ADT 2010

Other Approaches to XQuery Processing

Stefan Manegold
Stefan.Manegold@cwi.nl
http://www.cwi.nl/~manegold/

Schedule

- 09.11.2010:
  - RDBMS back-end support for XML/XQuery (1/2):
    - Document Representation (*XPath Accelerator*, *Pre/Post plane*)
- 16.11.2010:
  - XPath navigation (*Staircase Join*)
  - XQuery to Relational Algebra Compiler:
    - Item- & Sequence- Representation
    - Efficient FLWoR Evaluation (*Loop-Lifting*)
    - Optimization
- 23.11.2010:
  - RDBMS back-end support for XML/XQuery (2/2):
    - Updateable Document Representation
- ***30.11.2010:***
  - ***Other (DB-) approaches to XML/XQuery processing***

Topics

- **Other approaches & techniques (***selection, far from complete!***)**
  - Document storage / tree encoding:
    - ORDPATH
    - DataGuides
  - XPath processing:
    - Tree patterns, holistic twig joins

DataGuides

- XPath Accelerator, ORDPATH & similar encoding schemes
  - encode the document's tree structure in the node ranks/labels they assign

- DataGuides
  - Developed in the context of Lore project (DBMS for semi-structured data)
    - Stanford University, Goldman & Widom, VLDB 1997
  - encode the document's tree structure in relation names
  - Observation:
    - Each node is uniquely identified by its path from the root
    - Paths of siblings with equal tag names can be unified,
    - Provided we keep their relative order (*rank*) explicitly

DataGuides

Definition

given a semistructured data instance DB,
a *DataGuide* for DB is a graph G s.t.:

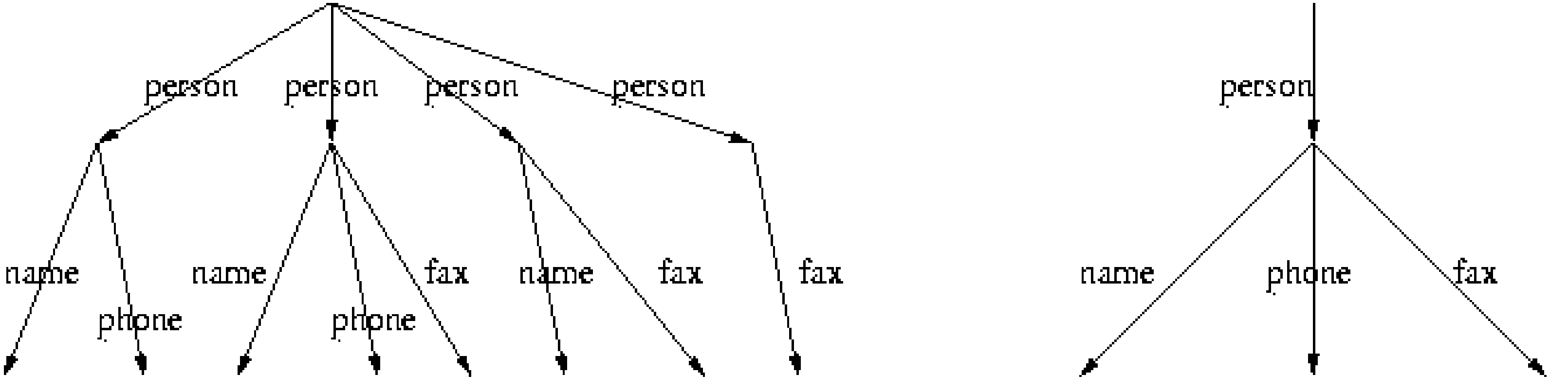
- every path in DB also occurs in G

- every path in G occurs in DB

- every path in G is unique

DataGuides

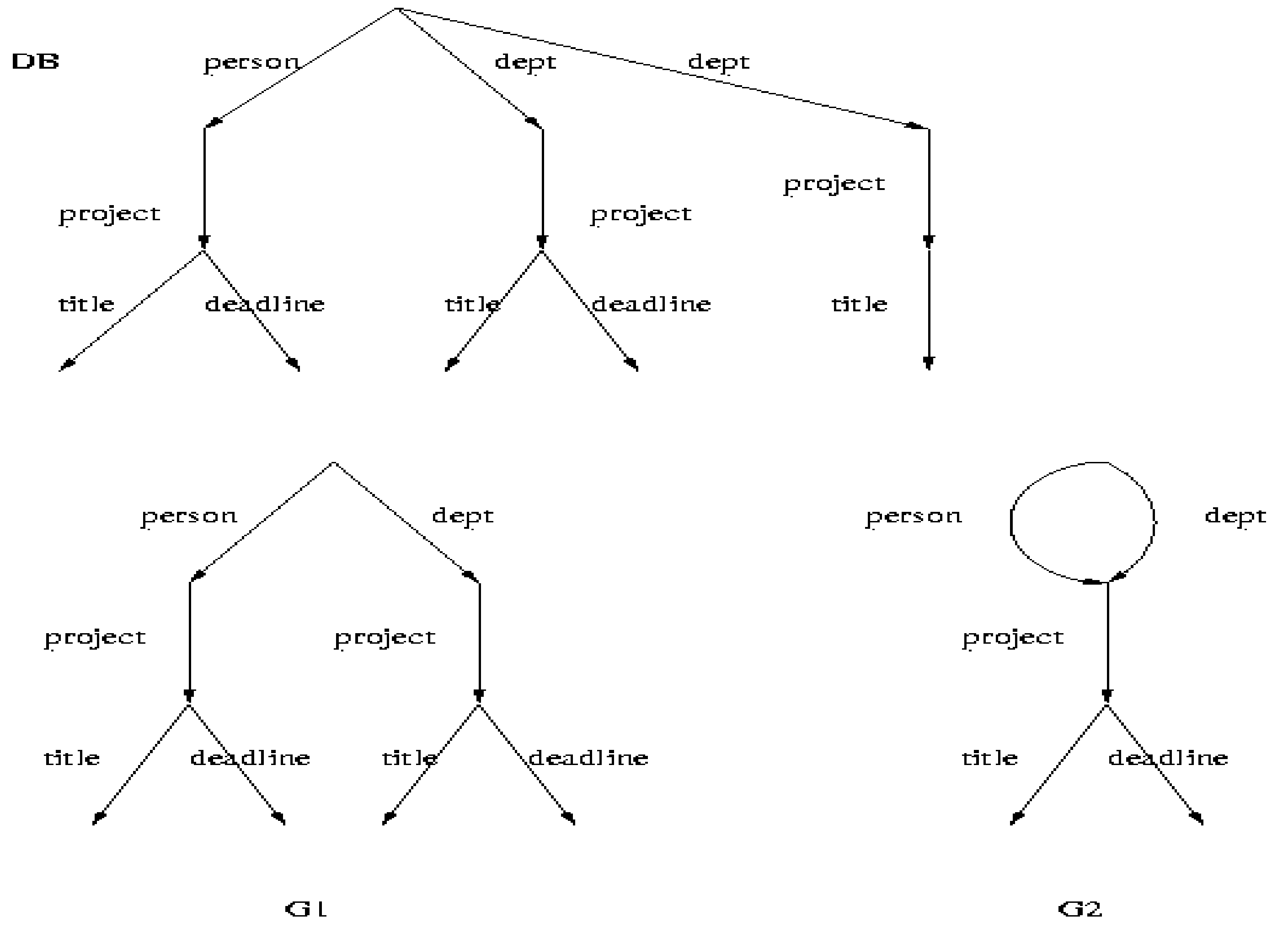
Example:



