Humboldt – Universität zu Berlin
Institut für Informatik

Holistic Twig Joins: Optimal XML Pattern Matching

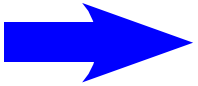
VL XML, XPath, XQuery: Neue Konzepte für Datenbanken

Jörg Pohle, pohle@informatik.hu-berlin.de
Daniel Apelt, apelt@informatik.hu-berlin.de

Übersicht

- Datenstrukturen
- Algorithmen
- Diskussion

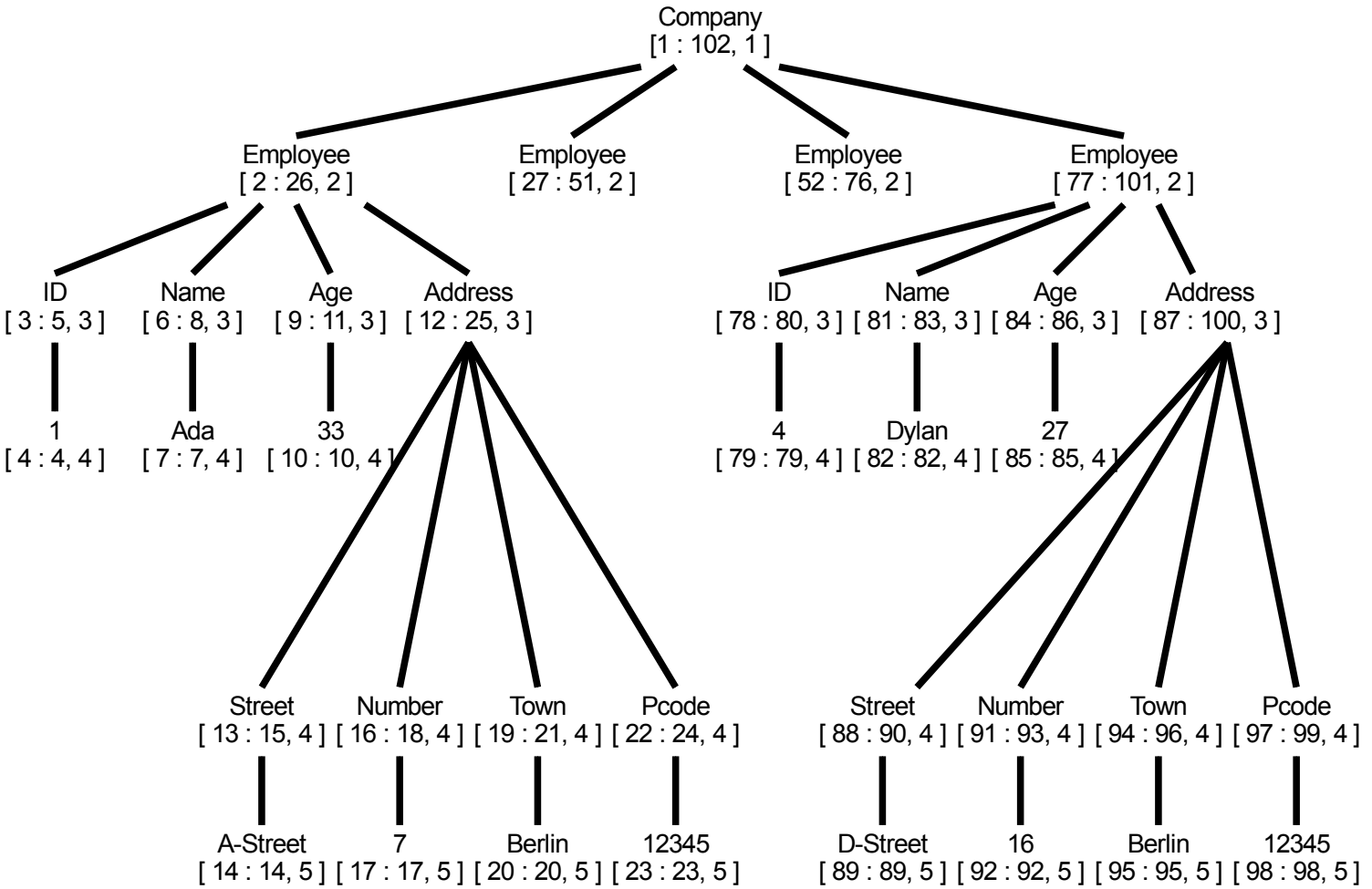
Datenstrukturen



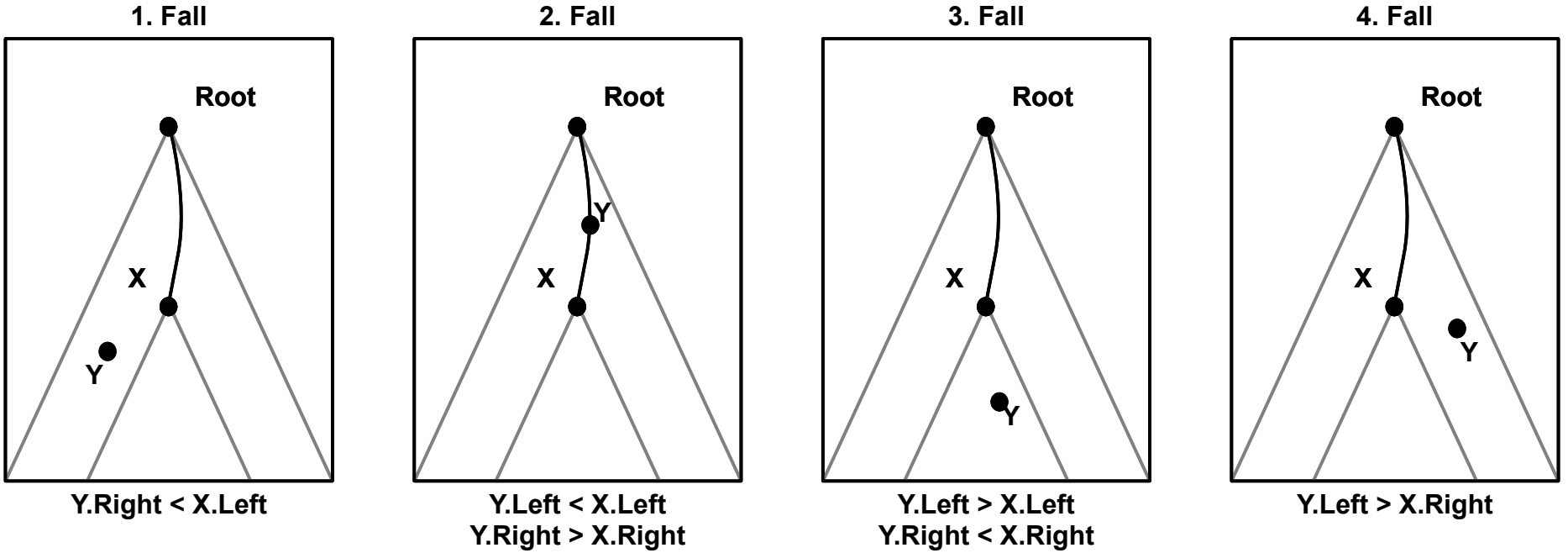
- Datenstrukturen
  - XML-Dokumente
  - twig patterns
  - Streams
  - Stacks
  - XB-Trees
- Algorithmen
- Diskussion

Von Bäumen...

