Part XVI

Serialization, Shredding, and More on *Pre*/*Post* Encoding

Outline of this part

Serialization ($\mathcal{E}^{-1}$)

Any encoding of XML documents into some database representation is typically meant to be *the only* representation of the stored XML documents.

- In particular, the original textual (serialized) form of the input XML documents will not be available, and
- XQuery expressions may construct **entirely new** documents.

Communicating the XML result of XQuery evaluation (dump to console, send over the wire), requires a process **inverse to encoding** $\mathcal{E}$ and is referred to as **serialization** ($\mathcal{E}^{-1}$).

W3C `http://www.w3.org/TR/xslt-xquery-serialization/`

Serialization & *pre*/*post* encoding

1. For XML elements, document order coincides with the relative order of opening tags in serialized XML text.
   - ⇒ We thus scan the nodes v in table `accel` in ascending *pre* column order and can **emit opening tags** as we scan.
   - Then push v onto a **stack** S to remember that we still need to print the closing tag of v.
2. Likewise, the postorder rank of v encodes the relative order of closing tags in the serialized XML text.
   - ⇒ **Emit closing tags** of nodes v' on stack S with $post(v') < post(v)$ before we process v itself.

Serialization & *pre/post* encoding

serialize(T): serialize encodings in table T

```
for v in T in ascending pre(v) order do
    while not(S.empty()) ∧ post(S.top()) < post(v) do
        print('</', name(S.top()), '>');
        S.pop();
    if kind(v) = elem then
        print('<', name(v), '>');
        S.push(v);
    else
        { process other node kinds here }
while not(S.empty()) do
    print('</', name(S.top()), '>');
    S.pop();
```


Serialization & *pre/post* encoding

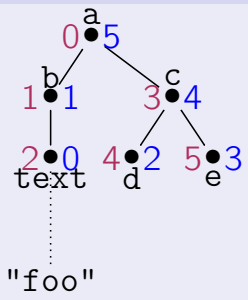
1. To serialize an encoded XML document in its entirety, invoke *serialize*(`accel`).
2. To serialize the XML fragment with root element v, invoke *serialize*($\cdot$) on the result of query Q, where

$$Q \equiv \mathit{path}(v\texttt{/descendant-or-self::node()}) \ .$$

Serialization: Example (1)

Sample XML fragment and *pre/post* encoding

```
<a>
  <b>foo</b>
  <c>
    <d/><e/>
  </c>
</a>
```



pre	*post*	*kind*	*tag*	*text*
0	5	*elem*	a	NULL
1	1	*elem*	b	NULL
2	0	*text*	NULL	foo
3	4	*elem*	c	NULL
4	2	*elem*	d	NULL
5	3	*elem*	e	NULL

To ensure a scan in order of the *pre* column, perform a forward scan of the `ipre` index ($\rightarrow$ yields RIDs).

- A function invocation like $kind(v)$ in $serialize(\cdot)$ thus corresponds to an RID-based tuple access on table `accel`.

Serialization: Example (2)

Scan of *pre/post* encoding

`accel`

	pre	*post*	*kind*	*tag*	*text*
$\rightarrow$	0	5	*elem*	a	NULL
	1	1	*elem*	b	NULL
	2	0	*text*	NULL	foo
	3	4	*elem*	c	NULL
	4	2	*elem*	d	NULL
	5	3	*elem*	e	NULL

Stack S

S

pre	*post*	*kind*	*tag*	*text*
0	5	*elem*	a	NULL

Output (console)

```
<a>
```

Serialization: Example (3)

Scan of *pre*/*post* encoding

accel

	pre	*post*	*kind*	*tag*	*text*
	0	5	*elem*	a	NULL
→	1	1	*elem*	b	NULL
	2	0	*text*	NULL	foo
	3	4	*elem*	c	NULL
	4	2	*elem*	d	NULL
	5	3	*elem*	e	NULL

Stack *S*

S

pre	*post*	*kind*	*tag*	*text*
1	1	*elem*	b	NULL
0	5	*elem*	a	NULL

Output (console)

```
<a>
  <b>
```


Serialization: Example (4)

Scan of *pre*/*post* encoding

accel

	pre	*post*	*kind*	*tag*	*text*
	0	5	*elem*	a	NULL
	1	1	*elem*	b	NULL
→	2	0	*text*	NULL	foo
	3	4	*elem*	c	NULL
	4	2	*elem*	d	NULL
	5	3	*elem*	e	NULL