DB G

DataGuides

- Multiple DataGuides for the same data:



DataGuides

Definition

Let p, p’ be two path expressions and G a graph; we define

$$p \equiv_G p' \text{ if } p(G) = p'(G)$$

i.e., p and p' are indistinguishable on G.

Definition

G is a *strong* dataguide for a database DB if $\equiv_G$ is the same as $\equiv_{DB}$

Example:

- G1 is a strong dataguide
- G2 is not strong

person.project $!\equiv_{DB}$ dept.project

person.project $!\equiv_{G1}$ dept.project

person.project $\equiv_{G2}$ dept.project

DataGuides

- Constructing the strong DataGuide G:

  Nodes(G)={{root}}

  Edges(G)=∅

  while changes do

    choose s in Nodes(G), a in Labels

    add s’={y|x in s, (x -a->y) in Edges(DB)} to Nodes(G)

    add (x -a->y) to Edges(G)

- Use hash table for Nodes(G)
- This is precisely the powerset automaton construction.

Monet XML approach

- Early attempt to store and query XML data in MonetDB
- By Albrecht Schmidt
- Not related to Pathfinder & MonetDB/XQuery

Monet XML approach

Definition 1. *An* XML document *is a rooted tree* $d = (V, E, r, label_E, label_A, rank)$ *with nodes* V *and edges* $E \subseteq V \times V$ *and a distinguished node* $r \in V$, *the root node. The function* $label_E : V \to \textbf{string}$ *assigns labels to nodes, i.e., elements;* $label_A : V \to \textbf{string} \to \textbf{string}$ *assigns pairs of strings, attributes and their values, to nodes. Character Data (CDATA) are modeled as a special 'string' attribute of cdata nodes,* $rank : V \to \textbf{int}$ *establishes a ranking to allow for an order among nodes with the same parent node. For elements without any attributes* $label_A$ *maps to the empty set.*

Definition 2. *A pair* $(o, \cdot) \in \textbf{oid} \times (\textbf{oid} \cup \textbf{int} \cup \textbf{string})$ *is called an* association.

Definition 3. *For a node* o *in the syntax tree, we denote the sequence of labels along the path (vertex* and *edge labels) from the root to* o *with* path(o).

Monet XML approach

DEFINITION 4. *Given an XML document d, the* Monet transform *is a quadruple* $M_t(d) = (r, \mathbf{R}, \mathbf{A}, \mathbf{T})$ *where*

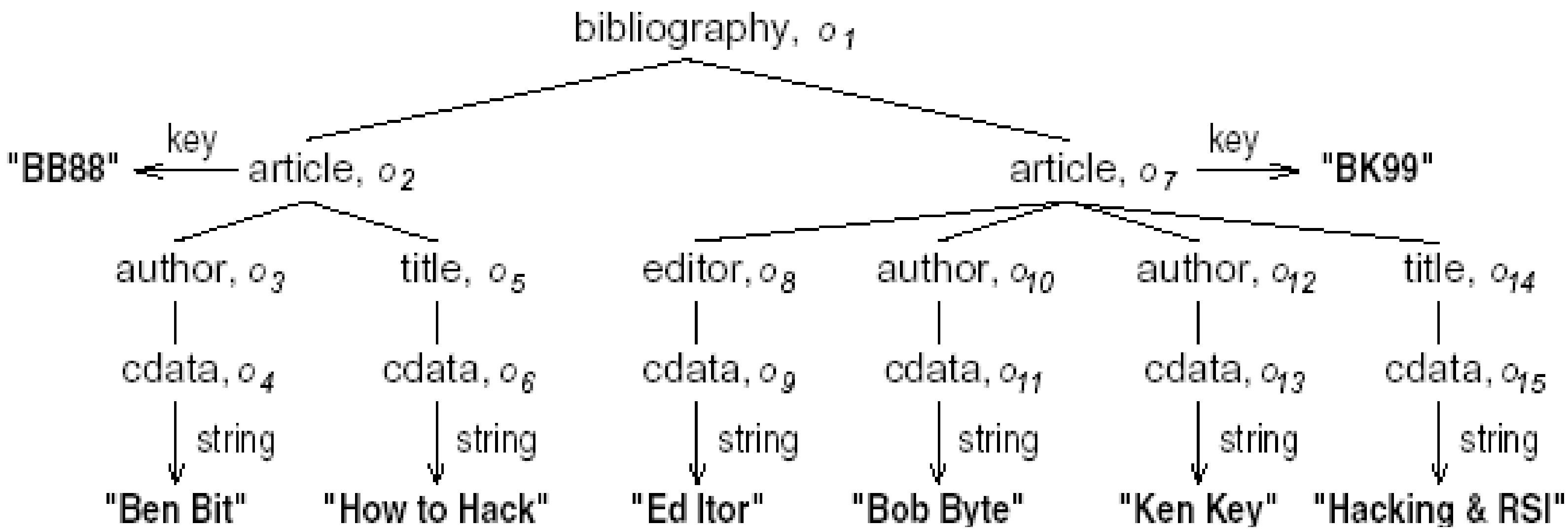
R *is the set of binary relations that contain* all *associations between nodes;*

A *is the set of binary relations that contain* all *associations between nodes and their attribute values, including character data;*

T *is the set of binary relations that contain* all *pairs of nodes and their rank;*

r remains the root of the document.

