Recall of XPath and XQuery

Tree patterns

For expressions in XPath 2.0

For $i$ in (10, 20), $j$ in (1, 2) return ($i + j$)

Getting all books of each author:

```
<bib><book><title>TCP/IP Illustrated</title>
  <author>Stevens</author></book>
<book><title>Unix Programming</title>
  <author>Stevens</author></book>
<book><title>Data on the Web</title>
  <author>Abiteboul</author><author>Buneman</author><author>Suciu</author></book>
</bib>
```

Other expressions in XPath 2.0

If-then-else:

```
if ($widget1/unit-cost < $widget2/unit-cost)
  then $widget1
else $widget2
```

Quantified expressions:

```
every $part in /parts/part satisfies $part/@discounted
some $emp in /emps/employees satisfies ($emp/bonus > 0.25 * $emp/salary)
```
From XPath to tree patterns

Tree patterns: convenient query abstraction

```xml
//person[mail]/name
```

or

```xml
for $p in //person[mail]
return name
```

for $p in */person
return $p, $p/name, $p/mail

---

XQuery with a FLWOR expression

"Find the description and average price of each red part that has at least 10 orders"

```xml
for $p in doc("parts.xml")//part[color = "Red"]
let $o := doc("orders.xml")//order[partno = $p/partno]
where count($o) >= 10
order by count($o)
return
<important_red_part>
  ($p/description)
  <avg_price> (avg($o/price)) </avg_price>
</important_red_part>
```

---

Difference between for and let

For:
- for $x in /company/employee
  binds $x successively to each employee

Let:
- let $x := /company/employee
  binds $x only once, to the list of all employees

---

Joining several documents

```xml
for $p in doc("www.irs.gov/taxpayers.xml")//person
for $n in doc("neighbors.xml")//neighbor[ssn = $p/ssn]
return
<person>
  <ssn> { $p/ssn } </ssn>
  { $n/name }
  <income> { $p/income } </income>
</person>
```

---

XQuery with aggregation and order

```xml
for $d in doc("depts.xml")//deptno
let $s := doc("emps.xml")//employee[deptno = $d]
where count($s) >= 10
order by avg($s/salary) descending
return <big-dept> ($d,
  <headcount>(count($s))<headcount>,
  <avgsal>(avg($s/salary))<avgsal>)
</big-dept>
```

Returns the departments having more than 10 employees, ordered by the salary.

---

From XQuery to tree patterns (1)

Previous XQuery query:

```xml
for $d in doc("depts.xml")//deptno
let $s := doc("emps.xml")//employee[deptno = $d]
where count($s) >= 10
order by avg($s/salary) descending
return <big-dept> ($d,
  <headcount>(count($s))<headcount>,
  <avgsal>(avg($s/salary))<avgsal>)
</big-dept>
```
From XQuery to tree patterns (2)

Remove arithmetic expressions and order:

for $d$ in doc("depts.xml")/deptno
let $e$ := doc("emps.xml")/employee[deptno = $d]
where count($e) > 10
order by avg($e/salary) descending
return <big-dept> { $d, 
    <headcount>{count($e)}</headcount>,
    <avgslt>{avg($e/salary)}</avgslt> }
</big-dept>

From XQuery to tree patterns (3)

Remove arithmetic expressions and order:

for $d$ in doc("depts.xml")/deptno
let $e$ := doc("emps.xml")/employee[deptno = $d]
return <big-dept> { $d, 
    <headcount>$e</headcount>,
    <avgslt>$e/salary</avgslt> }
</big-dept>

Value join

Not really!

Techniques for efficient XPath and XQuery evaluation

XML query evaluation techniques (1)

We will focus mostly on XPath and tree pattern evaluation.

Three broad classes of techniques:

1. In-memory
   - Transform an XML document into a tree made of memory objects
   - Compile query into operations on the in-memory tree
   - Efficient, relatively easy to implement
   - Limited for large-size documents; inefficient for selective queries

2. Streaming
   - Traverse the XML document once, from the beginning to the end
   - Process the elements found in the document for the needs of the query
   - Many streaming processors from research (XFilter, YFilter...)
   - Results sometimes (not always) fast (before traversing all the doc)
   - Possibility to evaluate simultaneously several queries
   - Inefficient for selective queries

3. External
   - Transform the XML document into a data structure
   - Compile query into operations on the external structure
   - Evaluate the query on the transformed document
   - Efficient for selective queries

XML query evaluation techniques (2)
XML query evaluation techniques (3)

Three broad classes of techniques:

3. Based on a persistent store
   - Store the document in one or several disk-resident structures
   - Build indices over these structures
   - Evaluate queries by reading data from these structures and possibly applying further processing
   - ex: eXist, MonetDB (among the fastest), commercial RDBMSs
   - Potentially most scalable due to indices and pre-computed data structures
   - +/- algebraic optimizations can apply
   - Heavy: loading data in the database takes time. Inefficient if data is queried only once.