(DocID, LeftPos:RightPos, Level)



XPath-Achsen

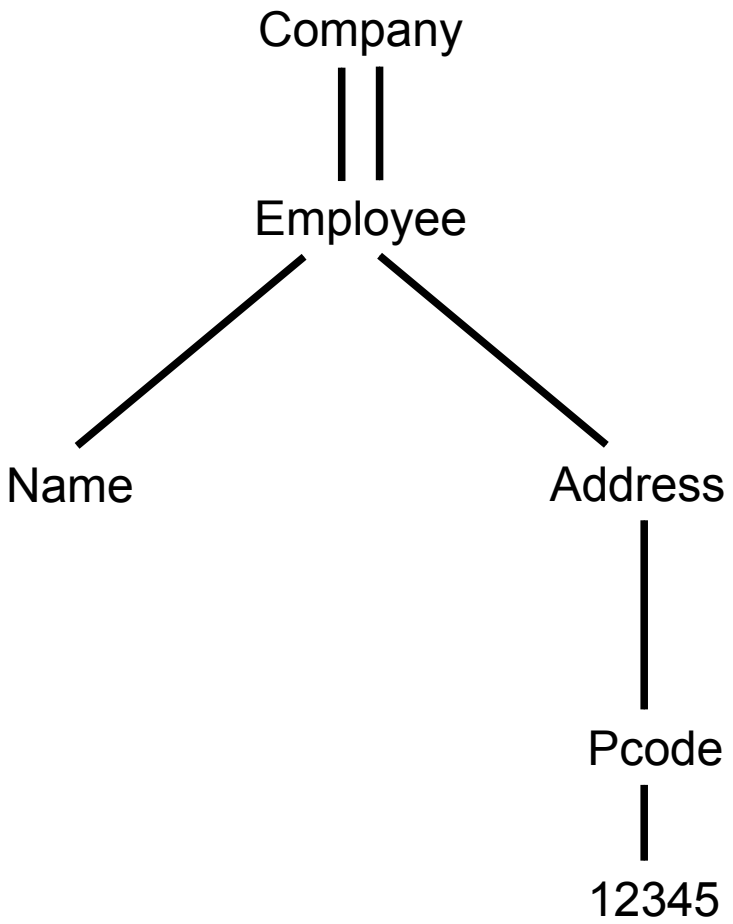


Diese vier Fälle können nur in folgender Reihenfolge auftreten:

(1|2)*3*4*

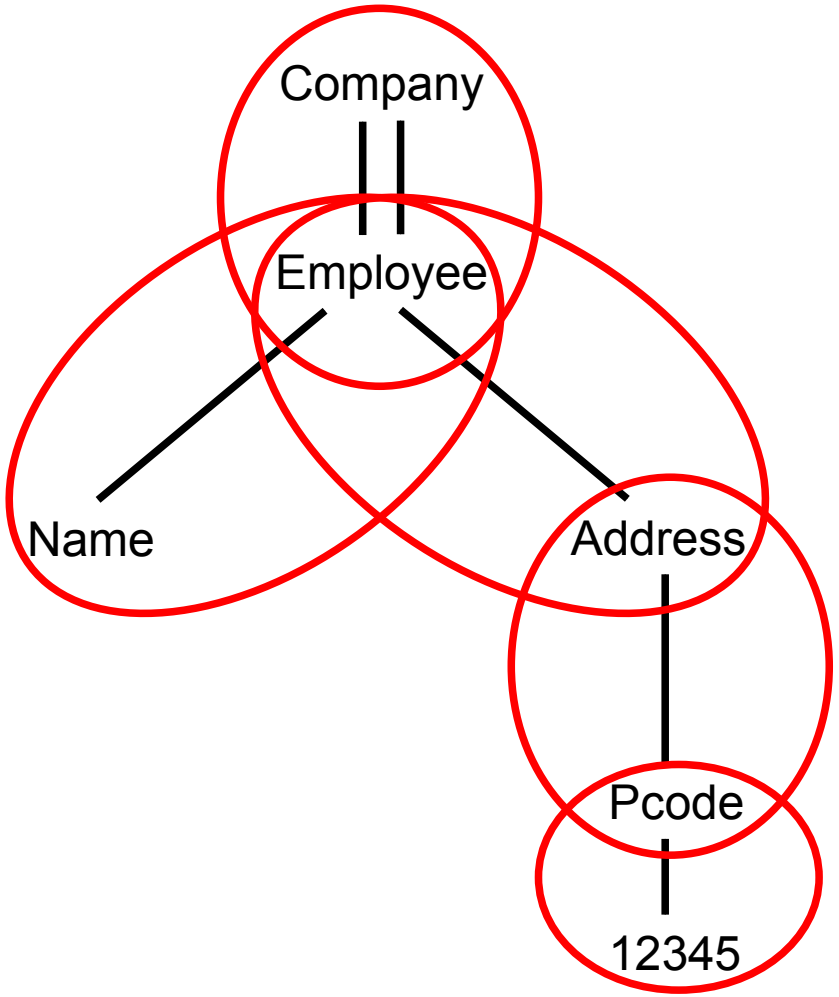
... und Zweigen

//Employee[Address/Pcode=12345]/Name



... und Zweigen

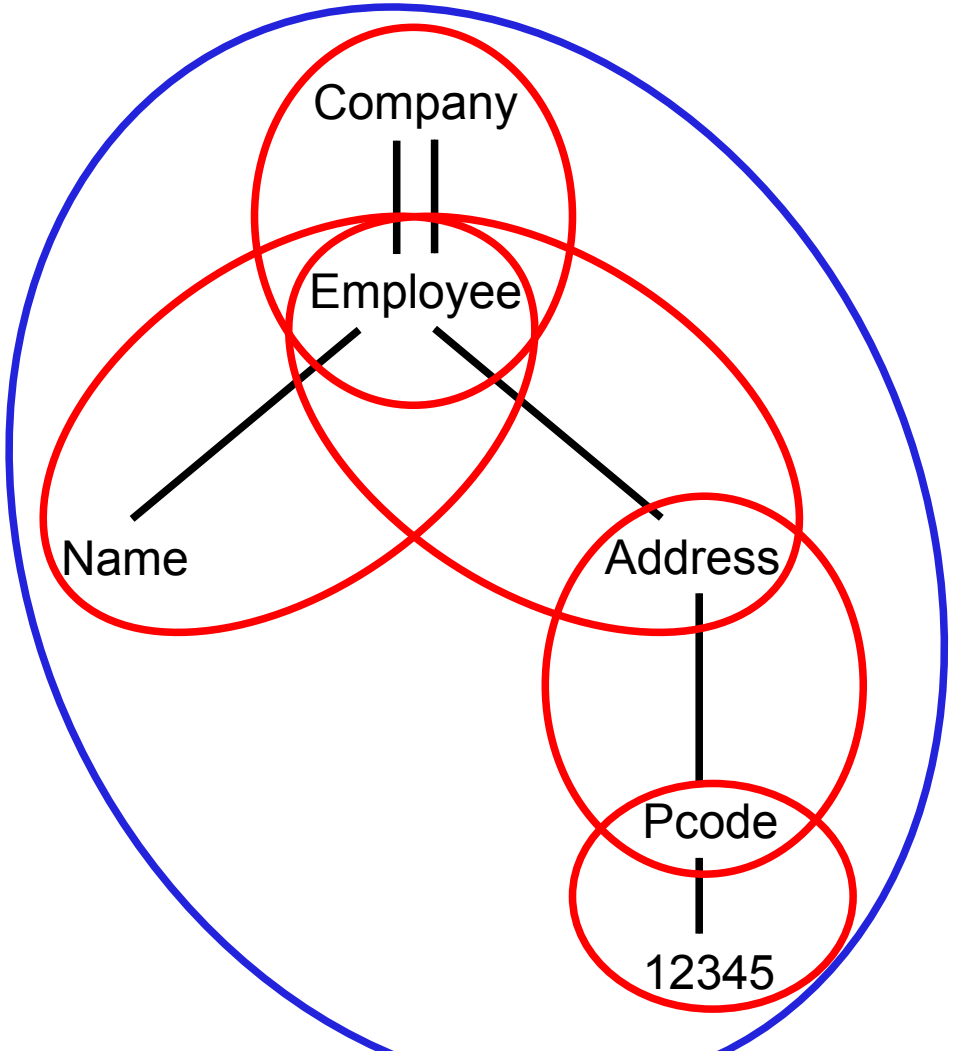
//Employee[Address/Pcode=12345]/Name



Bisherige Ansätze:
- binary patterns
- joins der Teilergebnisse

... und Zweigen

//Employee[Address/Pcode=12345]/Name



Bisherige Ansätze:
- binary patterns
- joins der Teilergebnisse

Alternative:
- Twig Patterns

Holzverarbeitung

Funktionen:

- `isLeaf: Node -> Bool`
- `isRoot: Node -> Bool`
- `parent: Node -> Node`
- `children: Node -> {Node}`
- `subtreeNodes: Node -> {Node}`

Problem:

- Datenstruktur und Operationen können A/B und A//B nicht unterscheiden

Ströme

- pro Anfrageknoten *q* ein Stream T_q
- nach (DocID, LeftPos) sortierte Liste
- enthält alle Datenknoten, die die Bedingungen erfüllen

$$T_E = E_1, E_2, E_3, E_4$$
$$T_A = A_1, A_2, A_3, A_4$$
$$T_P = P_1, P_4$$
$$T_N = N_1, N_2, N_3, N_4$$

Funktionen:

- `eof: Stream -> Bool`
- `advance: Stream -> Null`
- `next: Stream -> Index`
- `nextL: Stream -> Integer`
- `nextR: Stream -> Integer`

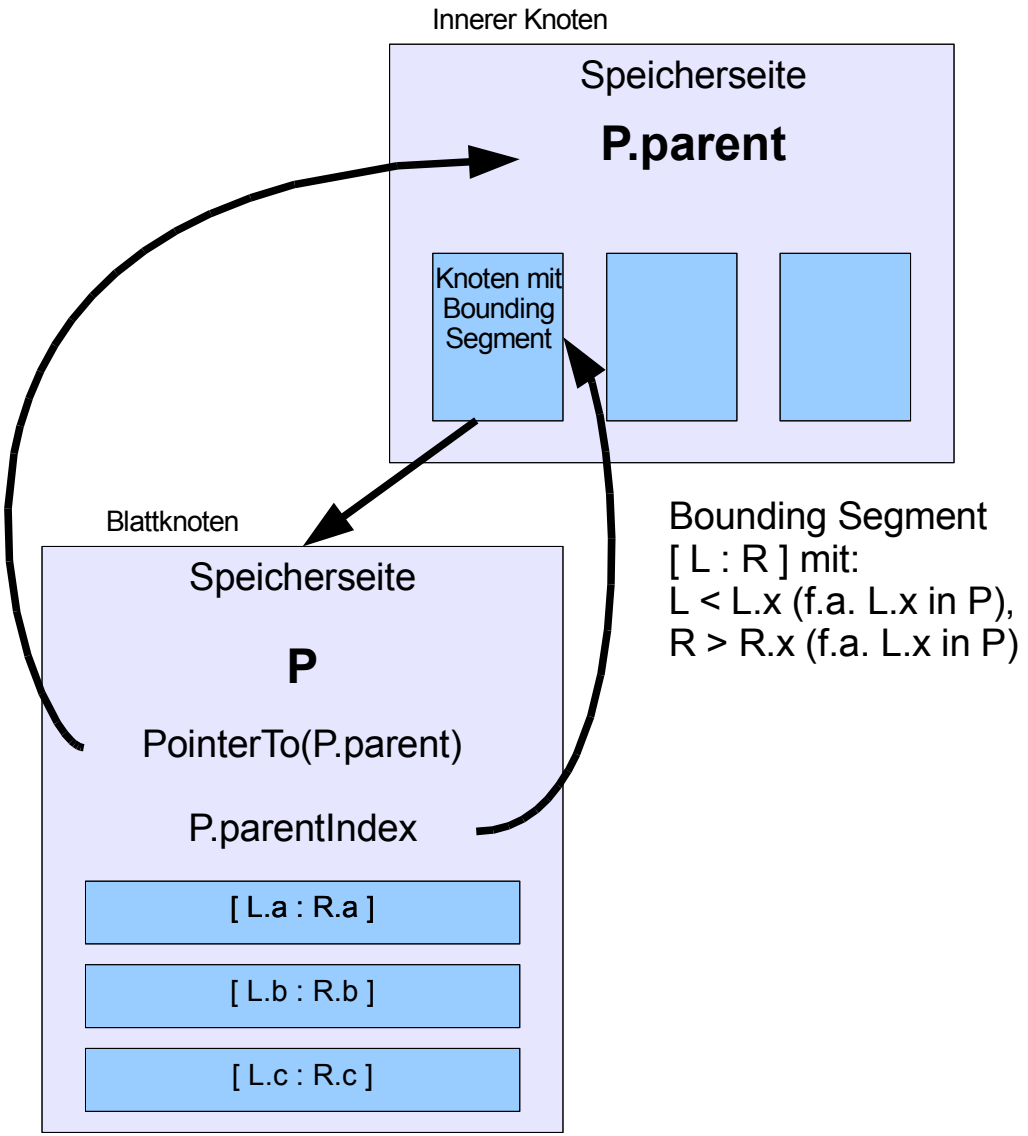
Stacks

- pro Anfrageknoten q ein Stack S_q
- jeder Eintrag ist ein Paar: ein Knoten aus T_q und ein Pointer zu einem Eintrag in $S_{parent(q)}$
- die Einträge im Stack S_q (von bottom zu top) liegen auf einem Wurzel-Blatt-Pfad
- auf den Stacks liegen „compact encodings“ der (Teil-)Ergebnisse der Anfrage