Monet XML approach



$$bibliography \xrightarrow{E} article = \{\langle o_1, o_2\rangle, \langle o_1, o_7\rangle\},$$

$$bibliography \xrightarrow{E} article \xrightarrow{E} author = \{\langle o_2, o_3\rangle, \langle o_7, o_{10}\rangle, \langle o_7, o_{12}\rangle\},$$

$$bibliography \xrightarrow{E} article \xrightarrow{E} author \xrightarrow{E} cdata = \{\langle o_3, o_4\rangle, \langle o_{10}, o_{11}\rangle, \langle o_{12}, o_{13}\rangle\},$$

$$bibliography \xrightarrow{E} article \xrightarrow{E} author \xrightarrow{E} cdata \xrightarrow{A} string = \{\langle o_4, \text{“Ben Bit”}\rangle, \langle o_{11}, \text{“Bob Byte”}\rangle, \langle o_{13}, \text{“Ken Key”}\rangle\},$$

$$bibliography \xrightarrow{E} article \xrightarrow{E} title = \{\langle o_2, o_5\rangle, \langle o_7, o_{14}\rangle\},$$

$$bibliography \xrightarrow{E} article \xrightarrow{E} title \xrightarrow{E} cdata = \{\langle o_5, o_6\rangle, \langle o_{14}, o_{15}\rangle\},$$

$$bibliography \xrightarrow{E} article \xrightarrow{E} title \xrightarrow{E} cdata \xrightarrow{A} string = \{\langle o_6, \text{“How to Hack”}\rangle, \langle o_{15}, \text{“Hacking \& RSI”}\rangle\},$$

$$bibliography \xrightarrow{E} article \xrightarrow{E} editor = \{\langle o_7, o_8\rangle\},$$

$$bibliography \xrightarrow{E} article \xrightarrow{E} editor \xrightarrow{E} cdata = \{\langle o_8, o_9\rangle\},$$

$$bibliography \xrightarrow{E} article \xrightarrow{E} editor \xrightarrow{E} cdata \xrightarrow{A} string = \{\langle o_9, \text{“Ed Itor”}\rangle\},$$

$$bibliography \xrightarrow{E} article \xrightarrow{A} key = \{\langle o_2, \text{“BB88”}\rangle, \langle o_7, \text{“BK99”}\rangle\}\}$$

Monet XML approach

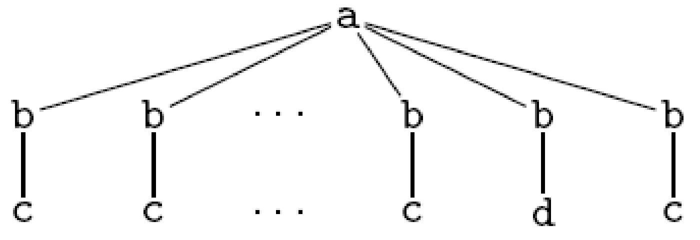
- Early attempt to store and query XML data in MonetDB
- By Albrecht Schmidt
- Not related to Pathfinder & MonetDB/XQuery
- No XQuery compiler
  - XMark queries are hand-crafted and -optimized in MIL
- Child, Descendant, Parent & Ancestor steps become regular expressions on the relation names (i.e., catalog)
- Open: preceeding & following steps?

Topics

- **Other approaches & techniques (*selection, far from complete!*)**
  - Document storage / tree encoding:
    - ORDPATH
    - DataGuides
  - XPath processing:
    - Tree patterns, holistic twig joins

Twig Join Algorithms

- So far: interpreted XPath expressions in an **imperative** manner
  - Evaluated XPath expressions **step-by-step**, as stated in the query
  - Given $/\alpha_1::\nu_1/\alpha_2::\nu_2/\ldots/\alpha_n::\nu_n$,
  - we first evaluated /, then XPath step $\alpha_1::\nu_1$, then step $\alpha_2::\nu_2$, ...
- This may not always be the best choice:
  - **Intermediate results** can get very large, even if the final result is small:



▷ /a/b/d produces many intermediate b nodes, but only a single result node.

- Database context => think in a **declarative** manner
  - DBMS optimizer / engine can evaluate query in “best” order

Tree Patterns

- In fact, XPath is a **declarative language**.
  - `/descendant::timeline/child::event`

    "Find all nodes v_1, v_2, and v_3, such that

    v_1 is a document root,

    v_2 is a descendant element of v_1 and is named `timeline`*, and*

    v_3 is a child element of v_2 and named `event`*.*

    All nodes of type v_3 form the query result.
- Observe the combination of

  (a) predicates **on single nodes**, and

  (b) structural conditions **between these nodes**.

Tree Patterns

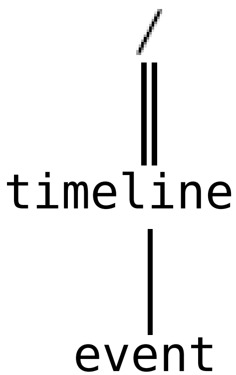
- Structural conditions: Intuitively expressed as **tree patterns:**

  - Nodes labeled with **node predicates**
  - Structural conditions:

    Double line: ancestor/descendant relationships

    Single line: parent/child relationships

- Arbitrary predicates are allowed, but typical are predicate on tag names:



  - Nodes labeled with requested tag name
  - Document root: label /

    If not /-node specified:

    search for pattern **anywhere in the document**

Tree Patterns

- Given such a tree pattern, 'query evaluation' means

  "Find all bindings of nodes in the document to nodes in the tree pattern, such that all structural and node constraints are fulfilled."

  ▷ Compare this to the tuple relational calculus:

  $$\{t \mid \exists r, \exists s : R(r) \wedge S(s) \wedge r[a] = s[a] \wedge t[a] = r[a] \wedge t[b] = s[b]\}$$

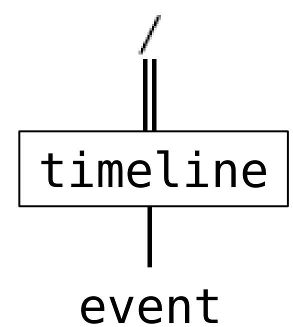
  We search for bindings for r and s that satisfy the given predicate.

