# 3. Bottom-Up Auswertung

1. Zielsetzung, Grundsätzliche Idee
2. EDB- vs. IDB-Prädikate
3. Prädikat-Abhängigkeitsgraph, Rekursion
4. Übersetzung von Regeln in Relationen-Algebra, SQL, etc.
5. Erster Compiler für Datalog
6. Seminaive Auswertung
7. Weitere Optimierungsmöglichkeiten
8. Schwierigkeiten bei der Implementierung auf kommerziellen RDBen

# Zielsetzung

## Gegeben:

- Relationale Datenbank mit Relationen für EDB-Prädikate
- Logisches Programm P, das IDB-Prädikate definiert

## Berechne:

- Minimales Modell
- D.h. Relationen für IDB-Prädikate (Extensionen).
- D.h. Materialisiere die durch P definierten Sichten.
- Eigentlich nur Extension des Prädikates „answer“ wichtig.

# Grundsätzliche Methode (1)

- D.h. berechne minimales Modell als kleinsten Fixpunkt von $\mathsf{T_P}$ iterativ. Einfachster Algorithmus:

  $\mathcal{I} := \emptyset;$

  $\mathcal{I}_{\mathsf{new}} := \mathsf{T_P}(\mathcal{I});$

  **while** $\mathcal{I}_{\mathsf{new}} \neq \mathcal{I}$ **do**

  $\qquad \mathcal{I} := \mathcal{I}_{\mathsf{new}};$

  $\qquad \mathcal{I}_{\mathsf{new}} := \mathsf{T_P}(\mathcal{I});$

  **od**;

  **print** $\mathcal{I}$(answer);

- Implementiere $\mathsf{T_P}$ mit Datenbank-Techniken.

# Grundsätzliche Methode (2)

## Probleme:

- Zuweisung ganzer DB-Zustände?
  Besser jeweils nur einzelne Relationen ändern.
- EDB-Prädikate liegen schon fertig vor.
- Nichtrekursive Regeln jedesmal neu anwenden?
  Allgemein: Möglichst jedes Fakt nur einmal herleiten.
- Oft sind nicht alle Fakten im minimalen Herbrand-Modell relevant zur Beantwortung der Anfrage.
- Häufig nicht interpretiert, sondern compiliert:
  Übersetze dazu Datalog in imperative Programme
  (z.B. relationale Algebra + Kontrollstrukturen)

# Beispiel (Eingabe)

$$
\begin{array}{lcl}
\mathrm{elternteil}(X, Y) & \leftarrow & \mathrm{mutter}(X, Y). \\
\mathrm{elternteil}(X, Y) & \leftarrow & \mathrm{vater}(X, Y). \\
\mathrm{vorfahr}(X, Y) & \leftarrow & \mathrm{elternteil}(X, Y). \\
\mathrm{vorfahr}(X, Z) & \leftarrow & \mathrm{elternteil}(X, Y) \wedge \mathrm{vorfahr}(Y, Z). \\
\mathrm{answer}(X) & \leftarrow & \mathrm{vorfahr}(\mathrm{julia}, X).
\end{array}
$$

% vater und mutter seien hier EDB-Prädikate.

% (D.h. in der Datenbank schon als Relation abgespeichert.)

# Beispiel (Ausgabe)

elternteil $:=$ mutter $\cup$ vater;

vorfahr $:= \emptyset$;

vorfahr_neu $:=$ elternteil;

**while** vorfahr_neu $\neq \emptyset$ **do**

vorfahr $:=$ vorfahr $\cup$ vorfahr_neu;

vorfahr_neu $:= \pi_{\$1,\$3}(\text{elternteil} \underset{\$2=\$1}{\bowtie} \text{vorfahr\_neu})$;

vorfahr_neu $:=$ vorfahr_neu $-$ vorfahr;

**od**;

answer $:= \pi_{\$2}\big(\sigma_{\$1=\text{'julia'}}(\text{vorfahr})\big)$;

**print** answer;

# Beispiel (Aufgabe)

Führen Sie das obige Programm auf folgender Datenbank aus:

| mutter | |
|---|---|
| Kind | Mutter |
| emil | birgit |
| frida | doris |
| julia | frida |

| vater | |
|---|---|
| Kind | Vater |
| emil | arno |
| frida | chris |
| julia | emil |

# EDB- vs. IDB-Prädikate (1)

## Bemerkung:

- Die Trennung zwischen EDB- und IDB-Prädikaten ist für die Semantik nicht wichtig, aber für die Anfrage-Auswertung.
- Bei der Beschreibung der Semantik haben wir angenommen, daß die EDB-Prädikate auch im logischen Programm definiert sind (über Fakten).
- Es gibt normalerweise aber Tausende oder Millionen von Fakten, aber nur wenige Regeln.
- Außerdem ändern sich die Fakten relativ häufig, während man ein einmal übersetztes Anfrage-Programm eventuell mehrfach ausführen will.

# EDB- vs. IDB-Prädikate (2)

- Ein Prädikat p $\in \mathcal{PRED}$ heißt IDB-Prädikat von einem Programm P gdw. P mindestens eine Regel $\mathsf{A} \leftarrow \mathsf{B}_1 \wedge \cdots \wedge \mathsf{B}_m$ mit pred(A) = p und $m \geq 1$ enthält.
- Alle anderen in P vorkommenden Prädikate heißen EDB-Prädikate bezüglich P.
- Wir schreiben $\mathcal{P}_{\mathcal{IDB}}(\mathsf{P})$ für die Menge der IDB-Prädikate von P und $\mathcal{P}_{\mathcal{EDB}}(\mathsf{P})$ für die Menge der EDB-Prädikate.
- $\mathsf{IDB}(\mathsf{P}) := \{\mathsf{A} \leftarrow \mathsf{B}_1 \wedge \cdots \wedge \mathsf{B}_m \in \mathsf{P} \mid \mathsf{pred}(\mathsf{A}) \in \mathcal{P}_{\mathcal{IDB}}(\mathsf{P}), m \geq 0\}.$

  $\mathsf{EDB}(\mathsf{P}) := \{\mathsf{A} \mid \mathsf{A} \leftarrow \mathsf{true} \in \mathsf{P}, \mathsf{pred}(\mathsf{A}) \in \mathcal{P}_{\mathcal{EDB}}(\mathsf{P})\}.$