Stack *S*

S

pre	*post*	*kind*	*tag*	*text*
1	1	*elem*	b	NULL
0	5	*elem*	a	NULL

Output (console)

```
<a>
  <b>foo
```

Serialization: Example (5)

Scan of *pre/post* encoding

accel

	pre	*post*	*kind*	*tag*	*text*
	0	5	*elem*	a	NULL
	1	1	*elem*	b	NULL
	2	0	*text*	NULL	foo
→	3	4	*elem*	c	NULL
	4	2	*elem*	d	NULL
	5	3	*elem*	e	NULL

Stack *S*

S

pre	*post*	*kind*	*tag*	*text*
3	4	*elem*	c	NULL
0	5	*elem*	a	NULL

Output (console)

```
<a>
  <b>foo</b>
  <c>
```


Serialization: Example (5)

Scan of *pre/post* encoding

accel

	pre	*post*	*kind*	*tag*	*text*
	0	5	*elem*	a	NULL
	1	1	*elem*	b	NULL
	2	0	*text*	NULL	foo
	3	4	*elem*	c	NULL
→	4	2	*elem*	d	NULL
	5	3	*elem*	e	NULL

Stack *S*

S

pre	*post*	*kind*	*tag*	*text*
4	2	*elem*	d	NULL
3	4	*elem*	c	NULL
0	5	*elem*	a	NULL

Output (console)

```
<a>
  <b>foo</b>
  <c>
    <d>
```

Serialization: Example (6)

Scan of *pre/post* encoding

accel

	pre	*post*	*kind*	*tag*	*text*
	0	5	*elem*	a	NULL
	1	1	*elem*	b	NULL
	2	0	*text*	NULL	foo
	3	4	*elem*	c	NULL
	4	2	*elem*	d	NULL
→	5	3	*elem*	e	NULL

Stack *S*

S

pre	*post*	*kind*	*tag*	*text*
5	3	*elem*	e	NULL
3	4	*elem*	c	NULL
0	5	*elem*	a	NULL

Output (console)

```
<a>
  <b>foo</b>
  <c>
    <d></d><e>
```


Serialization: Example (7)

Scan of *pre/post* encoding

accel

	pre	*post*	*kind*	*tag*	*text*
	0	5	*elem*	a	NULL
	1	1	*elem*	b	NULL
	2	0	*text*	NULL	foo
	3	4	*elem*	c	NULL
	4	2	*elem*	d	NULL
×	5	3	*elem*	e	NULL

Stack *S*

S

pre	*post*	*kind*	*tag*	*text*
3	4	*elem*	c	NULL
0	5	*elem*	a	NULL

Output (console)

```
<a>
  <b>foo</b>
  <c>
    <d></d><e></e>
```

Serialization: Example (8)

Scan of *pre/post* encoding

accel

pre	*post*	*kind*	*tag*	*text*
0	5	*elem*	a	NULL
1	1	*elem*	b	NULL
2	0	*text*	NULL	foo
3	4	*elem*	c	NULL
4	2	*elem*	d	NULL
× 5	3	*elem*	e	NULL

Stack *S*

S

pre	*post*	*kind*	*tag*	*text*
0	5	*elem*	a	NULL

Output (console)

```
<a>
  <b>foo</b>
  <c>
    <d></d><e></e>
  </c>
```


Serialization: Example (9)

Scan of *pre/post* encoding

accel

pre	*post*	*kind*	*tag*	*text*
0	5	*elem*	a	NULL
1	1	*elem*	b	NULL
2	0	*text*	NULL	foo
3	4	*elem*	c	NULL
4	2	*elem*	d	NULL
× 5	3	*elem*	e	NULL

Stack *S*

S

pre	*post*	*kind*	*tag*	*text*

Output (console)