- We have not, however, specified which of the pattern nodes to be the **query result**.

  ▷ Either return **tuples** of nodes, as binding to all the pattern nodes,

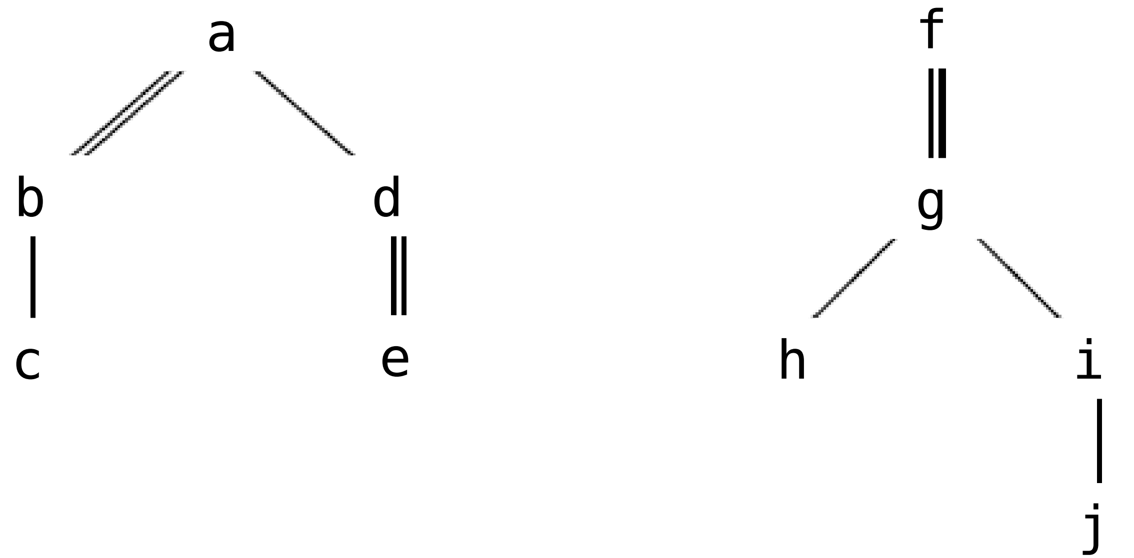
  ▷ or **mark** a specific node in the query as the result node.

  ▷ ✎ What is the XPath query for the tree pattern on the right?



Tree Patterns

- Not limited to **path patterns**
- May also be **twig patterns**
- Mapping between tree patterns and XPath is in general not trivial
- Examples:



PathStack Algorithm

- N. Bruno, N. Koudas, and D. Srivastava. "Holistic Twig Joins: Optimal XML Pattern Matching." In *Proceedings of the 21st Int'l ACM SIGMOD Conference on Management of Data.* Madison, Wisconsin, USA, 2002.
- Answer queries for **path patterns**.
- **Idea:**
  - ▷ Path patterns contain the **forward** axes `child` and `descendant` only.
  - ▷ To evaluate forward axes, it is sufficient to scan **forward in preorder only**.
  - ▷ Can we evaluate path queries in a **single document scan**?

PathStack Algorithm: Path Patterns

- During a sequential table read, maintain the path from the root to the current node with the help of a **stack**:

  For each node *n*

  ▷ Remove all nodes *v* from the stack that are not ancestors of *n* ($v.post < n.post$).

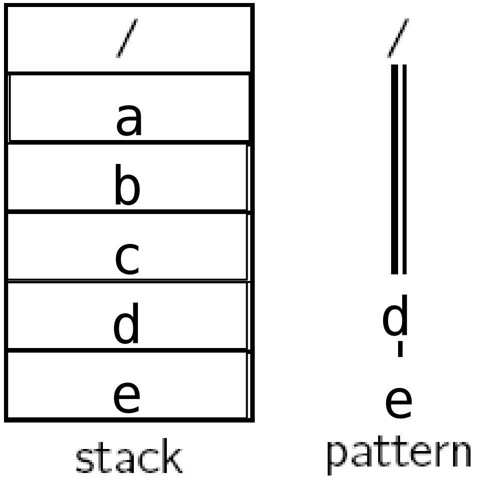
  ▷ Push *n* onto the stack.

  (This is similar to the stack we used to generate the *pre*/*post* encoding.)

- For any node check if we can **match** the stack against the query pattern.

  ▷ **Example:** Stack

  ▷ For `descendant` axes, we allow **gaps** for the match.



➠ We **can** find path patterns in a **single sequential read**.

PathStack Algorithm: Path Patterns

- The task is now to match the ancestor stack against the query pattern.
  - ▷ This requires **regular expression** matching.
  - ▷ Matching has to be triggered for each document node.
  - ▷ Regular expression matching is **expensive**.
- It is not sufficient to find **some** match, we need to find **all** query results.
  - ▷ There may be multiple matches on the same stack.
    (E.g., if the same tag name appears more than once on the stack.)

➠ Although we meet the **single scan** constraint, path evaluation is **tedious**.

- **Idea:**
  - ▷ While scanning, only put **interesting** nodes on the stack.
  - ▷ Add some more **structural information** to the stack.

PathStack Algorithm: Path Patterns

① **Test** the predicates **before** pushing nodes on the stack.

- ▷ Save work when evaluating the stack.

② Keep **separate** stacks **for each node in the query pattern**.

- ▷ We know which predicate each node belongs to afterwards.
- ▷ Each of the stacks contains the ancestor/descendant relationship of nodes satisfying the same predicate.

③ **Link** nodes in **different** stacks to represent their ancestor/descendant relationship.