# EDB- vs. IDB-Prädikate (3)

## Interpreter mit Unterscheidung EDB-IDB:

$\mathcal{I}_{\mathsf{db}} := \mathsf{EDB}(\mathsf{P});$

$\mathsf{P}' := \mathsf{IDB}(\mathsf{P});$

$\mathcal{I} := \emptyset;$

$\mathcal{I}_{\mathsf{new}} := \mathsf{T}_{\mathsf{P}'}(\mathcal{I} \cup \mathcal{I}_{\mathsf{db}});$

**while** $\mathcal{I}_{\mathsf{new}} \neq \mathcal{I}$ **do**

$\quad\quad \mathcal{I} := \mathcal{I}_{\mathsf{new}};$

$\quad\quad \mathcal{I}_{\mathsf{new}} := \mathsf{T}_{\mathsf{P}'}(\mathcal{I} \cup \mathcal{I}_{\mathsf{db}});$

**od**;

**print** $\mathcal{I}(\mathsf{answer});$

# Prädikat-Abhängigkeitsgraph (1)

**Definition:**

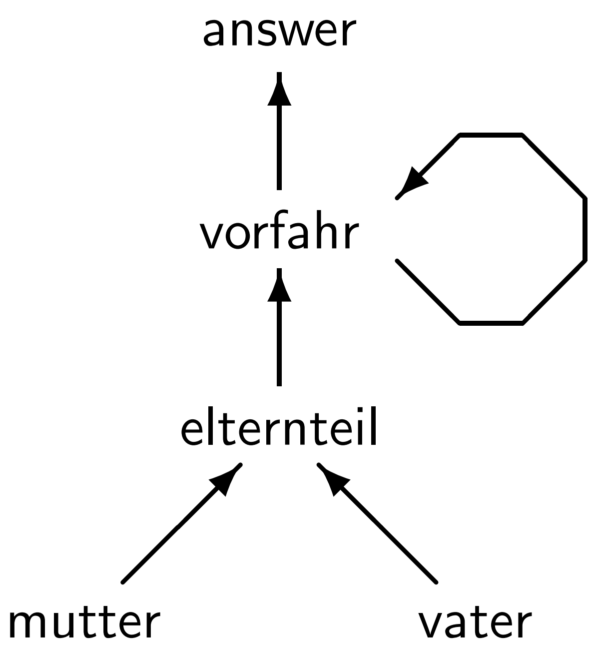
Der Prädikat-Abhängigkeitsgraph eines logischen Programms P ist der gerichtete Graph $\mathcal{DG}(\mathsf{P}) = (\mathsf{V}, \mathsf{E})$ mit

- Die Knotenmenge V ist die Menge aller in P vorkommenden Prädikate.
- Es gibt eine Kante von p nach q, d.h. $(\mathsf{p}, \mathsf{q}) \in \mathsf{E} \subseteq \mathsf{V} \times \mathsf{V}$, gdw. es eine Regel $\mathsf{A} \leftarrow \mathsf{B}_1 \wedge \cdots \wedge \mathsf{B}_m$ in P und $i \in \{1, \ldots, m\}$ gibt, so daß $\mathsf{pred}(\mathsf{A}) = \mathsf{q}$ und $\mathsf{pred}(\mathsf{B}_i) = \mathsf{p}$.

# Prädikat-Abhängigkeitsgraph (2)

## Beispiel:

$$\begin{array}{lll}
\mathrm{elternteil}(X,Y) & \leftarrow & \mathrm{mutter}(X,Y). \\
\mathrm{elternteil}(X,Y) & \leftarrow & \mathrm{vater}(X,Y). \\
\mathrm{vorfahr}(X,Y) & \leftarrow & \mathrm{elternteil}(X,Y). \\
\mathrm{vorfahr}(X,Z) & \leftarrow & \mathrm{elternteil}(X,Y) \wedge \\
 & & \mathrm{vorfahr}(Y,Z). \\
\mathrm{answer}(X) & \leftarrow & \mathrm{vorfahr}(\mathrm{julia},X).
\end{array}$$



# Prädikat-Abhängigkeitsgraph (3)

## Definitionen:

Sei ein logisches Programm P gegeben.

- Ein Prädikat p hängt von einem Prädikat q ab gdw. es in $\mathcal{DG}(\mathsf{P})$ einen Weg (aus ein oder mehr Kanten) von q nach p gibt.
- Ein Prädikat p heißt rekursiv gdw. p von sich selbst abhängt.
- Die Knoten $\{\mathsf{p}_1, \ldots, \mathsf{p}_k\}$ einer starken Zusammenhangs-komponente von $\mathcal{DG}(\mathsf{P})$ heißen rekursive Clique.
- Zwei Prädikate p und q, die zur gleichen rekursiven Clique gehören, heißen wechselseitig rekursiv.

# Prädikat-Abhängigkeitsgraph (4)

## Aufgabe:

Berechnen Sie den Prädikat-Abhängigkeitsgraph und (wechselseitig) rekursive Prädikate für folgendes Programm:

| | | |
|---|---|---|
| empty_until(Rule, $0$) | $\leftarrow$ | rule(Rule, _, _). |
| empty_until(Rule, Next) | $\leftarrow$ | empty_until(Rule, Pos) $\wedge$ |
| | | succ(Pos, Next) $\wedge$ |
| | | rhs(Rule, Next, NonTerm) $\wedge$ |
| | | empty(NonTerm). |
| empty(Left) | $\leftarrow$ | rule(Rule, Left, RightSize) $\wedge$ |
| | | empty_until(Rule, RightSize). |

# Prädikat-Abhängigkeitsgraph (5)

## Rekursive Regel:

Eine Regel $\mathsf{A} \leftarrow \mathsf{B}_1 \wedge \cdots \wedge \mathsf{B}_m$ heißt rekursiv gdw. es ein $i \in \{1, \ldots, m\}$ gibt, so daß $\mathsf{pred}(\mathsf{B}_i)$ von $\mathsf{pred}(\mathsf{A})$ abhängt.

## Notation:

Sei P ein logisches Programm und $\mathsf{C} \subseteq \mathcal{P}_{\mathcal{IDB}}(\mathsf{P})$.

- Wir schreiben $\mathsf{rec}(\mathsf{P}, \mathsf{C})$ für die Menge der rekursiven Regeln über Prädikate aus P.
- Entsprechend schreiben wir $\mathsf{nrec}(\mathsf{P}, \mathsf{C})$ für die Menge der nichtrekursiven Regeln über Prädikate aus C.