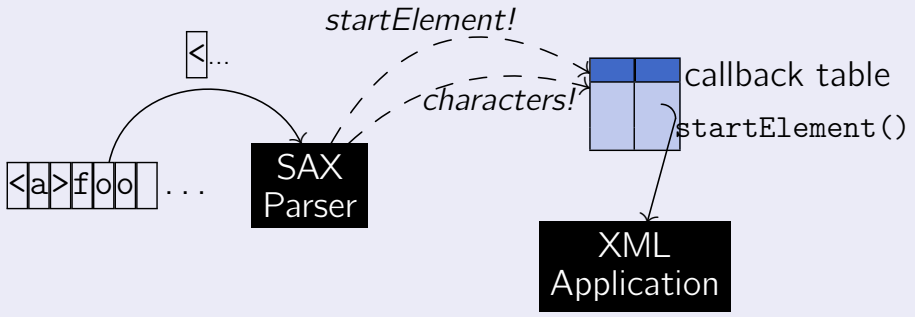
```
<a>
  <b>foo</b>
  <c>
    <d></d><e></e>
  </c>
</a>
```

SAX-based shredding ($\mathcal{E}$)

Recall that:

- **SAX** (*Simple API for XML*, `http://www.saxproject.org/`) parsers use constant space, regardless of XML input size.
- Communication between parser and client is **event-based** and does *not* involve an intermediate data structure.

SAX: Event-based XML parsing



SAX-based shredding

- A SAX parser reads its input (serialized XML) **sequentially** and **once** only, retaining no memory of what the parser has seen so far.
  - Selective memory may be built into the client, though.
- The client **acts on/ignores events** by populating a **function callback table.**
  - In effect, the client and the parser act **in parallel.**
- Here, we sketch the use of SAX to implement $\mathcal{E}$.

NB. SAX has more uses in the database-supported XML context, *e.g.*, the **stream-based evaluation of a subset of XPath** location steps (the so-called *forward axes*).

SAX callbacks for $\mathcal{E}$

The XPath Accelerator encoding table `accel` for an input XML document may readily be constructed in terms of few SAX **callback functions.**

- The callbacks perform SQL DML `INSERT` commands on table `accel` created via

```
CREATE TABLE accel (pre  INT PRIMARY KEY,
                    post INT UNIQUE NOT NULL,
                    par  INT,
                    kind INT(1),
                    tag  VARCHAR,
                    text VARCHAR)
```


SAX callbacks for $\mathcal{E}$

startDocument()

$pre \leftarrow 0$;
$post \leftarrow 0$;
create empty stack S;
$S.push(\langle pre, \sqcup, \texttt{NULL}, doc, \texttt{NULL}, \texttt{NULL}\rangle)$;
$pre \leftarrow pre + 1$;

startElement$(t, (a_1, v_1), \ldots, (a_n, v_n))$

$v \leftarrow \langle pre, \sqcup, S.top().pre, elem, t, \texttt{NULL}\rangle$;
$S.push(v)$;
$pre \leftarrow pre + 1$;
{ *process attributes* a_i *here* }

SAX callbacks for $\mathcal{E}$

endElement(*t*)

$v \leftarrow S.pop();$
$v.post \leftarrow post;$
`INSERT INTO accel VALUES` $v;$
$post \leftarrow post + 1;$

characters(*buf*)

$v \leftarrow \langle pre, post, S.top().pre, text, \texttt{NULL}, buf \rangle;$
`INSERT INTO accel VALUES` $v;$
$pre \leftarrow pre + 1;$
$post \leftarrow post + 1;$

SAX callbacks for $\mathcal{E}$

endDocument()

$v \leftarrow S.pop();$
$v.post \leftarrow post;$
`INSERT INTO accel VALUES` $v;$
`COMMIT WORK;`

✎ SAX-based XML document encoding ("shredding")

1. What is the maximum depth of stack S?
2. How can the shredder detect that the input is not well-formed (improper tag nesting)?
3. In which order are tuples inserted into `accel`?

SAX-based shredding: Example (1)

Input XML document

```
<?xml version="1.0"?>
<a>
  <b>foo</b>
  <c>
    <d></d><e></e>
  </c>
</a>
```

Current SAX event

startDocument()

Current *pre*, *post*

pre : 0 *post* : 0

Stack S

$\langle 0, \sqcup, \texttt{NULL}, \textit{doc}, \texttt{NULL}, \texttt{NULL}\rangle$

Table `accel`

pre	*post*	*par*	*kind*	*tag*	*text*

SAX-based shredding: Example (2)

Input XML document

```
<?xml version="1.0"?>
<a>
  <b>foo</b>
  <c>
    <d></d><e></e>
  </c>
</a>
```

Current SAX event

startElement(a)