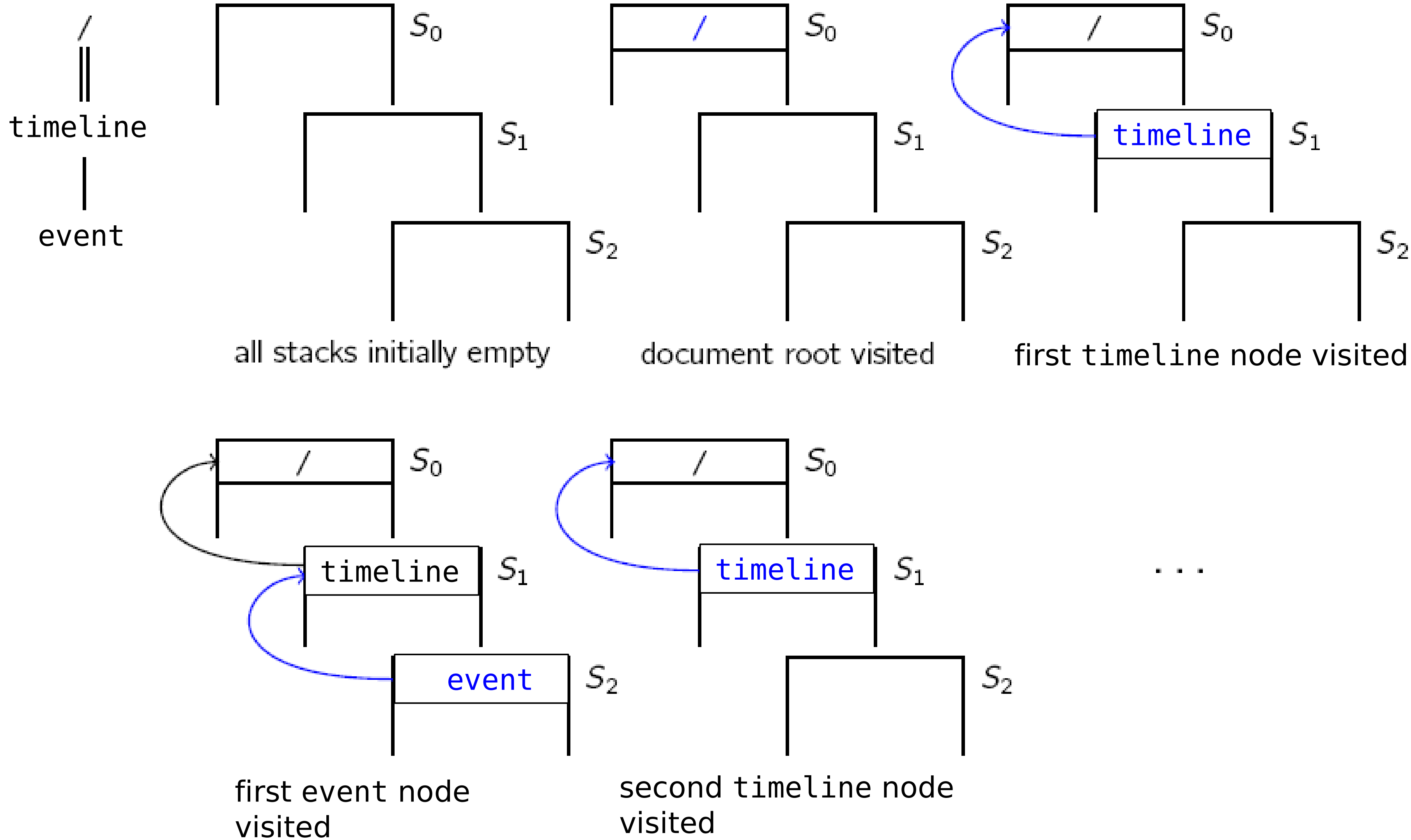
- ▷ Recover the information we lost in ②.

PathStack Algorithm: Path Patterns

- When a node is pushed onto the stack S_i, it is linked to the current top of S_{i-1}.
  - ▷ The **pointer** starting from node v always points to an **ancestor** of v.
- We insert a node into Stack S_i only if
  - ▷ the **parent stack** S_{i-1} is **not empty**, or
  - ▷ S_i is the stack of the **query root**, i.e. $i = 0$.
- Nodes within one stack are always in ancestor/descendant relationship.
  - ▷ From stack-bottom to top, all nodes are on a root-to-leaf path in the XML tree.
- For **descendant-only** patterns we have found an answer, as soon as there is a node in the **leaf stack**.
  - ▷ The child relationship has to be checked separately.
- The tree of stacks encodes **all** (partial) answers to the query pattern.
  - ▷ We will shortly see how to retrieve them.