# Prädikat-Abhängigkeitsgraph (6)

## Reduzierter Prädikat-Abhängigkeitsgraph:

Der reduzierte Prädikat-Abhängigkeitsgraph eines Programms P ist der gerichtete azyklische Graph $\mathcal{RG}(\mathsf{P}) = (\hat{\mathsf{V}}, \hat{\mathsf{E}})$ mit

- $\hat{\mathsf{V}} := \{\mathsf{C} \subseteq \mathcal{P}_{\mathcal{IDB}}(\mathsf{P}) \mid \mathsf{C}$ ist rekursive Clique$\}$
  $\cup \{\mathsf{p} \in \mathcal{P}_{\mathcal{IDB}}(\mathsf{P}) \mid \mathsf{p}$ ist nichtrekursiv$\}$.
- $(\mathsf{C}_1, \mathsf{C}_2) \in \hat{\mathsf{E}}$ gdw. $\mathsf{C}_1 \neq \mathsf{C}_2$ und es gibt $\mathsf{p}_1, \mathsf{p}_2 \in \mathcal{P}_{\mathcal{IDB}}(\mathsf{P})$ mit
  - $\mathsf{p}_1 \in \mathsf{C}_1$ bzw. $\mathsf{p}_1 = \mathsf{C}_1$,
  - $\mathsf{p}_2 \in \mathsf{C}_2$ bzw. $\mathsf{p}_2 = \mathsf{C}_2$,
  - es gibt eine Kante $\mathsf{p}_1 \longrightarrow \mathsf{p}_2$ in $\mathcal{DG}(\mathsf{P})$.

# Prädikat-Abhängigkeitsgraph (7)

## Auswertungsreihenfolge für Prädikate:

- Sei P ein Programm und $\mathcal{RG}(\mathsf{P}) = (\hat{\mathsf{V}}, \hat{\mathsf{E}})$ sein reduzierter Prädikat-Abhängigkeitsgraph. Eine Reihenfolge seiner Knoten $\mathsf{C}_1, \ldots, \mathsf{C}_k$ heißt Prädikat-Auswertungsreihenfolge für P gdw.

  für alle $i, j \in \{1, \ldots, k\}$: $(\mathsf{C}_i, \mathsf{C}_j) \in \hat{\mathsf{E}} \implies i < j$

  (d.h. es gibt nur vorwärts gerichtete Kanten).

- Man bekommt so eine Auswertungsreihenfolge durch topologisches Sortieren von $\hat{\mathsf{V}}$ bezüglich der Ordnungsrelation $\hat{\mathsf{E}}$.
  So eine Auswertungsreihenfolge existiert immer.

# Prädikat-Abhängigkeitsgraph (8)

## Interpreter mit Auswertungsreihenfolge:

$\mathcal{I}_{\mathsf{db}} :=$ EDB(P);

Berechne Auswertungsreihenfolge $\mathsf{C}_1, \ldots, \mathsf{C}_k$ für P;

$\mathcal{I} := \emptyset$;

**for** $i := 1$ **to** $k$ **do**

/* Erweitere $\mathcal{I}$ durch Auswertung von $\mathsf{C}_i$: */

... siehe nächste Folie ...

**od**;

**print** $\mathcal{I}$(answer);

# Prädikat-Abhängigkeitsgraph (9)

**if** $\mathsf{C}_i$ ist nichtrekursives Prädikat p **then**

$\mathcal{I} := \mathcal{I} \cup \mathsf{T}_{\mathsf{nrec}(\mathsf{P},\mathsf{C}_i)}(\mathcal{I} \cup \mathcal{I}_{\mathsf{db}});$

**else** $/*$ $\mathsf{C}_i$ ist rekursive Clique $*/$

$\mathcal{I}_{\mathsf{new}} := \mathsf{T}_{\mathsf{nrec}(\mathsf{P},\mathsf{C}_i)}(\mathcal{I} \cup \mathcal{I}_{\mathsf{db}});$

**while** $\mathcal{I}_{\mathsf{new}} \neq \emptyset$ **do**

$\mathcal{I} := \mathcal{I} \cup \mathcal{I}_{\mathsf{new}};$

$\mathcal{I}_{\mathsf{new}} := \mathsf{T}_{\mathsf{rec}(\mathsf{P},\mathsf{C}_i)}(\mathcal{I} \cup \mathcal{I}_{\mathsf{db}});$

$\mathcal{I}_{\mathsf{new}} := \mathcal{I}_{\mathsf{new}} - \mathcal{I};$

**od**;

**fi**;

# Übersetzung von Regeln (1)

- Im obigen Algorithmus müssen Anweisungen wie

$$\mathcal{I} := \mathcal{I} \cup \mathsf{T}_{\mathsf{nrec}(\mathsf{P},\mathsf{C}_i)}(\mathcal{I} \cup \mathcal{I}_{\mathsf{db}});$$

  noch weiter verfeinert werden.

- Bei nichtrekursiven Programm-Komponenten bzw. bei der seminaiven Auswertung (s.u.) rekursiver Komponenten können Regeln nacheinander ausgeführt werden. D.h.

$$\mathcal{I} := \mathcal{I} \cup \mathsf{T}_{\{\mathcal{R}_1,\ldots,\mathcal{R}_n\}}(\mathcal{I})$$

  ist (falls Kopf- und Rumpf-Prädikate disjunkt) äquivalent zu

$$\mathcal{I} := \mathcal{I} \cup \mathsf{T}_{\{\mathcal{R}_1\}}(\mathcal{I});\ \ldots\ \mathcal{I} := \mathcal{I} \cup \mathsf{T}_{\{\mathcal{R}_n\}}(\mathcal{I});$$

# Übersetzung von Regeln (2)

## Beispiel:

- $\mathsf{p}(\mathsf{X}, \mathsf{a}) \leftarrow \mathsf{q}(\mathsf{X}, \mathsf{Y}) \land \mathsf{r}(\mathsf{Y}, \mathsf{b})$.
- Selektion für Konstanten im Rumpf
  (bzw. gleiche Variablen in einem Literal),