Funktionen:

- `empty: Stack -> Bool`
- `pop: Stack -> Null`
- `push: Stack, Node -> Null`
- `topL: Stack -> Integer`
- `topR: Stack -> Integer`

XB-Trees



Bounding Segment
[L : R] mit:
L < L.x (f.a. L.x in P),
R > R.x (f.a. L.x in P)

- Blattknoten sortiert nach LeftPos-Werten
- jede Speicherseite ist indiziert (actPage)
- jeder Knoten auf der Seite auch (actIndex)
- pointer(actPage, actIndex)

Funktionen:

- `advance`
- `drillDown`

Algorithmen

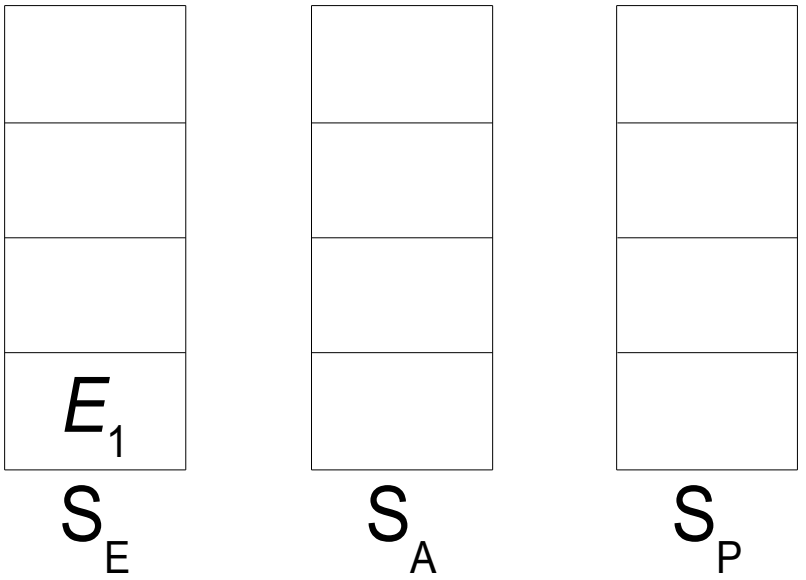
- Datenstrukturen
- Algorithmen
  - PathStack
  - PathMPMJ
  - showSolutions
  - TwigStack
- Diskussion

PathStack

- bestimmt Ergebnismenge einer Pfadanfrage
- twig patterns müssen vorher zerlegt werden
- je ein Stream T_q und Stack S_q pro Anfrageknoten q
- Funktionsweise von PathStack an Beispiel
- `//Employee[Address/Pcode=12345]`

PathStack - Beispiel

```
//Employee[Address/Pcode=12345]
```

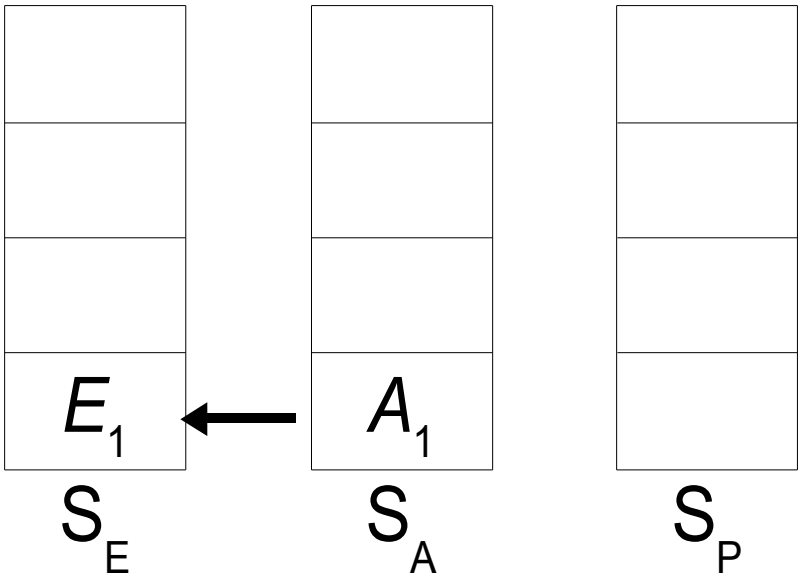


- $q_{min} = E_1$
- cleanStack
- push(S_E, (E_1,*NULL*))
- advance(T_E)

T_E = (E_1[2:26], ***E*$_2$[27:51]**, E_3[52:76], E_4[77:101])
T_A = (A_1[12:25], A_2[37:50], A_3[62:75], A_4[87:100])
T_P = (P_1[22:24], P_4[97:99])

PathStack - Beispiel

```
//Employee[Address/Pcode=12345]
```

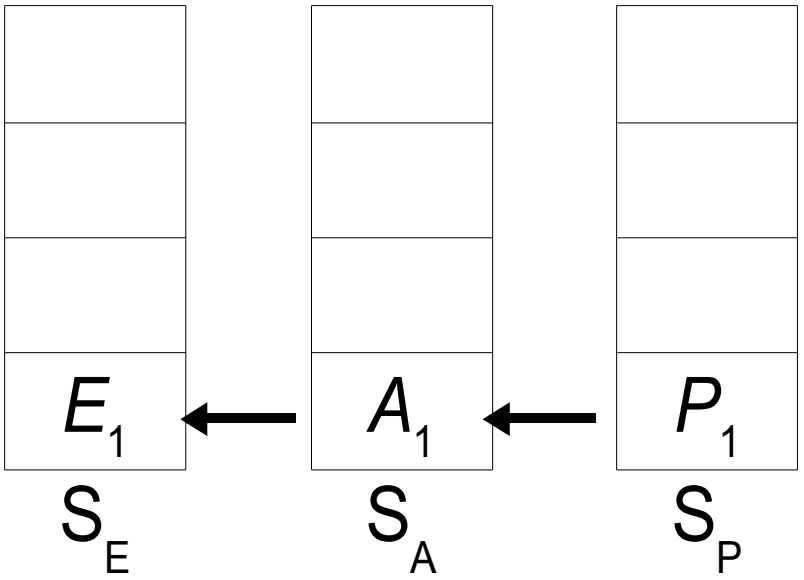


- $q_{min} = A_1$
- cleanStack
- push(S_A, (A_1,E_1))
- advance(T_A)