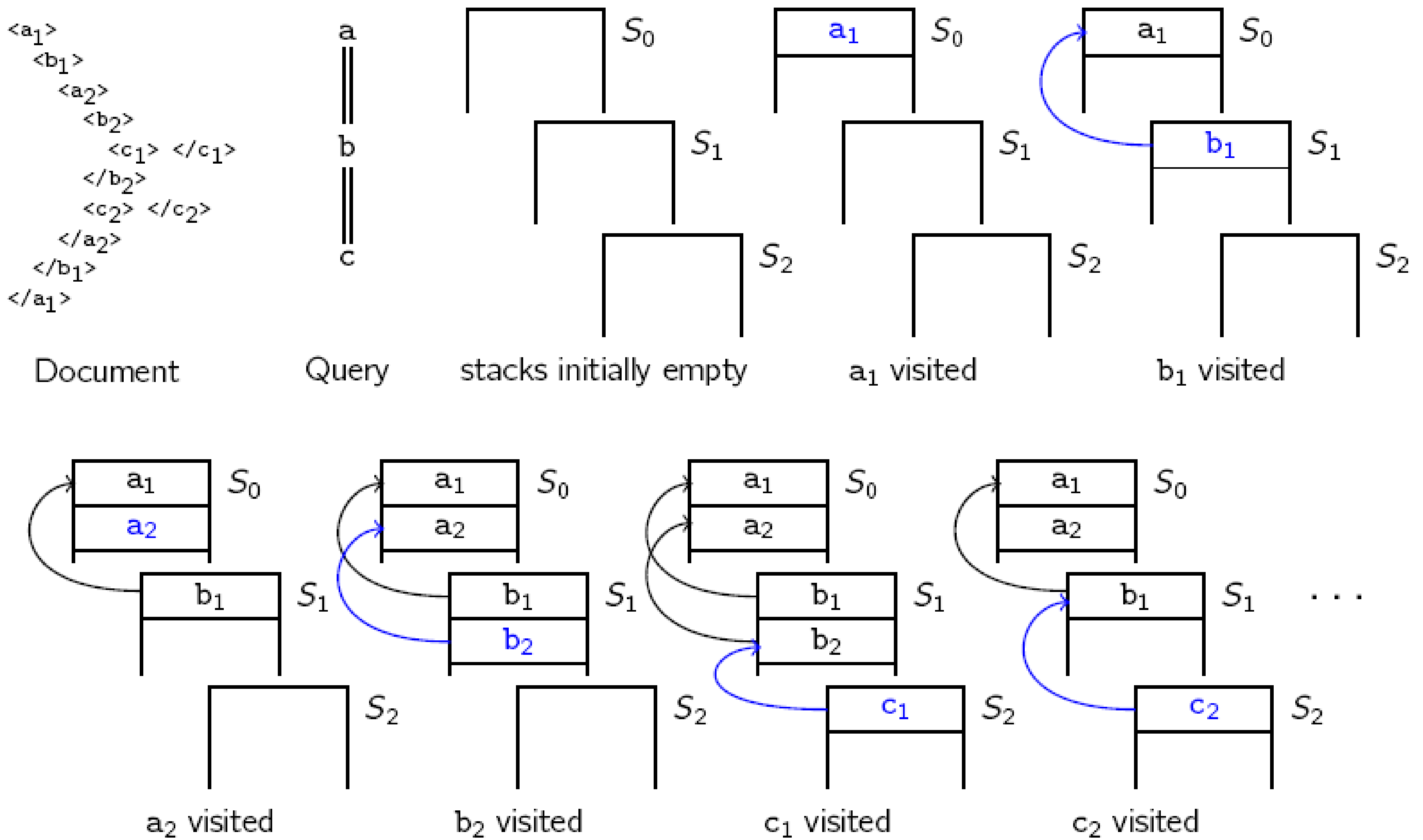
PathStack Algorithm: Path Patterns

Example:



PathStack Algorithm: Path Patterns

Example: Recursive XML



PathStack Algorithm: Path Patterns

- For each tuple t in the document relation, the PathStack algorithm performs three steps:

  ① Clean stacks.

  ▷ Remove all nodes in all stacks that **precede** the current node t. ($v \in t/\texttt{preceding} \Leftrightarrow v.pre < t.pre \land v.post < t.post$)

  ② Push t on the appropriate stack.

  ▷ Push if t matches a predicate in q.

  ▷ Only push if t matches the **query root**, or the **parent stack** is not empty.

  ③ If t matches the **query leaf**, output all solutions.

  ▷ We are then sure to find a path from the root to t that contains a match for each query predicate.

- If **overlapping predicates** are required, i.e. a node can satisfy more than one of the predicates, the algorithm needs to be rewritten slightly.

PathStack Algorithm: Path Patterns

Function `PathStack` (q : query pattern, doc : table (pre, $post$))

```
foreach t ∈ doc in pre-order do
    foreach n_i ∈ q do
        while ¬ empty(S_i) ∧ S_i.top().post < t.post do
            S_i.pop();    /* clean stacks */
    if t matches a predicate p_i in q then
        if i = 0 then
            S_0.push(t, nil);    /* deal with query root node */
        else if ¬ empty(S_{i−1}) then
            S_i.push(t, stack position of S_{i−1}.top());
        if q_i is a leaf in the query pattern and t has been pushed onto a stack then
            showSolutions(i, stack position of S_i.top());
            S_i.pop();
```

PathStack Algorithm: Path Patterns

Back-tracing the solutions

- We are now left with the output of the actual query solution.
- Without the request for a specific binding in the query pattern, we return **all bindings** to **all query nodes**.
- **Idea:**
  - ▷ From each node v in each stack S_i, we find its **ancestors**
    - – below v in stack S_i, and
    - – in stack S_{i-1}, if we follow the **parent pointer** of v.
  - ▷ We find all solutions by following all these ancestors until the **root stack**.

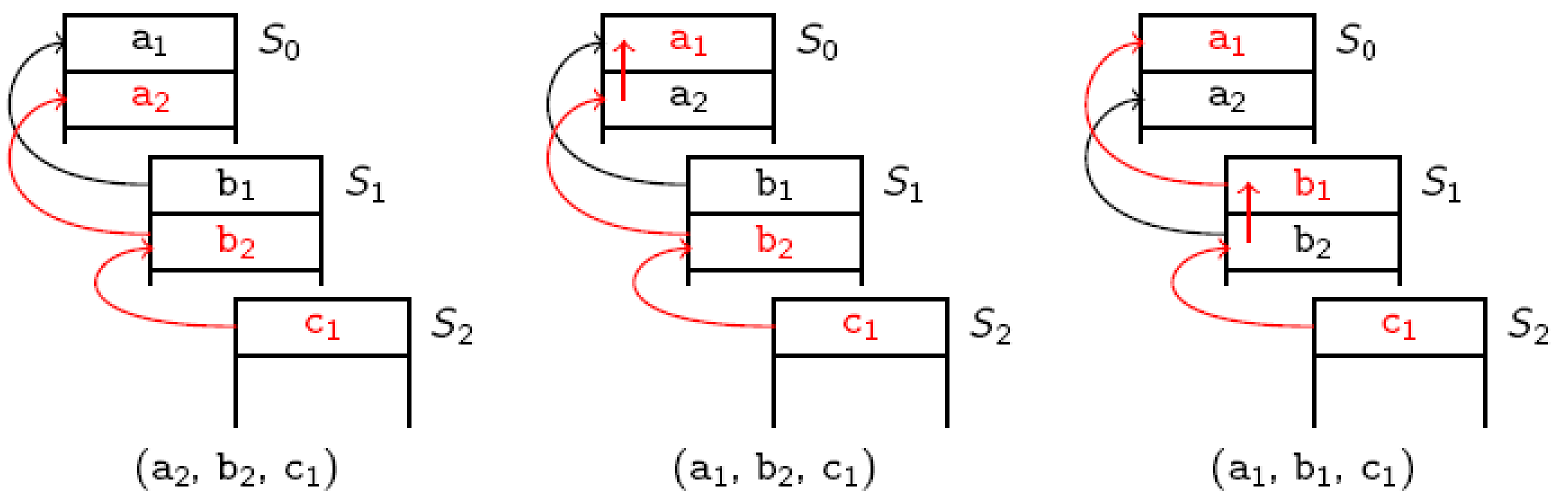
PathStack Algorithm: Path Patterns

Example: Recursive XML document

```
<a1>
  <b1>
    <a2>
      <b2>
        <c1> </c1>
      </b2>
      <c2> </c2>
    </a2>
  </b1>
</a1>
```