  Verbund über gemeinsamen Variablen verschiedener Literale,

  Projektion auf den Kopf.
- $\mathsf{p} := \mathsf{p} \cup \pi_{\$1,\mathsf{a}}\big(\mathsf{q} \underset{\$2=\$1}{\bowtie} \sigma_{\$2=\mathsf{b}}(\mathsf{r})\big)$.
- ```
  INSERT INTO p (SELECT DISTINCT B1.$1, 'a'
                 FROM   q B1, r B2
                 WHERE  B1.$2=B2.$1 AND B2.$2='b')
  ```
  ```

# Übersetzung von Regeln (3)

- Direkte Implementierung mittels „Nested Loop Join“:

**foreach** $\mathsf{B}_1 \in \mathsf{q}$ **do**

  $\mathsf{X} := \mathsf{B}_1[1]$; $\mathsf{Y} := \mathsf{B}_1[2]$;

  **foreach** $\mathsf{B}_2 \in \mathsf{r}$ **do**

    **if** $\mathsf{B}_2[1] = \mathsf{Y}$ **and** $\mathsf{B}_2[2] =$ 'b' **then**

      /∗ Tupel in p einfügen ∗/

      . . . siehe nächste Folie . . .

    **fi**;

  **od**;

**od**;

# Übersetzung von Regeln (4)

Duplicate := false;

**foreach** A $\in$ p **do**

  **if** A$[1] =$ X **and** A$[2] =$ 'a' **then**

    Duplicate := true; **break**;

  **fi**;

**od**;

**if not** Duplicate **then**

  **new** A$'$; A$'[1] =$ X; A$'[2] =$ 'a';

  **append** A$'$ **to** p;

**fi**;

# Übersetzung von Regeln (5)

## Aufgaben:

- Welche Alternativen gibt es zur Auswertung dieser Regel? (Z.B. andere Join-Methode, Verwendung von Indexen.)
  Andere Methode für die Duplikat-Eliminierung?
- Falls dies die einzige/erste Regel über p ist, könnte man dann auf den Duplikattest verzichten? Oder braucht man noch weiteres Wissen (Schlüsselbedingung)?
  Stören Duplikate überhaupt (bei nichtrekursiven Programmen)?
- Welche Vor- und Nachteile hat die Verwendung eines kommerziellen DBS (Übersetzung in SQL) im Vergleich zur direkten Übersetzung in Pascal etc.?

# Übersetzung von Regeln (6)

## Bemerkungen:

- Es ist nicht schwer, ein Programm zur Übersetzung von Regeln in Relationen-Algebra, SQL, etc. zu schreiben.
  Eine gute Übersetzung in Programmcode (Pascal, C, abstrakte Maschine), die einen DB-Optimierer enthält und mit Externspeicher umgeht, ist dagegen schon schwer.
- Man muß sich beim ersten Auftreten einer Variablen ihren Wert (oder eine Referenz darauf) merken, und bei jedem weiteren Auftreten einen Vergleich ausführen.
  Bei der Übersetzung in SQL wird man eine Tabelle haben, in der z.B. nach Verarbeitung des ersten Rumpfliterals für die Variable X `"B1.$1"` eingetragen ist.

# Nichtrekursive Programme (1)

- Hat man mehrere Regeln über ein Prädikat, so kann man die entsprechenden Algebra-Ausdrücke/SQL-Anfragen mit $\cup$/`UNION` zusammenfügen.
- Bei einem nichtrekursiven Programm kann man im Algebra-Ausdruck für das Prädikat answer sukzessive die Ausdrücke für die IDB-Prädikate einsetzen — man bekommt dann einen einzigen Algebra-Ausdruck, in dem nur noch EDB-Prädikate vorkommen.