Current *pre*, *post*

pre : 1 *post* : 0

Stack S

$\langle 1, \sqcup, 0, \textit{elem}, \texttt{a}, \texttt{NULL}\rangle$
$\langle 0, \sqcup, \texttt{NULL}, \textit{doc}, \texttt{NULL}, \texttt{NULL}\rangle$

Table `accel`

pre	*post*	*par*	*kind*	*tag*	*text*

SAX-based shredding: Example (3)

Input XML document

```
<?xml version="1.0"?>
<a>
  <b>foo</b>
  <c>
    <d></d><e></e>
  </c>
</a>
```

Current SAX event

startElement(b)

Current *pre*, *post*

pre : 2 *post* : 0

Stack S

⟨2, ␣, 1	, *elem*, b	, NULL⟩	
⟨1, ␣, 0	, *elem*, a	, NULL⟩	
⟨0, ␣, NULL	, *doc*	, NULL, NULL⟩	

Table accel

pre	*post*	*par*	*kind*	*tag*	*text*

SAX-based shredding: Example (4)

Input XML document

```
<?xml version="1.0"?>
<a>
  <b>foo</b>
  <c>
    <d></d><e></e>
  </c>
</a>
```

Current SAX event

characters(foo)

Current *pre*, *post*

pre : 3 *post* : 0

Stack S

⟨2, ␣, 1	, *elem*, b	, NULL⟩	
⟨1, ␣, 0	, *elem*, a	, NULL⟩	
⟨0, ␣, NULL	, *doc*	, NULL, NULL⟩	

Table accel

pre	*post*	*par*	*kind*	*tag*	*text*
3	0	2	*text*	NULL	foo

SAX-based shredding: Example (5)

Input XML document

```
<?xml version="1.0"?>
<a>
  <b>foo</b>
  <c>
    <d></d><e></e>
  </c>
</a>
```

Current SAX event

endElement(b)

Current *pre*, *post*

pre : 4 *post* : 1

Stack S

⟨1, ␣, 0	, *elem*, a	, NULL⟩			
⟨0, ␣, NULL, *doc*	, NULL, NULL⟩				

Table accel

pre	*post*	*par*	*kind*	*tag*	*text*
3	0	2	*text*	NULL	foo
2	1	1	*elem*	b	NULL

SAX-based shredding: Example (6)

Input XML document

```
<?xml version="1.0"?>
<a>
  <b>foo</b>
  <c>
    <d></d><e></e>
  </c>
</a>
```

Current SAX event

startElement(c)

Current *pre*, *post*

pre : 4 *post* : 2

Stack S

⟨4, ␣, 1	, *elem*, c	, NULL⟩			
⟨1, ␣, 0	, *elem*, a	, NULL⟩			
⟨0, ␣, NULL, *doc*	, NULL, NULL⟩				

Table accel

pre	*post*	*par*	*kind*	*tag*	*text*
3	0	2	*text*	NULL	foo
2	1	1	*elem*	b	NULL

Completing the *pre*/*post* encoding table layout

- As discussed up to now, table `accel` lacks some critical details to really support XQuery evaluation. We need to
  1. add support for **attribute** nodes,
  2. reflect the fact that **multiple tree fragments** may be constructed by an XQuery compression (with more than one fragment "alive" at a time),
  3. add support for **multiple documents** referenced in a single query.

"Alive" fragments and XPath evaluation

Multiple alive fragments in a single XQuery expression

```
let $a := <a><b><c/></b></a>
let $d := <d><e/></d>
return ($a/b/following::node(), $d)
```

- Fragments bound to variables `$a` and `$d` are encoded in a table of **transient** trees:

Alive fragments at

pre	*post*	···	*tag*	···
0	2		a	
1	1		b	
2	0		c	
3	4		d	
4	3		e	

- Axis `following::node()` at `b` produces `d`, `e`?

Attributes and XPath evaluation

Remember the XQuery DM: attribute nodes are *not* children of their containing elements.

Axes `child` *vs.* `attribute`

```
let $a := <a b="foo"><c/><!--d--></a>
return ($a/child::node(),
        $a/attribute::*,
        $a/(./child::node() | ./attribute::*))
```

⇓

```
(</c>, <!--d-->,
 attribute b {"foo"},
 attribute b {"foo"}, <c/>, <!--d-->)
```

⇒ Storing attribute nodes with other XML node kinds implies **filtering overhead** for both, the `attribute` axis and all other axes.