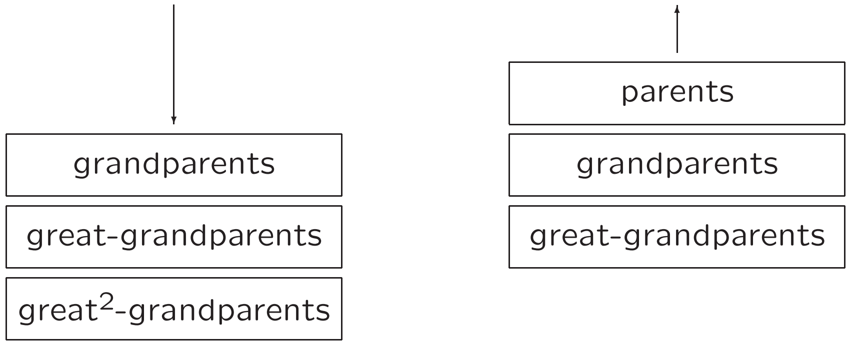
$$\texttt{ancestor\_new(X, Z)} \leftarrow \texttt{parent(X, Y)} \wedge \texttt{ancestor(Y, Z)}$$



Seminaive Evaluation (6)

Naive Evaluation, Iteration 3:

$$\texttt{ancestor\_new(X, Z)} \leftarrow \texttt{parent(X, Y)} \wedge \texttt{ancestor(Y, Z)}$$



Seminaive Evaluation (7)

Basic Idea:

- Solution: Seminaive/Differential Evaluation.
- If there is only one recursive body literal, it suffices to match it only with new facts, i.e. facts that were derived in the last iteration.
- In the example, one could replace `ancestor` in the body by `ancestor_new`:

$$\texttt{ancestor\_new} := \pi_{\$1,\$3}(\texttt{parent} \underset{\$2=\$1}{\bowtie} \texttt{ancestor\_new});$$

Seminaive Evaluation (8)

- The general case is a bit more complicated:
  - ◇ If there are several recursive body literals, one also has to consider combinations of old and new body literals.
  - ◇ It might be a problem that the same relation is used in head and body.

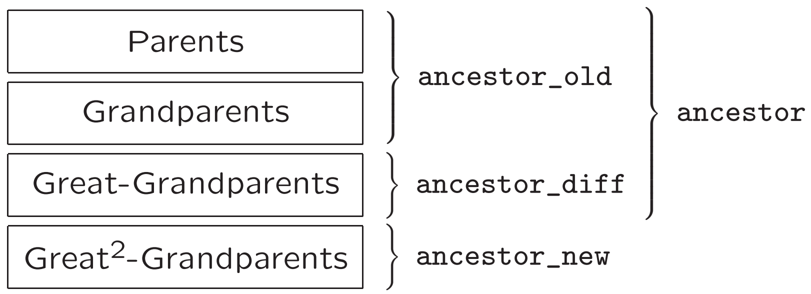
    Chaos could occur when a relation is modified while it is accessed. However, if one uses an SQL database, this would make sure that the `SELECT`-query in an `INSERT`-statement is completely evaluated before the tuples are actually inserted. Then there could be only problems when there are several rules or several mutually recursive predicates (the rule applications in one iteration should be based on the same database state).

Seminaive Evaluation (9)

- Seminaive evaluation is defined by a program transformation using new, system-defined predicates.
- For every recursive predicate p, three additional variants are introduced:
  - ◇ p_old: Tuples that existed already when the previous iteration started.
  - ◇ p_diff: Tuples that were newly derived in the previous iteration.
  - ◇ p: Value after previous iteration ($p\_old \cup p\_diff$).
  - ◇ p_new: Tuples to be computed in this iteration.

Seminaive Evaluation (10)

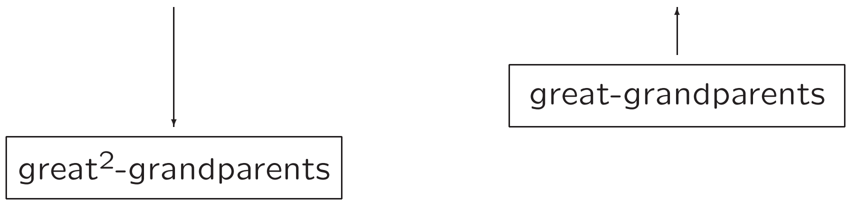
Seminaive Evaluation, Iteration 3:



Seminaive Evaluation (11)