- In SQL92 geht es ganz analog, sonst verwende man Sichten. Man kann aber (etwas komplizierter) zu jedem nichtrekursiven Programm auch eine SQL86-Anfrage erzeugen (Distributivgesetz für $\times$ und $\cup$ bzw. „Unfolding“ im Programm).

# Nichtrekursive Programme (2)

- Da der Optimierer dann mehr Freiheitsgrade hat (er ist nicht mehr durch die Einteilung in verschiedene Prädikat-Definitionen gebunden) wird eine solche Anfrage meist effizienter ausgewertet als eine Hintereinanderausführung aller Regeln.
- Außerdem muß so nicht mehr jedes IDB-Prädikat materialisiert werden. Datenbanksysteme verwenden intern häufig Iteratoren, die jeweils auf Anforderung das nächste Tupel berechnen (nicht gleich das ganze Zwischenergebnis). Da der Hauptspeicher begrenzt ist, ist es wichtig, Zwischenergebnisse nur wenn unbedingt nötig (z.B. bei Sortierungen) explizit abzuspeichern.

# Nichtrekursive Programme (3)

## Aufgabe:

Geben Sie einen Algebra-Ausdruck und eine SQL-Anfrage für das folgende Programm an:

$$\begin{array}{lcl}
\mathsf{elternteil}(\mathsf{X},\mathsf{Y}) & \leftarrow & \mathsf{mutter}(\mathsf{X},\mathsf{Y}). \\
\mathsf{elternteil}(\mathsf{X},\mathsf{Y}) & \leftarrow & \mathsf{vater}(\mathsf{X},\mathsf{Y}). \\
\mathsf{großvater}(\mathsf{X},\mathsf{Z}) & \leftarrow & \mathsf{elternteil}(\mathsf{X},\mathsf{Y}) \wedge \mathsf{vater}(\mathsf{Y},\mathsf{Z}). \\
\mathsf{answer}(\mathsf{X}) & \leftarrow & \mathsf{großvater}(\mathsf{julia},\mathsf{X}).
\end{array}$$

Falls Sie „Informationssysteme II“ gehört haben:
Führen Sie eine algebraische Optimierung durch und diskutieren Sie mögliche Auswertungspläne.

# Rekursive Programme

- In diesem Fall hat man zwei verschiedene Interpretationen:

$$\mathcal{I}_{\mathsf{new}} := \mathsf{T}_{\mathsf{rec}(\mathsf{P},\mathsf{C}_i)}(\mathcal{I} \cup \mathcal{I}_{\mathsf{db}});$$

  Sogar drei, aber $\mathcal{I}$ und $\mathcal{I}_{\mathsf{db}}$ definieren unterschiedliche Prädikate. Dagegen haben $\mathcal{I}_{\mathsf{new}}$ und $\mathcal{I}$ nicht-disjunkte Definitionsbereiche.

- Lösung: Man verwendet nur eine Interpretation (Datenbank), aber führt mehrere Varianten eines Prädikates ein. Z.B.

$$\mathsf{vorfahr}(\mathsf{X},\mathsf{Z}) \leftarrow \mathsf{elternteil}(\mathsf{X},\mathsf{Y}) \wedge \mathsf{vorfahr}(\mathsf{Y},\mathsf{Z})$$

  wird übersetzt wie

$$\mathsf{vorfahr\_neu}(\mathsf{X},\mathsf{Z}) \leftarrow \mathsf{elternteil}(\mathsf{X},\mathsf{Y}) \wedge \mathsf{vorfahr}(\mathsf{Y},\mathsf{Z}).$$

# Compiler für Datalog (1)

- Man kommt von einem Interpreter zu einem Compiler durch partielle Auswertung.
- Man führt den Interpreter aus, soweit möglich. Wenn immer Befehle von den tatsächlichen Daten abhängen, druckt man sie aus, anstatt sie wirklich auszuführen.

## Notation/Voraussetzungen:

- $\mathcal{P}^{\mathsf{new}}_{\mathcal{IDB}}(\mathsf{P}) := \{\mathsf{p\_new} \mid \mathsf{p} \in \mathcal{P}_{\mathcal{IDB}}(\mathsf{P}),\ \mathsf{p}\ \text{rekursiv}\}$.