Part 1

Streaming processing of tree pattern queries [CBZ06 among others]

Streaming processing of tree pattern queries

XML document:
```xml
<r>
  <a1>
    <b1>
      <c1/>
      <c2/>
    </b1>
    <b2>
      <c3/>
      <c4/>
    </b2>
  </a1>
  <a2>
    <c5/>
    <b3/>
  </a2>
</r>
```

Query: //a/b/c

Traverse the document sequentially and issue events:
- Start element x / end element x / text s

When pushed, matches are open

On begin element x:
- If there is a stack for x
  - Then if the element appears in the right context
    - Push it on the stack; connect it to the parent match

On end element x:
- If there is a stack for x
  - Then if x is on top of the stack
    - If x lacks some required children
      - Pop x, possibly some descendents
Streaming processing of tree pattern queries

On begin element x:
If there is a stack for x
Then if the element appears in the right context
then push it on the stack; connect it to the parent match

On end element x:
If there is a stack for x
Then if x is on top of the stack
Then if x lacks some required children
then pop x, possibly some desc

After end element, a match is closed
Streaming processing of tree pattern queries

On begin element x:
- If there is a stack for x
  - Then if the element appears in the right context
    - push it on the stack;
    - connect the parent match

On end element x:
- If there is a stack for x
  - Then if x is on top of the stack
    - then pop x, possibly some desc

Then if x is on top of the stack
- the element appears
- pop x, possibly some desc.

M2R IAC, 2012

Ioana Manolescu

Streaming processing of tree pattern queries

On begin element x:
- If there is a stack for x
  - Then if the element appears in the right context
    - push it on the stack;
    - connect it to the parent match

On end element x:
- If there is a stack for x
  - Then if x is on top of the stack
    - then pop x, possibly some desc

Then if x is on top of the stack
- the element appears
- pop x, possibly some desc.

M2R IAC, 2012

Ioana Manolescu
Variations: capturing element content

Sa

\[ \text{Query: } /a/b \]

\[ \text{Query: } /a/b \]

Other variations

Testing value predicates on nodes: do not push unless predicates are satisfied

// edges: no change

- labeled query nodes: use S*, on which any node of the right context can be pushed

Optional query nodes: do not prune parent match if a child match is lacking

Time and space complexity

Time: size of the document |D| x size of the query |q|

Space:
- Number of stacks = number of query nodes
- Maximal stack height = maximal depth of matches which are ancestors of one another < document depth
- Total bounded by |q| x |D|: polynomial size encoding of exponentially many results (|D| |q|)
- If string results are returned, string buffers may be large!
- In some cases the string buffers can be flushed as we go

Interesting for small-to-moderate documents, not very selective patterns
Not interesting for large documents with highly selective patterns

Plan

Naive XPath evaluation algorithm
- Syntax-driven
- Exponential time complexity
- Present in many initial implementations

Advanced evaluation algorithms [GKP03,GKP05]
- Polynomial time complexity
- Similar results in [S03], [GKPS05]

Part 2

In-memory evaluation of tree pattern queries
Naive XPath evaluation algorithm

In 2012, Ioana Manolescu from M2R IAC introduced a naive XPath evaluation algorithm inspired directly by the W3C specification. For simplicity, consider only linear queries of the form

\[ \text{step1 step2 step3 ... stepn} \]

where `step1` is the head and `step2 step3 ... stepn` is the tail.

**Procedure**

```
 process-location-step(n0, Q)
 begin
 node set S := apply Q.head to node n0;
 if (Q.tail is not empty) then
 for each node n in S do process-location-step(n, Q.tail);
 end
```

**Example 1:**

XML document: `<a><b/><b/></a>`
Query: `//a/b/parent::a/b/parent::a/b/parent::a/b/...//parent::a/b`
Exponentially many visited nodes (thus, exponential time) in the query size!