T_E = (E_1[2:26], E_2[27:51], E_3[52:76], E_4[77:101])
T_A = (A_1[12:25], $\boldsymbol{A_2}$**[37:50]**, A_3[62:75], A_4[87:100])
T_P = (P_1[22:24], P_4[97:99])

PathStack - Beispiel

```
//Employee[Address/Pcode=12345]
```

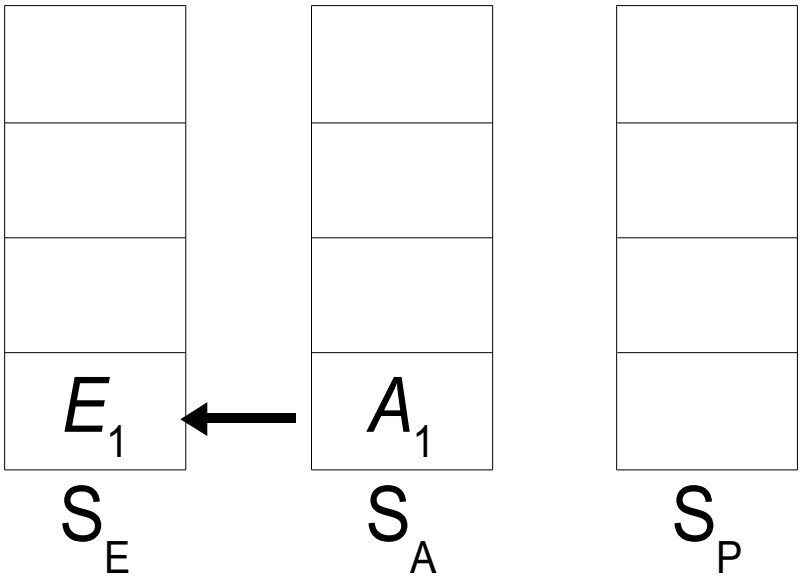


- $q_{min} = P_1$
- cleanStack
- push(S_P, (P_1,A_1))
- advance(T_P)

T_E = (E_1[2:26], E_2[27:51], E_3[52:76], E_4[77:101])
T_A = (A_1[12:25], A_2[37:50], A_3[62:75], A_4[87:100])
T_P = (P_1[22:24], $\boldsymbol{P_4}$**[97:99]**)

PathStack - Beispiel

```
//Employee[Address/Pcode=12345]
```

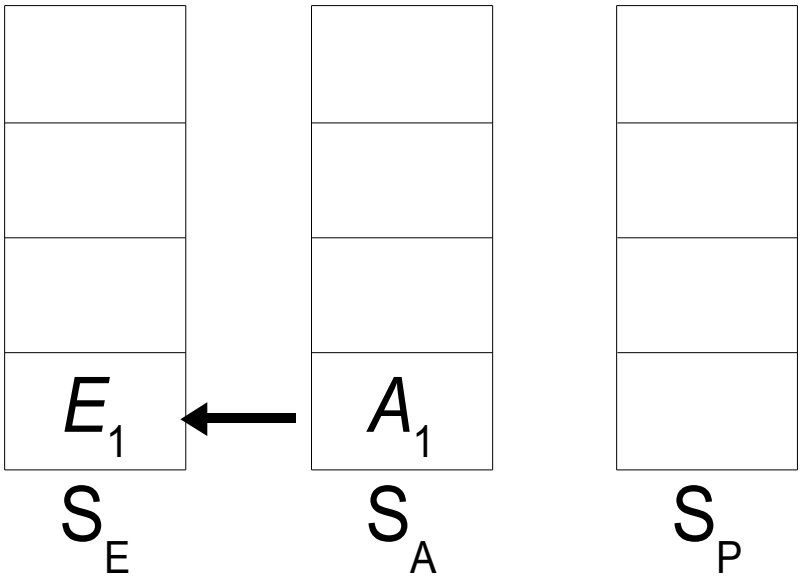


- *isLeaf*(P_1) = *TRUE*
- showSolutions
- pop(S_P)

T_E = (E_1[2:26], E_2[27:51], E_3[52:76], E_4[77:101])
T_A = (A_1[12:25], A_2[37:50], A_3[62:75], A_4[87:100])
T_P = (P_1[22:24], $\boldsymbol{P_4}$**[97:99]**)

PathStack - Beispiel

```
//Employee[Address/Pcode=12345]
```

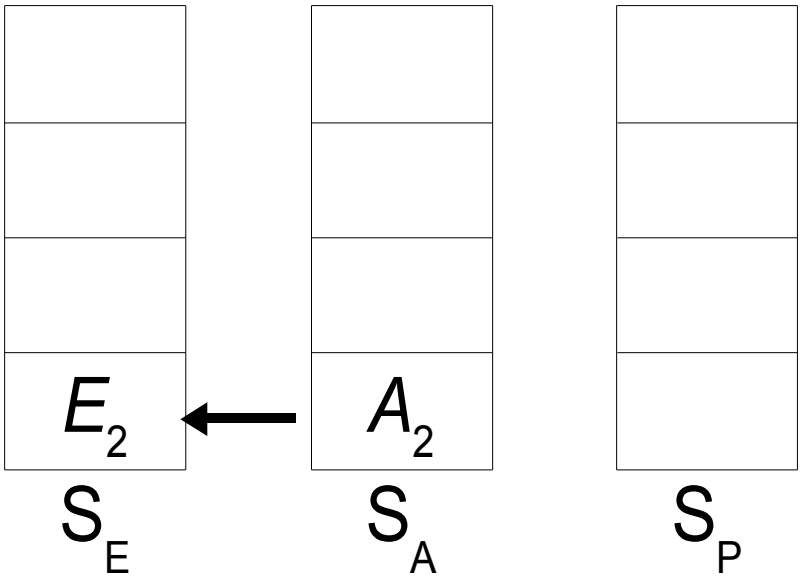


- $q_{min} = E_2$
- CleanStack: E_1, A_1
- pop(S_E), pop(S_A)

T_E = (E_1[2:26], E_2[27:51], E_3[52:76], E_4[77:101])
T_A = (A_1[12:25], A_2[37:50], A_3[62:75], A_4[87:100])
T_P = (P_1[22:24], P_4[97:99])