- translate(C) übersetze Regelmengen wie oben beschrieben.
- make_head_new(C) ändere jedes Prädikat p im Kopf einer Regel in p_new.

# Compiler für Datalog (2)

**foreach** p $\in \mathcal{P}_{\mathcal{IDB}}(\mathsf{P}) \cup \mathcal{P}^{\mathsf{new}}_{\mathcal{IDB}}(\mathsf{P})$ **do**

    **print** "`CREATE TABLE p (...)`"; **od**;

Berechne Auswertungsreihenfolge $\mathsf{C}_1, \ldots, \mathsf{C}_k$ für P;

**for** $i := 1$ **to** $k$ **do** /* Werte $\mathsf{C}_i$ aus: */

    ... siehe nächste Folie ...

**od**;

**print** "`SELECT * FROM` answer";

**foreach** p $\in \mathcal{P}_{\mathcal{IDB}}(\mathsf{P}) \cup \mathcal{P}^{\mathsf{new}}_{\mathcal{IDB}}(\mathsf{P})$ **do**

    **print** "`DROP TABLE p`"; **od**;

# Compiler für Datalog (3)

**if** $\mathsf{C}_i$ ist nichtrekursives Prädikat p **then**

translate$\big($nrec$(\mathsf{P}, \mathsf{C}_i)\big)$;

**else** /∗ $\mathsf{C}_i$ ist rekursive Clique ∗/

translate$\big($make_head_new$($nrec$(\mathsf{P}, \mathsf{C}_i))\big)$;

**print** `"WHILE "`;

print_while_cond$(\mathsf{C}_i)$;

**print** `" DO "`;

print_while_body$(\mathsf{C}_i, \mathsf{P})$;

**print** `" OD;"`;

**fi**;

# Compiler für Datalog (4)

**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 für Datalog (5)

**procedure** print_while_body($C_i$, P):

  **foreach** p $\in C_i$ **do**

    **print** "p := p $\cup$ p_new;";

  **od**;

  translate(make_head_new(rec(P, $C_i$)));

  **foreach** p $\in C_i$ **do**

    **print** "p_new := p_new $-$ p;";

  **od**;

# Seminaive Auswertung (1)

- Der erste Algorithmus, der einfach den $T_P$-Operator iteriert, bis sich nichts mehr ändert, heißt „naive Bottom-Up Auswertung“.
- Schlecht: Bei jedem Schleifendurchlauf werden die schon vorher bekannten Fakten noch einmal hergeleitet.
- Bei nichtrekursiven Regeln haben wir das Ziel erreicht, daß jede anwendbare Regelinstanz nur einmal angewendet wird.
- Natürlich möchten wir eigentlich jedes Fakt nur einmal herleiten. Wenn ein Fakt aus verschiedenen Regelinstanzen folgt, ist das aber schwierig.
- Bei rekursiven Programmen ist aber noch alles beim alten.

# Seminaive Auswertung (2)

## Bisheriges Übersetzungsergebnis:

vorfahr_neu $:=$ elternteil;

**while** vorfahr_neu $\neq \emptyset$ **do**

  vorfahr $:=$ vorfahr $\cup$ vorfahr_neu;

  vorfahr_neu $:= \pi_{\$1,\$3}($elternteil $\underset{\$2=\$1}{\bowtie}$ vorfahr$)$;

  vorfahr_neu $:=$ vorfahr_neu $-$ vorfahr;

**od**;

# Seminaive Auswertung (3)

## Bisherige (naive) Auswertung, 1. Iteration:

$$\text{vorfahr\_neu}(X, Z) \leftarrow \text{elternteil}(X, Y) \wedge \text{vorfahr}(Y, Z)$$

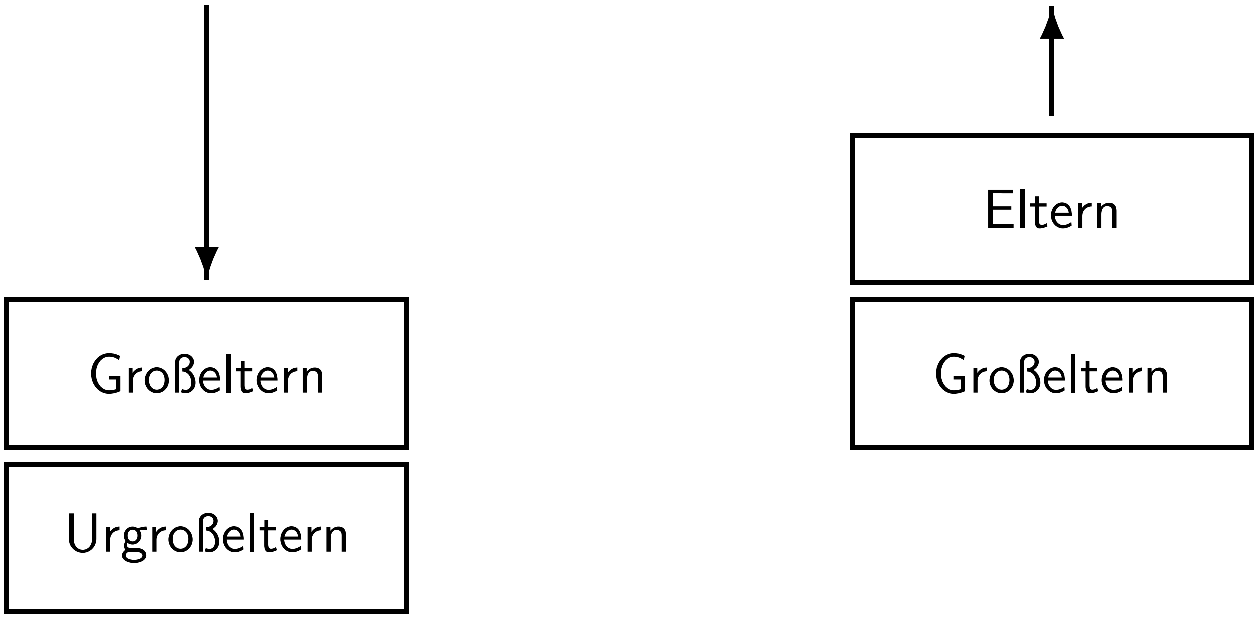
Eltern

Großeltern

# Seminaive Auswertung (4)

## Bisherige (naive) Auswertung, 2. Iteration:

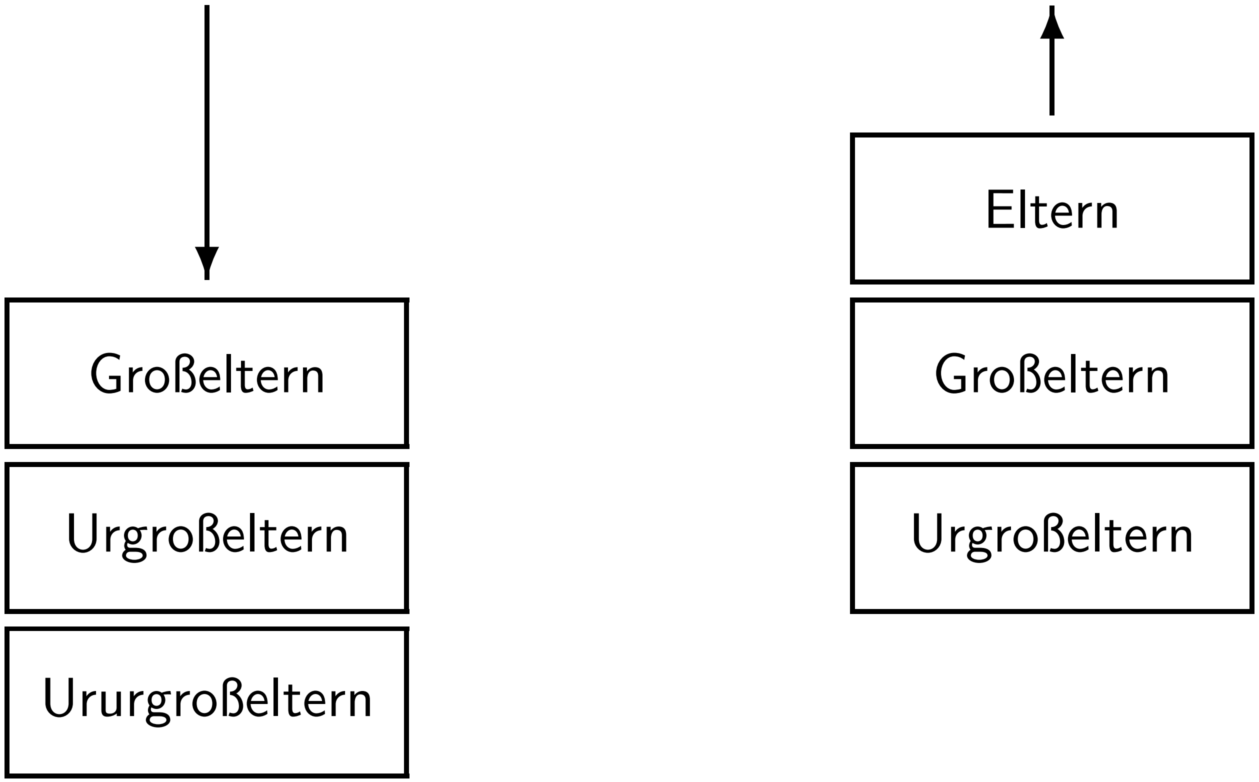