Performance of naive XPath evaluation algorithm

**Procedure**

```
 process-location-step(n0, Q)
 begin
 node set S := apply Q.head to node n0;
 if (Q.tail is not empty) then
 for each node n in S do process-location-step(n, Q.tail);
 end
```

**Example 2:**

XML document `Doc(i)`: `<a> <b>c</b> <b>c</b> ... <b>c</b> </a>`
Query `Q(1): //* [parent::a/child::*='c']`
Query `Q(2): //* [parent::a/child::* [parent::a/child::*='c'] = 'c']`
Query `Q(3): //* [parent::a/child::* [parent::a/child::* ['c'] = 'c'] = 'c']`
Exponentially many visited nodes (thus, exponential time) in the query size!

Efficient XPath evaluation algorithms [GKP05]

Basic axes: `firstchild` and `nextsibling` (also seen as functions or relations)

\[ n.firstchild = m \iff \text{firstchild}(n) = m \]
\[ n.nextsibling = m \iff \text{nextsibling}(n) = m \]

The other axes can be expressed using `firstchild`, `nextsibling` and `self`

Examples:

- child := `firstchild.nextsibling`
- parent := `nextsibling`\^1 \cdot `firstchild`\^1
- descendant := `firstchild.nextsibling`\^1
- ancestor := `firstchild`\^1 \cdot `nextsibling`\^1
- descendant-or-self := `descendant` \cup `self`

Efficient algorithm for evaluating 1 axis [GKP05]

**Input:** set of nodes S, axis \( \chi \)

**Output:** \( \chi(S) \), the nodes obtained by applying the axis \( \chi \) to the nodes in S

**Method** `eval\chi(S)`

1. `eval\chi(S) := S`
2. `eval\chi.nextsibling(S) := \{ firstChild(n) \mid n \in S \}`
3. `eval\chi.nextsibling(S) := \{ nextsibling(n) \mid n \in S \}`
4. For the other axes, use the expression defining \( \chi \) in terms of (self, firstchild, nextsibling) and a set of combination rules:
   - \[ \text{eval1.u2(S)} := \text{eval1(eval2(S))} \]
   - \[ \text{eval1.u2(S)} := \text{eval1(S) U eval2(S)} \]
Efficient algorithm for evaluating 1 axis [GKP05]

**Input:** set of nodes $S$, axis $\chi$

**Output:** $\chi(S)$, the nodes obtained by applying the axis $\chi$ to the nodes in $S$

**Method eval(\(S\))**

1. $\text{eval}_0(S) := S$
2. $\text{eval}_1(S) := \{\text{firstChild}(n) \mid n \in S\}$
3. $\text{eval}_2(S) := \{\text{nextSibling}(n) \mid n \in S\}$
4. For the other axes, use the expression defining $\chi$ in terms of $\{\text{self}, \text{firstchild}, \text{nextSibling}\}$ and a set of combination rules:
   - $\text{eval}_i(R_1 \cup R_2 \cup \ldots \cup R_n)$: inflationary algorithm based on $R_i(S)$, $1 \leq i \leq n$

**Complexity:** $O(D)$ for any axis (assuming a bit array implementation)

---

Efficiently evaluating XPath expressions

Relies on the notion of context of evaluation of any expression $c = <x, k, n>$

$x$ is the context node, $k$ is the context position and $n$ is the context size

**Domain of contexts** $C = \text{dom } x \{<k, n> \mid 1 \leq k \leq n \leq |\text{dom}|\}$

(dom is the domain of all document nodes)

**Bottom-up XPath semantics: a function** $E : \text{Expression} \rightarrow 2^C \times (\text{node sets} \cup \text{numbers} \cup \text{strings} \cup \text{booleans})$

"Given an expression $q$, which is the set of (context, value) pairs, such that the expression evaluated in that context yields that value"

Bottom-up semantics and the context-value table

Bottom-up XPath semantics: a function

$E : \text{Expression} \rightarrow 2^C \times (\text{node sets} \cup \text{numbers} \cup \text{strings} \cup \text{booleans})$

"Given an expression $q$, which is the set of (context, value) pairs, such that the expression evaluated in that context yields that value"

The results of the bottom-up semantics are stored in a context-value table, recursively defined for more and more complex expressions.