Document

Query



PathStack Algorithm: Path Patterns

Function `showSolutions`(stackno : int, slotno : int)

positions[stackno] $\leftarrow$ slotno;
if stackno $= 0$ **then**
    output $(S_0[positions[0]], \ldots, S_{n-1}[positions[n-1]])$;
else
    foreach $j < S_{\text{stackno}}[\text{slotno}].parent$ **do**
        `showSolutions`(stackno - 1, j);

- n is the number of nodes in the query pattern.
- *positions* is an array of length n that holds the current position within all stacks traversed so far.
- We assume that we can reach an entry within a stack by an **index**, starting from 0.
- If we reach the query root stack S_0, we output the node in each stack we traversed to reach the root stack.
- Otherwise we follow the parent pointer (the *parent* field is the **index** within the parent stack) and recurse for that parent and all its ancestors in the parent stack.

PathStack Algorithm: Path Patterns

- showSolutions() returns all query answers for **descendant-only** queries.
- To support the **child** axis, we additionally need test the *level* properties.
- ✎ How can we rewrite showSolutions() to support the child axis?

PathStack Algorithm: Path Patterns

- The showSolutions() algorithm with support for the child axis:

Function `showSolutions`(stackno : int, slotno : int)

```
positions[stackno] ← slotno;
if stackno = 0 then
    output (S_0[positions[0]], ..., S_{n-1}[positions[n-1]]);
else
    if stackno - 1 → stackno is a descendant axis then
        foreach j < S_stackno[slotno].parent do
            showSolutions(stackno - 1, j);
    else
        foreach j < S_stackno[slotno].parent do
            if S_{stackno-1}[j].level = S_stackno[slotno].level - 1 then
                showSolutions(stackno - 1, j);
```

PathStack Algorithm: Path Patterns

- showSolutions() returns nodes in **leaf-to-root order**.
  - ▷ If another order is desired, we need to **block** processing.
- No duplicate elimination is performed.
  - ▷ If we **remove** each leaf node from the stack, as soon as its results are returned, we can avoid duplicates with respect to **all bindings**.
  - ▷ If only some bindings are requested, explicit **duplicate elimination** must be performed.
- PathStack does evaluate any **path pattern** in a single sequential read.
  - ▷ We touch at most |document| nodes.
  - ▷ Sequential access is (again) cache efficient.

PathStack Algorithm: Twig Patterns

- So far we only considered **path patterns**
- Can we extend our ideas for efficient **twig pattern** evaluation?

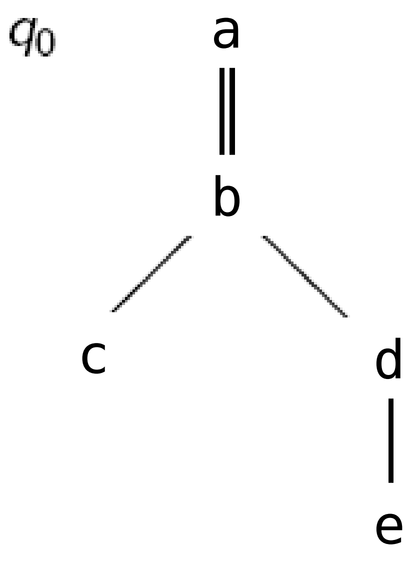
Idea:

- Decompose **twig patterns** into multiple **path patterns**.
- All path patterns start from the same **root**.
- Use PathStack for each of them and **merge** their results.

PathStack Algorithm: Twig Patterns

- **Example:** Decompose twig pattern into path patterns

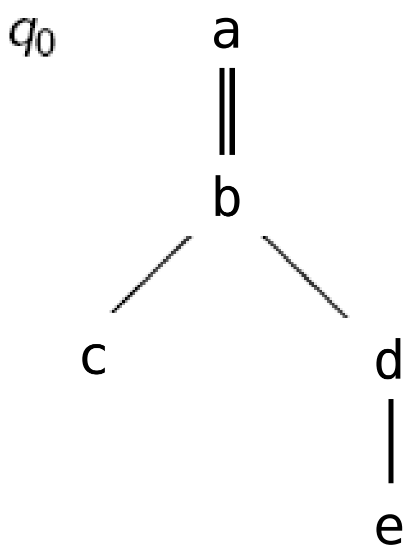
Original twig query q_0:



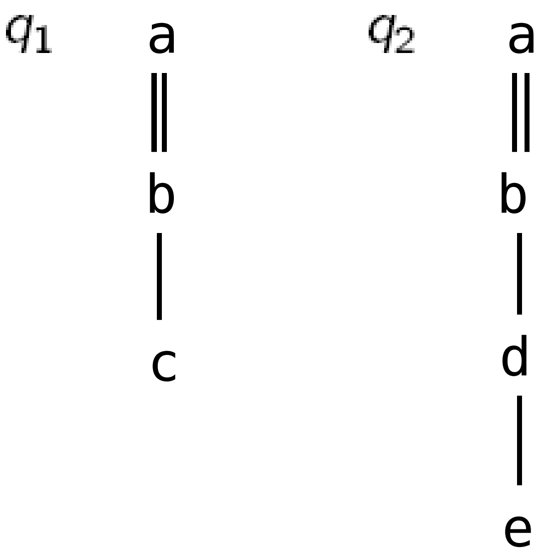
PathStack Algorithm: Twig Patterns

- **Example:** Decompose twig pattern into path patterns

Original twig query q_0:



Split into path patterns q_1 and q_2:

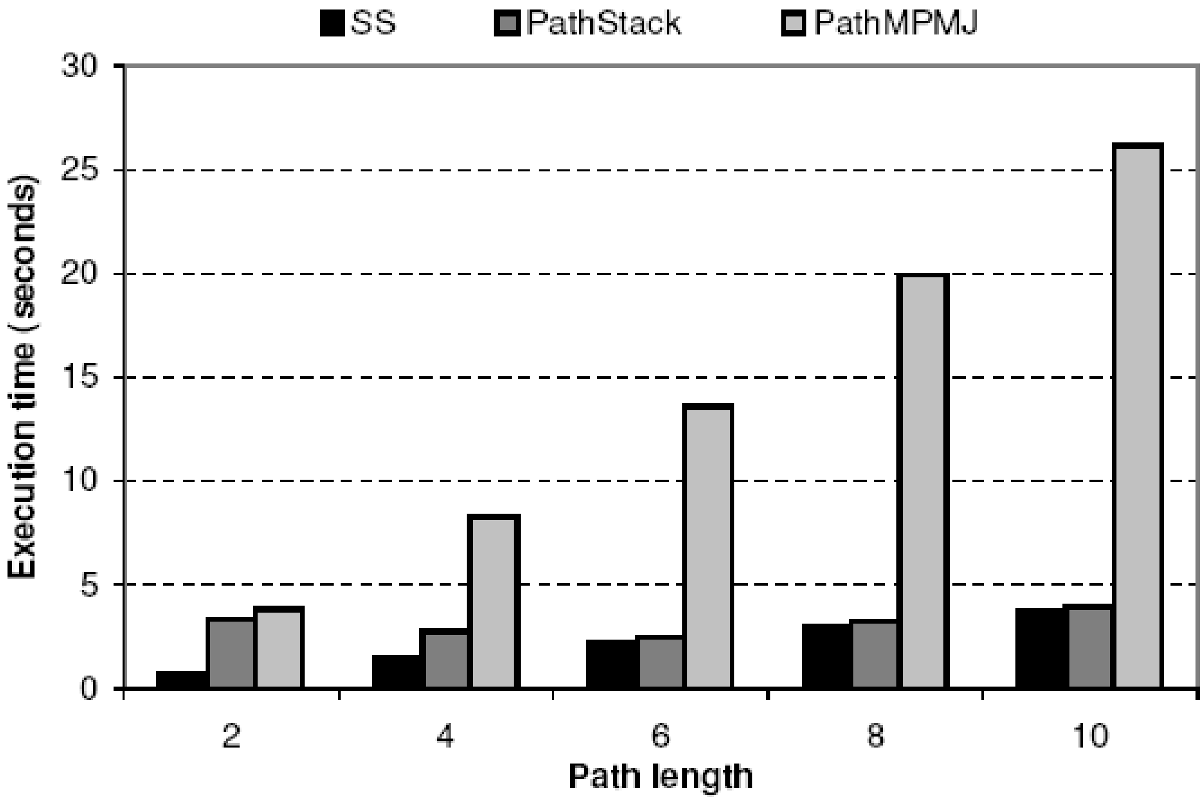


PathStack Algorithm: Twig Patterns

- We're now back at our original problem:
  - ▷ To evaluate twig patterns, we first produce **intermediate results**.
  - ▷ These intermediate results may get **huge**, even if the final result is **small**.
- Can we **avoid** some of the intermediate results that won't contribute anyway?
- **Idea:**
  - ▷ Before pushing a node onto a stack, **peek** at each descendant tuple stream.
  - ▷ Only push a node, if we can find nodes in the stream heads that allow the creation of **at least** one twig solution.
- This way the **TwigStack** algorithm **skips** irrelevant intermediate results.
  - ▷ The stream processing model allows this "peeking forward".
  - ▷ For the sequential document read, we need to **materialize** intermediate results.

PathStack Algorithm: Twig Patterns

PathStack performance



- The graphic shows the performance of PathStack, compared to a simple evaluation strategy, similar to a nested loop ("PathMPMJ").
- The time needed for a sequential read of the data is labeled "SS".

Summary (1/5)

- **XML**
  - Document markup
  - Data exchange
  - Semi-structured
  - Tree model
  - DTDs
  - XML Schema
- **XPath**
  - Navigation, location steps, axes, node tests, predicates, functions
- **XQuery**
  - Sequences & Iterations (FLWoR expressions)

Summary (2/5)

- **XML Data Management**
  - XML file processors
  - XML databases
  - XML integration platforms
  - RDBMS with XML functionality, SQL/XML
  - Relational XML storage: schema-based vs. schema-oblivious

Summary (3/5)

- **Purely Relational XML/XQuery processing: MonetDB/XQuery**
  - Document encoding: XPath Accelerator (pre/post plane)
  - XPath navigation: Staircase Join
  - XQuery to Relational Algebra translation
    - Item- & Sequence-representation
    - Iterations: Loop-lifting
    - Loop-lifted staircase join
    - Peephole Optimization
    - Order-awareness, sort avoidance
  - XML/XQuery Update Support

Summary (4/5)

- **Other approaches & techniques**
  - Document storage/encoding:
    - ORDPATH
    - DataGuides
  - XPath processing:
    - Tree patterns, holistic twig joins

Summary (5/5)

- **Literature**
  - Slides
  - Literature references on slides
  - Literature references on website:

    http://www.cwi.nl/~manegold/teaching/adt/html/xquery.html

- **Tentamen / Exam:**
  - *Tuesday December 21 2010*
  - *09:00 – 11:00*
  - *Zaal / Room: A1.14*

Projects: Join the MonetDB Team!

- **Own ideas, suggestions, initiative welcome!**
- **Master Student Projects (6 Months)**
  - Various projects, each consisting of both research & implementation
  - See monetdb.cwi.nl/Development/Research/Projects/ for a sample list
  - Feel free to come with your own idea(s)!
- **Implementation Projects**
  - Both short-term & long-term
  - E.g. open feature requests: sf.net/tracker/?group_id=56967
  - Become owner/maintainer of some (new) part of MonetDB
  - We are (*desperately*) looking for <u>Windows</u> SW-development & system *experts*!

We Offer...

- **24x7x365 support & advice**
- **Membership in a kind & friendly Family-Team of Experts**
- **Chance to participate in & contribute to a large & successful open-source research project**
- **Lots of experiences, exiting research & fun**
- **Desk & workstation at CWI**
  - Fridge, micro-wave, free coffee, free soup, free cake (occasionally)
  - Master Students only (possibly part-time)
  - Limited availability => FCFS!
- **Some pocket money (***stage vergoeding***)**
  - Master Students only
  - Limited availability => FCFS!
- ...