$$\mathsf{vorfahr\_neu}(X,Z) \leftarrow \mathsf{elternteil}(X,Y) \wedge \mathsf{vorfahr}(Y,Z)$$



# Seminaive Auswertung (5)

## Bisherige (naive) Auswertung, 3. Iteration:

$$\mathrm{vorfahr\_neu}(X, Z) \leftarrow \mathrm{elternteil}(X, Y) \wedge \mathrm{vorfahr}(Y, Z)$$



# Seminaive Auswertung (6)

- Abhilfe: Seminaive/Differentielle Auswertung.
- Für rekursives Rumpfliteral nur bei der letzten Iteration neu hergeleitete Tupel einsetzen.
- In diesem Fall also vorfahr durch vorfahr_neu ersetzen:

$$\text{vorfahr\_neu} := \pi_{\$1,\$3}(\text{elternteil} \underset{\$2=\$1}{\bowtie} \text{vorfahr\_neu});$$

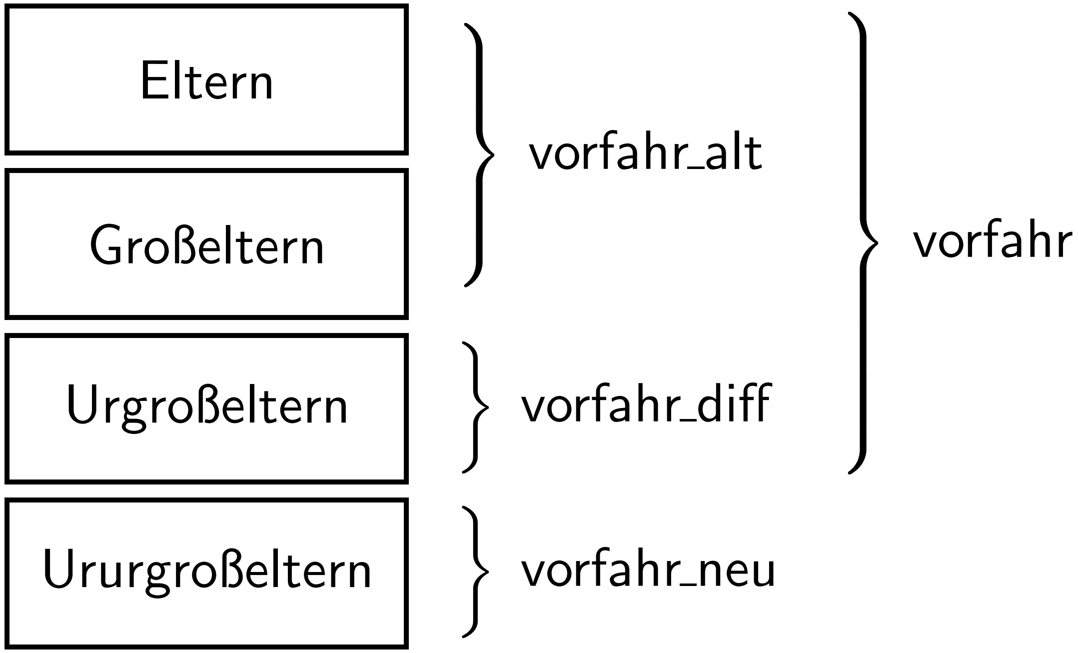
- Im allgemeinen Fall ist es etwas komplizierter:
  - Bei mehreren rekursiven Rumpfliteralen muß man auch Kombinationen alter mit neuen Tupel berücksichtigen.
  - Man möchte vermeiden, daß die gleiche Relation in Kopf und Rumpf verwendet wird.

# Seminaive Auswertung (7)

- Die seminaive Auswertung wird beschrieben durch eine Programm-Transformation unter Verwendung neuer, system-definierter Prädikate.
- Zu jedem Prädikat p werden noch drei Varianten eingeführt:
  - p_old: Bei der letzten Iteration schon vorhandene Tupel.
  - p_diff: Bei der letzen Iteration neue Tupel.
  - p: Wert von p nach der letzten Iteration (p_old $\cup$ p_diff).
  - p_new: In dieser Iteration zu berechnende Tupel.

# Seminaive Auswertung (8)

## Seminaive Auswertung, 3. Iteration:



# Seminaive Auswertung (9)

## Seminaive Auswertung, 3. Iteration:

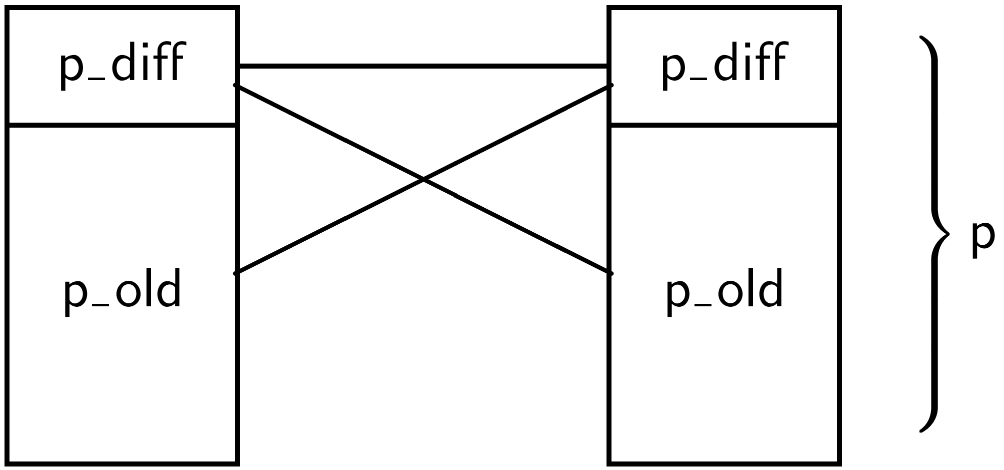
$$\mathrm{vorfahr\_neu}(X, Z) \leftarrow \mathrm{elternteil}(X, Y) \wedge \mathrm{vorfahr\_diff}(Y, Z)$$

Urgroßeltern

Ururgroßeltern

# Seminaive Auswertung (10)

- Beispiel: $p(X, Z) \leftarrow p(X, Y) \wedge p(Y, Z)$.




- Ergebnis der seminaiven Transformation:

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

# Seminaive Auswertung (11)

- Bei $n$ rekursiven Rumpfliteralen sind $2^n - 1$ Kombinationen von „old“ und „diff“ zu betrachten.
- Weil man aber zusätzlich die Vereinigung hat, kann das mit $n$ Regeln aufgeschrieben werden.
- Man spart dabei nur die Kombination lauter „old“-Tupel. Da sich diese Tupel aber schon über eventuell viele Iterationen angesammelt haben, gibt es häufig wesentlich mehr „old“-Tupel als „diff“-Tupel.
- Außerdem ist es für die eventuelle Einsparung von Duplikat-Tests und für Aggregationen mit Duplikaten wichtig, daß jede Regelinstanz nur einmal angewendet wird.

# Seminaive Auswertung (12)

- Initialisierung (vor der **while**-Schleife):

  p_old := $\emptyset$;

  p_diff := Fakten aus nichtrekursiven Regeln;

  p := p_diff;

  p_new := $\emptyset$;

- Nach jeder Iteration:

  p_old := p_old $\cup$ p_diff; /* Äquivalent: p_old := p; */

  p_diff := p_new $-$ p_old; /* p_diff := p_new $-$ p; */

  p := p_old $\cup$ p_diff;

  p_new := $\emptyset$;