PathStack - Beispiel

```
//Employee[Address/Pcode=12345]
```

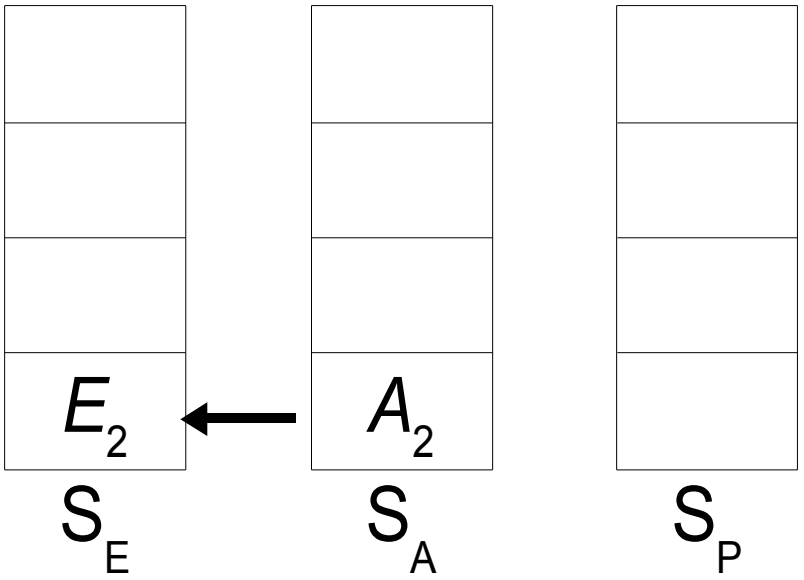


- $q_{min} = E_2$
- push(S_E, (E_2,*NULL*))
- advance(T_E)
- $q_{min} = A_2$
- push(S_A, (A_2,E_2))
- advance(T_A)

T_E = (E_1[2:26], E_2[27:51], E_3[52:76], E_4[77:101])
T_A = (A_1[12:25], A_2[37:50], ***A*$_3$[62:75]**, A_4[87:100])
T_P = (P_1[22:24], P_4[97:99])

PathStack - Beispiel

```
//Employee[Address/Pcode=12345]
```

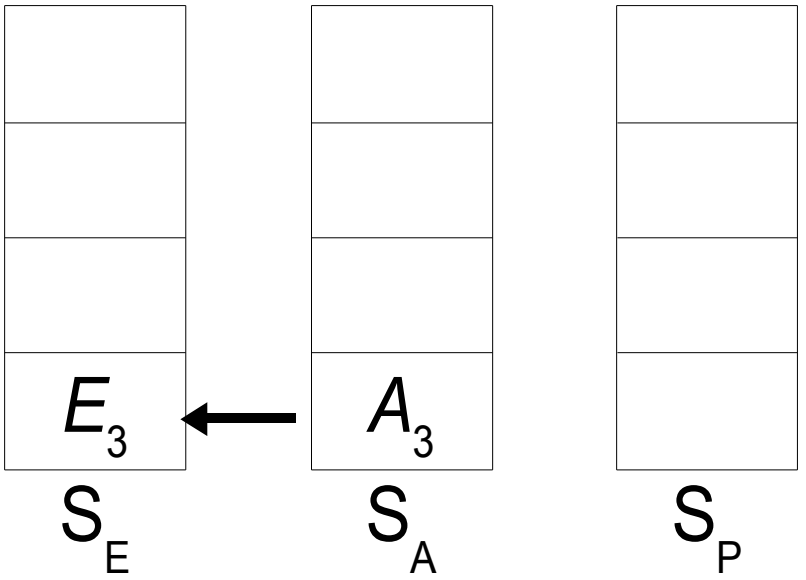


- $q_{min} = E_3$
- CleanStack: E_2, A_2
- pop(S_E), pop(S_A)

T_E = (E_1[2:26], E_2[27:51], E_3[52:76], E_4[77:101])
T_A = (A_1[12:25], A_2[37:50], A_3[62:75], A_4[87:100])
T_P = (P_1[22:24], P_4[97:99])

PathStack - Beispiel

```
//Employee[Address/Pcode=12345]
```

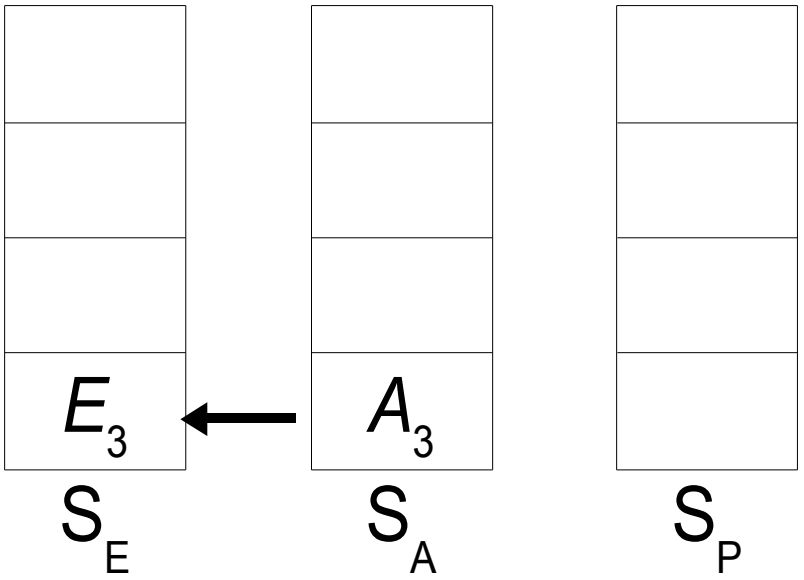


- $q_{min} = E_3$
- push(S_E, (E_3,*NULL*))
- advance(T_E)
- $q_{min} = A_3$
- push(S_A, (A_3,E_3))
- advance(T_A)

T_E = (E_1[2:26], E_2[27:51], E_3[52:76], E_4[77:101])
T_A = (A_1[12:25], A_2[37:50], A_3[62:75], ***A*$_4$[87:100]**)
T_P = (P_1[22:24], P_4[97:99])

PathStack - Beispiel

```
//Employee[Address/Pcode=12345]
```

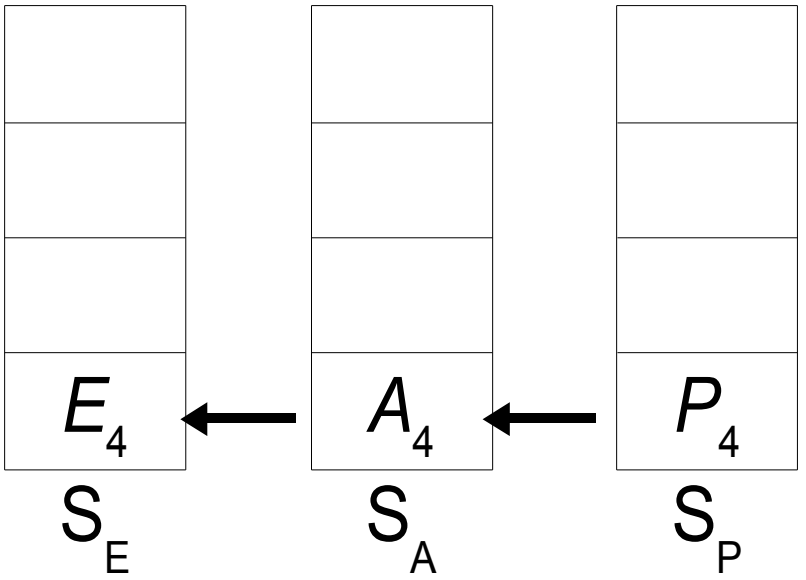


- $q_{min} = E_4$
- CleanStack: E_3, A_3
- pop(S_E), pop(S_A)