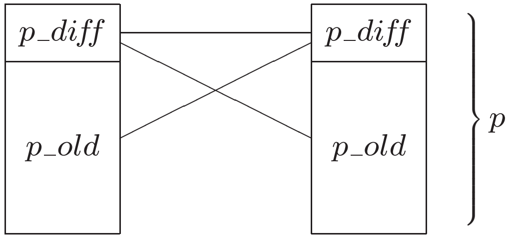
Seminaive Evaluation, Iteration 3:

$$\texttt{ancestor\_new}(\texttt{X},\texttt{Z}) \leftarrow \texttt{parent}(\texttt{X},\texttt{Y}) \land \texttt{ancestor\_diff}(\texttt{Y},\texttt{Z})$$



Seminaive Evaluation (12)

- Example: $p(\mathtt{X},\mathtt{Z}) \leftarrow p(\mathtt{X},\mathtt{Y}) \wedge p(\mathtt{Y},\mathtt{Z})$.



- Result of the transformation:

$$\begin{array}{lcll} p\_new(\mathtt{X},\mathtt{Z}) & \leftarrow & p\_diff(\mathtt{X},\mathtt{Y}) & \wedge\ p(\mathtt{Y},\mathtt{Z}). \\ p\_new(\mathtt{X},\mathtt{Z}) & \leftarrow & p\_old(\mathtt{X},\mathtt{Y}) & \wedge\ p\_diff(\mathtt{Y},\mathtt{Z}). \end{array}$$

Seminaive Evaluation (13)

- For n recursive body literals, $2^n - 1$ combinations of "old" and "diff" must be considered.

  In practice, this would not be as bad as it sounds theoretically, because rules with more than two recursive body literals are very seldom.

- However, since there is also a relation for the union ($p = p\_old \cup p\_diff$), one can write this down with n rules.
- Of the 2^n possible combinations of "old" and "diff", seminaive evaluation only avoids one combination, namely, where all tuples are old.

Seminaive Evaluation (14)

- However, since the "old" tuples are possibly collected over many iterations, there are often many more "old" tuples than "diff" tuples (only 1 iter.).
- In this case, avoiding the combination of old with old tuples really saves work.
- Furthermore, it might be important that every rule instance is applied only once:
  - ◇ For avoiding duplicate tests.
  - ◇ For using the right number of duplicates in aggregation functions like `sum` and `avg`.

Seminaive Evaluation (15)

- Initialization (before the **while**-loop):

  (1) $p\_old := \emptyset$;
  (2) $p\_diff :=$ Facts from nonrecursive rules;
  (3) $p := p\_diff$;
  (4) $p\_new := \emptyset$;

- After every iteration:

  (1) $p\_old := p\_old \cup p\_diff$; /* Or: $p\_old := p$; */
  (2) $p\_diff := p\_new - p\_old$;
  (3) /* Or: $p\_diff := p\_new - p$; */
  (4) $p := p\_old \cup p\_diff$;
  (5) $p\_new := \emptyset$;

Seminaive Evaluation (16)

Managing different variants of a relation:

- Of course, it is quite inefficient to really use four relations per recursive predicate.

  Especially the copying of tuples between different relations seems superfluous work.

- But with a standard SQL database, there seems to be no perfect solution.

  As mentioned above, if there are not many recursive body literals, one could avoid one of the four relations. Furthermore, if there is only a single rule with a single recursive body literal, two relations would suffice as shown in the `ancestor`-example.
  Another approach is to use a single relation with an additional column for the iteration step in which the tuple was derived.

Seminaive Evaluation (17)

- If one builds a deductive DBMS from scratch, one can use a new data structure, e.g. consisting of:
  - ◇ A B-tree/hash table to detect duplicates. Derived tuples are immediately checked, only $p\_new{-}p$ is stored.
  - ◇ A file/linked list, in which new tuples are only appended at the end. Then p_old, p_diff and p_new can be implemented by position pointers.

    End of the list before the previous iteration step, after the previous iteration step, and the current end of the list. Then after each iteration only these pointers must be updated, no tuples copied.