# Seminaive Auswertung (13)

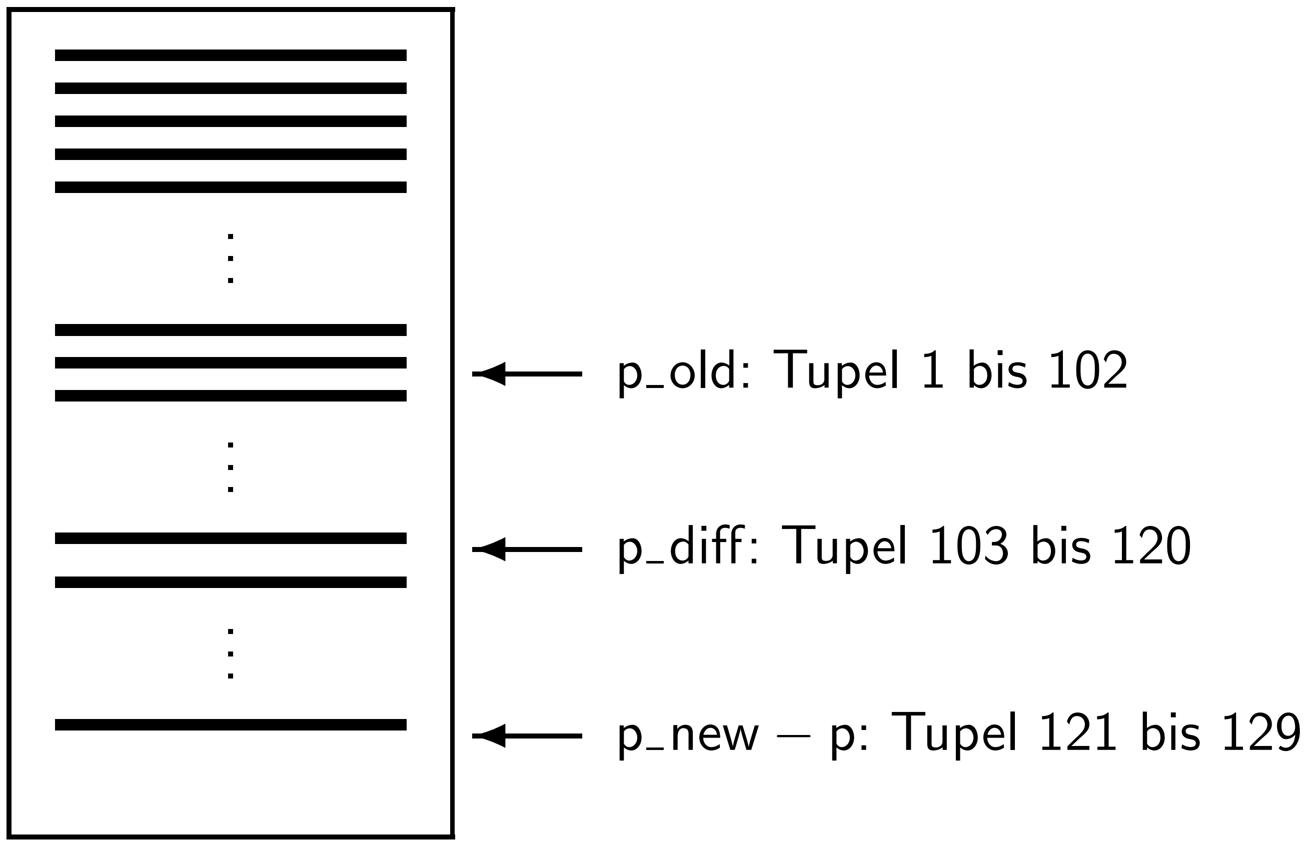
## Verwaltung der Varianten einer Relation:

- Möglichst nicht vier Relationen für jedes rekursive Prädikat.
- Beispiel für eigene Datenstruktur:
  - B-Baum oder Hashtabelle zum Erkennen von Duplikaten. Hergeleitete Tupel gleich prüfen, nur p_new − p abspeichern.
  - Datei oder verkettete Liste, in die neue Tupel immer nur am Ende angefügt werden. Dann können p_old, p_diff und p_new über Positionszeiger in diese Liste implementiert werden (Ende der Liste bei vorletzter / letzter / aktueller Iteration). Nach jeder Iteration müssen nur diese Zeiger umgesetzt werden.

  Tupel zweimal abgespeichert: Im B-Baum in Sortierreihenfolge, in der Liste in Herleitungsreihenfolge.

# Seminaive Auswertung (14)

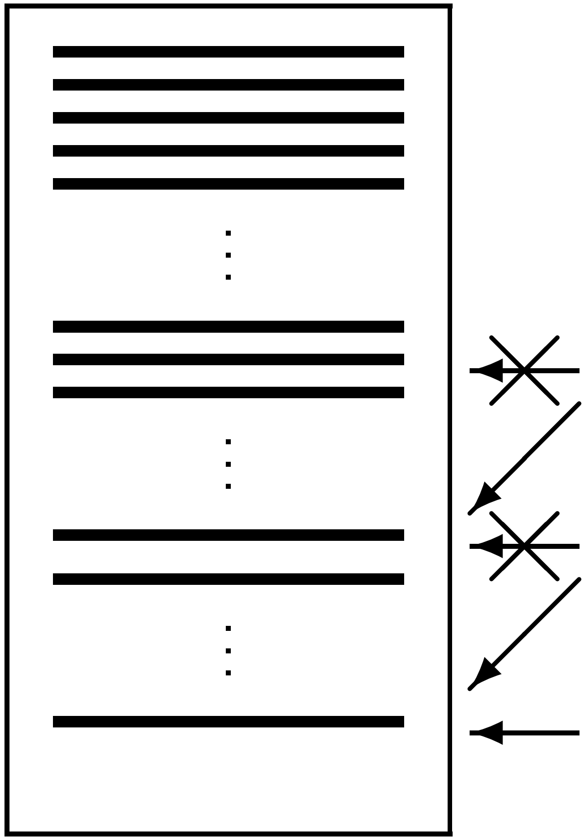
## Verwaltung der Varianten einer Relation, Forts.:



# Seminaive Auswertung (15)

## Verwaltung der Varianten einer Relation, Forts.:

Nach Umschaltung auf nächste Iterationsstufe



p_old: Tupel 1 bis 120

p_diff: Tupel 121 bis 129

p_new – p: Noch keine Tupel

# Schwierigkeiten mit fertigen RDBMS

- IDB-Prädikate sind nur temporär. Es schadet nichts, wenn sie bei einem Systemabsturz verloren gehen. Sie werden auch nur von einem Prozeß gleichzeitig verwendet. Für sie ist also keine Transaktionsverwaltung nötig.
- Für IDB-Prädikate nur `INSERT`. Kein `UPDATE/DELETE`.
- Spezielle Datenstrukturen für seminaive Iteration nützlich. Oder man könnte folgenden Befehl gebrauchen: Füge Tupel in `R` und `S` ein, falls noch nicht in `R` vorhanden.
- Die Größe der Relationen ändert sich im Laufe der Iteration stark. Dies bringt Optimierer durcheinander, die statistische Daten verwenden. Besser: Laufzeit-Optimierung.

# Weitere Optimierungsmöglichkeiten (1)

- Tests haben gezeigt, daß ein recht großer Teil der Laufzeit für Duplikateliminierungen verwendet wird. Es wäre nützlich, Regeln zu erkennen, bei denen keine Duplikate auftreten können.

- Bei rekursiven Regeln bekommt man immer größere und größere Zwischen-Ergebnisse, die man bis zum Ende der Iteration aufbewahrt. Es wäre nützlich, die berechneten Tupel gleich weiterzuverwenden und dann zu löschen.

  Auch hier ist es wichtig, dem Programm anzusehen, daß keine Duplikate in verschiedenen Iterationsstufen berechnet werden. Sonst braucht man die alten Tupel noch für die Terminierung.

# Weitere Optimierungsmöglichkeiten (2)

- Optimal wäre es, wenn nur ein Tupel des rekursiven Prädikates gleichzeitig gebraucht wird.

/* Gesucht: Position des ersten Leerzeichens in String */

| | | |
|---|---|---|
| first_space(Pos) | $\leftarrow$ | no_space_before(Pos) $\land$ |
| | | string(Pos, ' '). |
| no_space_before(1). | | |
| no_space_before(Next) | $\leftarrow$ | no_space_before(Pos) $\land$ |
| | | string(Pos, Char) $\land$ |
| | | Char $\neq$ ' ' $\land$ |
| | | succ(Pos, Next). |