T_E = (E_1[2:26], E_2[27:51], E_3[52:76], E_4[77:101])
T_A = (A_1[12:25], A_2[37:50], A_3[62:75], A_4[87:100])
T_P = (P_1[22:24], P_4[97:99])

PathStack - Beispiel

```
//Employee[Address/Pcode=12345]
```

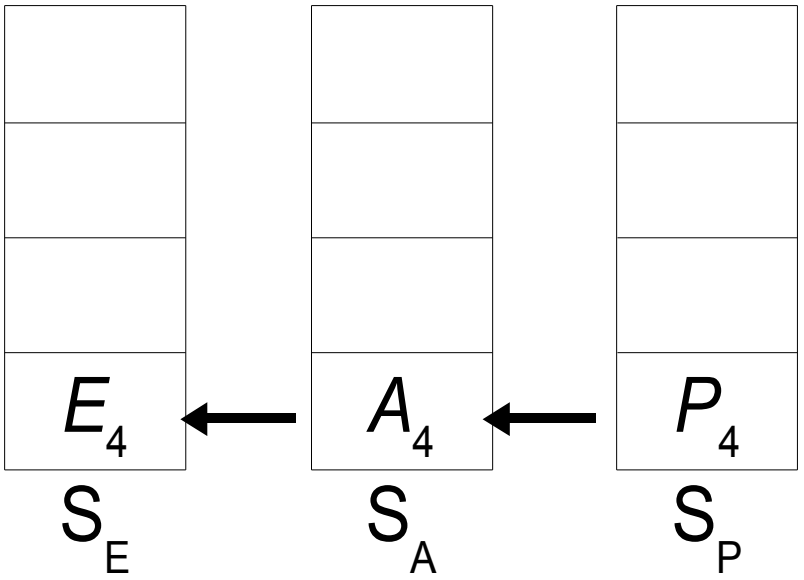


- $q_{min} = E_4$
- push(S_E, (E_4,*NULL*))
- $q_{min} = A_4$
- push(S_A, (A_4,E_4))
- $q_{min} = P_4$
- push(S_P, (P_4,A_4))

T_E = (E_1[2:26], E_2[27:51], E_3[52:76], E_4[77:101])
T_A = (A_1[12:25], A_2[37:50], A_3[62:75], A_4[87:100])
T_P = (P_1[22:24], P_4[97:99])

PathStack - Beispiel

```
//Employee[Address/Pcode=12345]
```



- *isLeaf*(P_4) = TRUE
- showSolutions

T_E = (E_1[2:26], E_2[27:51], E_3[52:76], E_4[77:101])
T_A = (A_1[12:25], A_2[37:50], A_3[62:75], A_4[87:100])
T_P = (P_1[22:24], P_4[97:99])

PathStack - Beispiel

```
//Employee[Address/Pcode=12345]
```

Lösungen:

$E_1 \leftarrow A_1 \leftarrow P_1$

$E_4 \leftarrow A_4 \leftarrow P_4$

Problem:

- Lösung ist nicht äquivalent zur Lösung der XQuery-Anfrage (**alle** Daten der gesuchten „Employee“)
- weitere Berechnungen/Suchoperationen sind notwendig

PathMPMJ

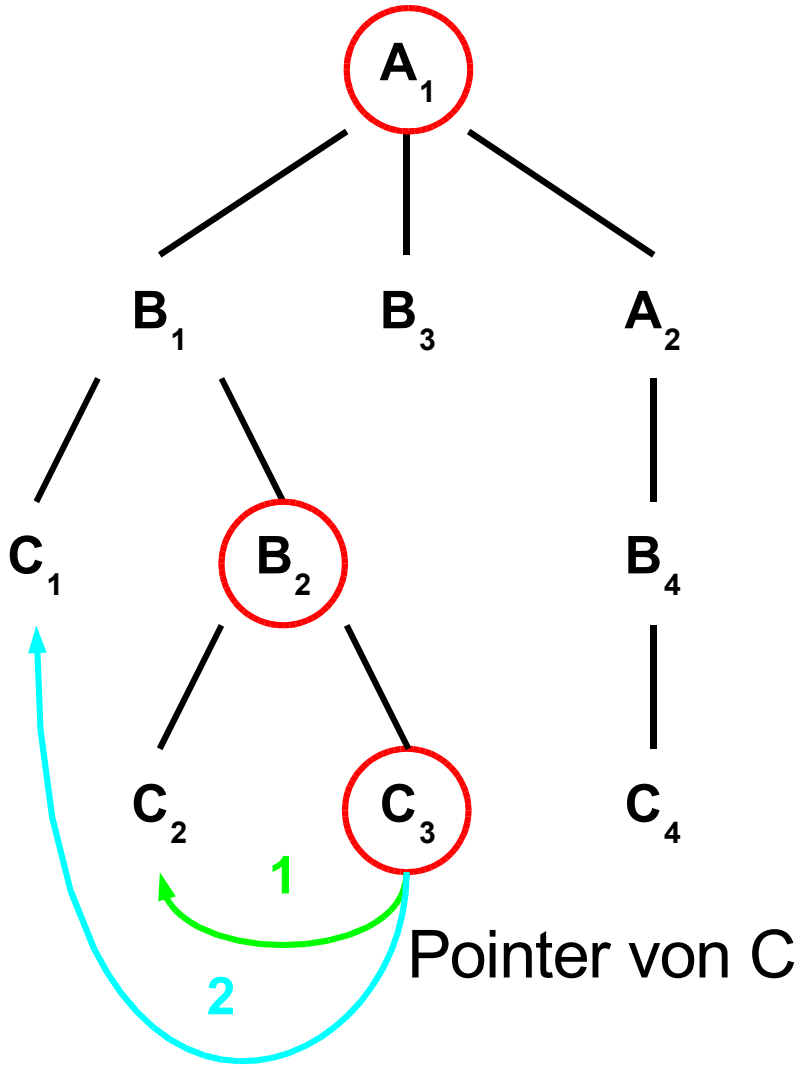
Naiv:

- im Prinzip geschachtelte for-Schleifen über die Elemente von T_{qi}
- Backtracking jeweils zu $q_{n.m}$, so daß $q_{1.a}$, ..., $q_{n-1.x}$, $q_{n.m}$ Anfangsstück einer Lösung sein kann
- jeweils ein Pointer pro Stream für Backtracking

Alternativ:

- wir haben den Ansatz nicht ganz verstanden
- dennoch hier folgender Erklärungsansatz:

PathMPMJ



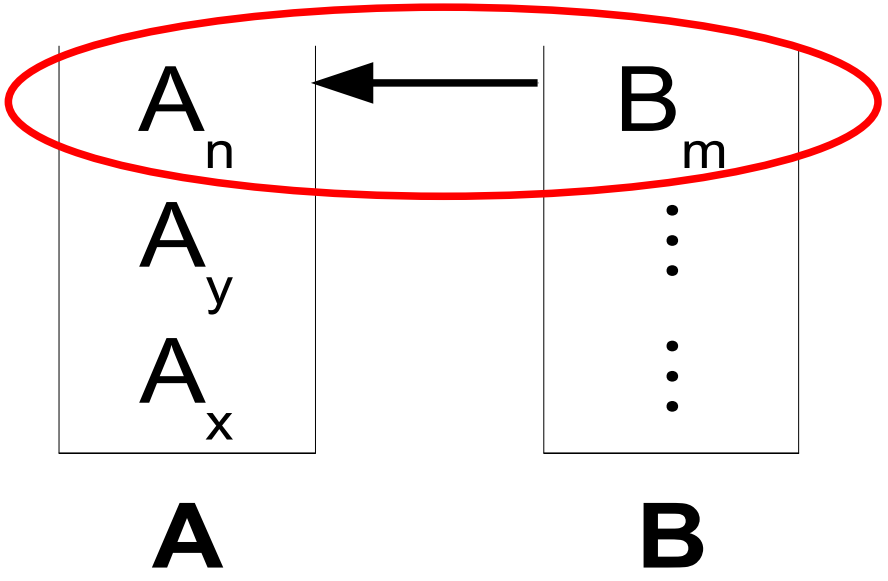
Query: A/B/C

$T_A = A_1, A_2$
$T_B = B_1, B_2, B_3, B_4$
$T_C = C_1, C_2, C_3, C_4$

Lösungen:

A_1, B_1, C_1
A_2, B_4, C_4

ShowSolutionWithBlocking



Lösung durch Stackabarbeitung:

A_n ... B_m
A_y ... B_m
A_x ... B_m

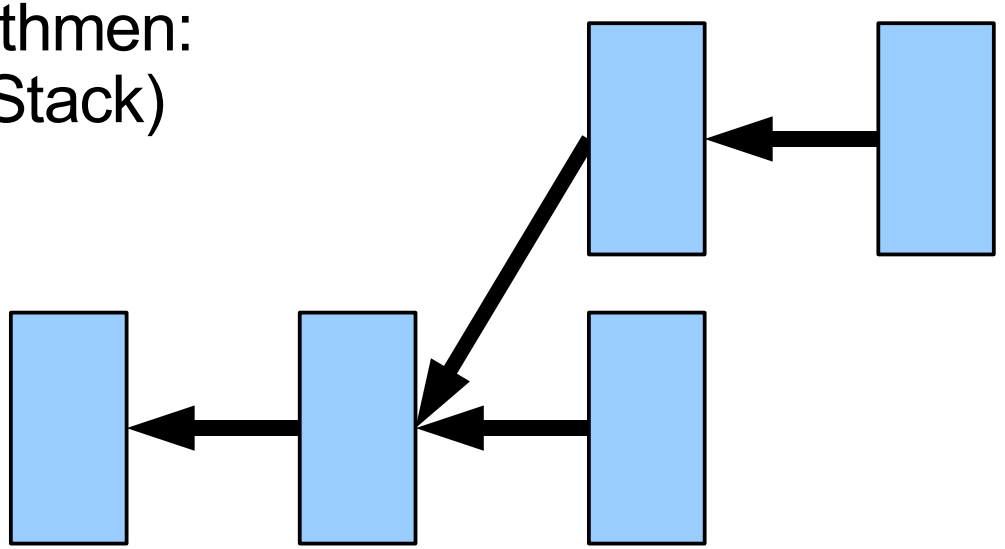
Richtige Lösung:
(ordnungserhaltend)

A_x ... B_m
A_y ... B_m
A_n ... B_m

TwigStack

Erweiterung von PathStack:
- statt einzelner Pfade wird das ganze twig pattern betrachtet
- Stacks werden als Baum organisiert

Zwei verschieden Algorithmen:
- mit Streams (wie PathStack)
- mit XB-Trees



Company/Employee[Address/Pcode=12345]/Name

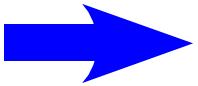
TwigStack

`getNext` ist die wesentliche Erweiterung:

- startet auf Wurzelknoten der Query
- bevor ein Knoten K_{qx} auf den Stack geschrieben wird
  - $\forall\ q_i \in \text{children}(q_x)$: $K_{qi} \in \text{subtreeNodes}(K_{qx})$ und
  - die gleiche Eigenschaft haben auch alle K_{qi}
  - dies wird rekursiv überprüft
- wird ein Knoten K_q auf den Stack gelegt, werden Knoten, die nicht Nachfolger von K_q sein können nur von S_q und $S_{parent(q)}$ gelöscht
  - die anderen Knoten könnten noch zur Konstruktion anderer Pfade eines Zweiges benötigt werden
  - nicht speicherplatzoptimal

Diskussion

- Datenstrukturen
- Algorithmen
- Diskussion



Diskussion

Vorteile:

- kleinere Zwischenergebnisse (als bei binären join-Methoden)
- optimal für ancestor/descendant-Relationen
- Verringerung der Anfragezeiten

Nachteile:

- Datenstruktur der twig patterns kann A/B und A//B nicht unterscheiden
- experimentelle Ergebnisse werden durch einseitige Testdaten verzerrt
    1. eher A//B-Query als A/B-Query betrachtet, daher binäre joins überproportional schlecht (die Autoren räumen dies ein)
    2. Szenario konstruiert

Diskussion

Tree-Pattern Queries on a Lightweight XML Processor
Mirella M. Moro, Zografoula Vagena, Vassilis J. Tsotras
Proceedings of the 31st VLDB Conference,Trondheim, Norway, 2005

Vergleich von 5 Ansätzen:

- stackbasierte, sequenzbasierte und DFA-basierte Verfahren
- das DFA-basierte ist das langsamste Verfahren
- Indizierung bringt sehr viel
- eine indexbasierte Reimplementation von TwigStack war das schnellste Verfahren
- ohne Index kommt es stark auf die Daten und Anfragen an

Literatur

Holistic Twig Joins: Optimal XML Pattern Matching.
Nicolas Bruno, Nick Koudas, Divesh Srivastava, 2002

Holistic Twig Joins: Optimal XML Pattern Matching, Technical Report
Nicolas Bruno, Nick Koudas, Divesh Srivastava, 2002

Tree-Pattern Queries on a Lightweight XML Processor
Mirella M. Moro, Zografoula Vagena, Vassilis J. Tsotras
Proceedings of the 31st VLDB Conference,Trondheim, 2005

Twig-Query-Processing auf Bäumen
Philipp Hussels
Seminar: Graphenmanagement in Datenbanken, HU, 2006

Vielen Dank für Eure Aufmerksamkeit