Filling in the context value table for a query gradually (in a bottom-up fashion) = evaluating the query

---

Context-value table example (1)

Document $<a><b1/><b2/><b3/><b4/></a>$

Query descendant::b/following-sibling::*[position() != last]

Q: $E_1/E_2$

$E_1$: descendant::b

$E_2$: $E_3[E_4]$

$E_3$: following-sibling::*

$E_4$: $E_5 = E_6$

$E_5$: position()

$E_6$: last

Context-value table for $E_1$

<table>
<thead>
<tr>
<th>$r$</th>
<th>$b_1$, $b_2$, $b_3$, $b_4$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$a$</td>
<td>$b_1$, $b_2$, $b_3$, $b_4$</td>
</tr>
<tr>
<td>$b_1$</td>
<td>$b_2$, $b_3$, $b_4$</td>
</tr>
<tr>
<td>$b_2$</td>
<td>$b_3$, $b_4$</td>
</tr>
<tr>
<td>$b_3$</td>
<td>$b_4$</td>
</tr>
<tr>
<td>$b_4$</td>
<td>$b_4$</td>
</tr>
</tbody>
</table>

K and n omitted for readability. All legal combinations are OK here.

Context-value table for $E_3$

<table>
<thead>
<tr>
<th>$r$</th>
<th>$b_1$, $b_2$, $b_3$, $b_4$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$a$</td>
<td>$b_1$, $b_2$, $b_3$, $b_4$</td>
</tr>
<tr>
<td>$b_1$</td>
<td>$b_2$, $b_3$, $b_4$</td>
</tr>
<tr>
<td>$b_2$</td>
<td>$b_3$, $b_4$</td>
</tr>
<tr>
<td>$b_3$</td>
<td>$b_4$</td>
</tr>
<tr>
<td>$b_4$</td>
<td>$b_4$</td>
</tr>
</tbody>
</table>

Context-value table example (2)

Document $<a><b1/><b2/>><b3/>><b4/></a>$

Query descendant::b|following-sibling::*[position() != last]

Q: $E_1/E_2$

$E_1$: descendant::b

$E_2$: $E_3[E_4]$

$E_3$: following-sibling::*

$E_4$: $E_5 = E_6$

$E_5$: position()

$E_6$: last

Context-value table for $E_3$

<table>
<thead>
<tr>
<th>$r$</th>
<th>$b_1$, $b_2$, $b_3$, $b_4$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$a$</td>
<td>$b_1$, $b_2$, $b_3$, $b_4$</td>
</tr>
<tr>
<td>$b_1$</td>
<td>$b_2$, $b_3$, $b_4$</td>
</tr>
<tr>
<td>$b_2$</td>
<td>$b_3$, $b_4$</td>
</tr>
<tr>
<td>$b_3$</td>
<td>$b_4$</td>
</tr>
<tr>
<td>$b_4$</td>
<td>$b_4$</td>
</tr>
</tbody>
</table>

Context-value table example (3)

Document $<a><b1/>><b2/>><b3/>><b4/></a>$

Query descendant::b|following-sibling::*[position() != last]

Q: $E_1/E_2$

$E_1$: descendant::b

$E_2$: $E_3[E_4]$

$E_3$: following-sibling::*

$E_4$: $E_5 = E_6$

$E_5$: position()

$E_6$: last

Context-value table for $E_3$

<table>
<thead>
<tr>
<th>$r$</th>
<th>$b_1$, $b_2$, $b_3$, $b_4$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$a$</td>
<td>$b_1$, $b_2$, $b_3$, $b_4$</td>
</tr>
<tr>
<td>$b_1$</td>
<td>$b_2$, $b_3$, $b_4$</td>
</tr>
<tr>
<td>$b_2$</td>
<td>$b_3$, $b_4$</td>
</tr>
<tr>
<td>$b_3$</td>
<td>$b_4$</td>
</tr>
<tr>
<td>$b_4$</td>
<td>$b_4$</td>
</tr>
</tbody>
</table>
Polynomial XPath evaluation algorithms

Complexity of the bottom-up evaluation algorithm E(Q)

- Time complexity: O(|D|^5 x |Q|^2)
- Space complexity: O(|D|^4 x |Q|^2)

Based on storing all context-value tables

More efficient algorithms described in [GKP05):

1. Top-down
   - Time complexity: O(|D|^4 x |Q|^2)
   - Space complexity: O(|D|^4 x |Q|^2)

2. Algorithm MinContext [GKP05]
   - Time complexity: O(|D|^4 x |Q|^2)
   - Space complexity: O(|D|^2 x |Q|^2)

Avoiding exponentials is good, but |D|^2 may be unacceptable for large documents.

References for streaming and in-memory evaluation

References