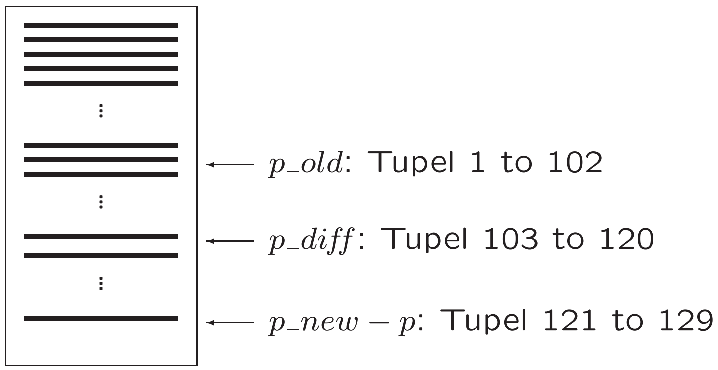
Seminaive Evaluation (18)

- In the above data structure, every tuple is stored twice: In the B-tree in sort order, in the list in derivation sequence.
- In a standard SQL DBMS, a heap file with a B-tree index over all attributes would look similar.
- However, standard DBMS do not guarantee to store tuples in the heap file in insertion sequence.

  Actually, effort is invested to ensure that if a short tuple arrives after a large tuple, the short tuple might be used to fill a block that did not have enough space for the large tuple. Furthermore, it is not clear whether the order on the ROWIDs is meaningful.

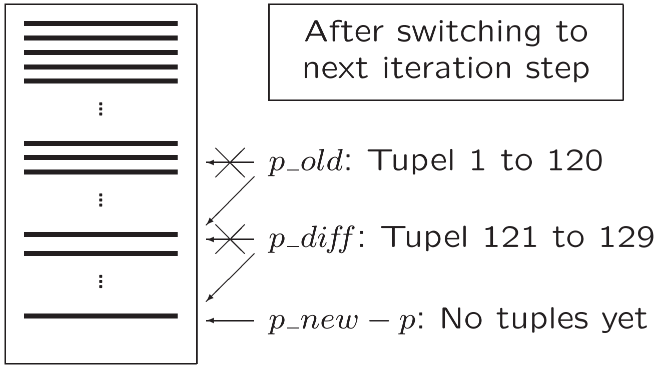
Seminaive Evaluation (19)

Own data structure for recursive predicates:



Seminaive Evaluation (20)

Own data structure for recursive predicates:



Overview

1. Basic Approach of Bottom-Up Evaluation
2. Predicate Dependency Graph
3. Translation of Rules
4. First Compiler for Datalog
5. Seminaive Evaluation
6. Further Remarks: Optimizations, Problems

Using a RDBMS (1)

- As mentioned before, a relatively easy implementation option for a deductive DBMS is to use a relational DBMS as a basis, and translate the given logic program into SQL plus control structures.
- This will not give optimal performance:
  - ◇ Versioned relations as needed for seminaive evaluation are not supported.

    Alteratively, the following kind of statement would be useful: Insert a tuple into R and S, if it is not already in R.
  - ◇ (continued on next slide)

Using a RDBMS (2)

- Reasons for suboptimal performance when using a standard relational DBMS, continued:
  - ◇ Only `INSERT` is needed for IDB-relations, no `UPDATE` or `DELETE`. The storage structures should be optimized for this case.
  - ◇ No size information is known about the IDB-relations when the logic program is translated, and during seminaive iteration, the size changes drastically.

    Runtime optimization might help here.

Using a RDBMS (3)

- Other important issued when implementing a deductive DBMS based on a standard RDBMS:
  - ◇ Modern RDBMS support temporary relations (no recovery, no multi-user access). These should be used for IDB-relations.
  - ◇ It is important not to materialize IDB predicates, at least when the predicate is used only once.
  - ◇ One should use stored procedures in order to avoid network traffic between server and client during the execution of the logic program.

Further Optimizations (1)

- Tests have shown that a large fraction of the runtime is used for duplicate elimination.
  - ◇ Therefore, the optimizer should do an analysis which rules can never produce duplicates.
- Recursive rules produce larger and larger intermediate results, which are usually kept until the end of the iteration.
  - ◇ It would be better to use the tuples immediately and then delete them.

    This can only work if the recursive rule will not produce duplicates. Otherwise the old tuples are needed to ensure termination.

Further Optimizations (2)

- An optimal situation is when only a single tuple of the recursive predicate is needed at each time point.

/* Computes position of first space in string */

$$
\begin{array}{lll}
first\_space(Pos) & \leftarrow & no\_space\_before(Pos) \wedge \\
 & & string(Pos,\ '\ '). \\
no\_space\_before(1). & & \\
no\_space\_before(Next) & \leftarrow & no\_space\_before(Pos) \wedge \\
 & & string(Pos,\ Char) \wedge \\
 & & Char \neq '\ ' \wedge \\
 & & succ(Pos,\ Next).
\end{array}
$$