Part XII

Mapping XML to Databases

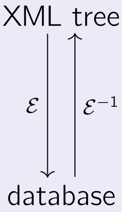
Outline of this part

Mapping XML to Databases

We now start to look at our preferred mapping direction:

- How do we put XML data into a database?
- . . . and how do we get it back *efficiently*?
- . . . and how do we run (XQuery) queries on them?

Mapping XML data to a database (and getting it back)



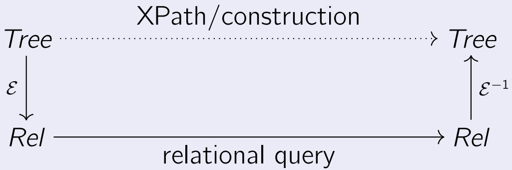
We will call the mapping $\mathcal{E}$ an *encoding* in the sequel.

Exploiting DB technology

In doing so, our main objective is to use as much of existing DB technology as possible (so as to avoid having to re-invent the wheel).

- **XQuery operations** on trees, XPath traversals and node construction in particular, should be **mapped into operations over the encoded database:**

  Our goal: let the database do the work!



- Obviously, $\mathcal{E}$ needs to be chosen judiciously. In particular, a faithful **back-mapping** $\mathcal{E}^{-1}$ is absolutely required.

How can we exploit DB technology?

1. Reuse knowledge gained by the DB community while you **implement a "native" XML database management system** from scratch.
   - It is often argued that, if you want to implement a new data model *efficiently*, there's no other choice.
2. Reuse existing DB technology and systems by defining an appropriate mapping of data structures and operations.
   - Often, *relational* DBMS technology is most promising, since it is most advanced and mature.
   - The challenge is to gain efficiency and not lose benchmarks against "native" systems!

Native XML processors

... need external memory representations of XML documents, too!

- Main-memory representations, such as a DOM tree, are insufficient, since they are only suited for "toy" examples (even with today's huge main memories, you want *persistent* storage).
- Obviously, native XML databases have more choices than those offered *on top of* a relational DBMS.
- We will have to see whether this additional freedom buys us significant performance gains, and
- what price is incurred for "replicating" RDBMS functionality.

Relational XML processors (1)

Recall our principal mission in this course:

Database-supported XML processors

We will use **relational database technology** to develop a highly efficient, scalable processor for **XML** languages like XPath, XQuery, and XML Schema.

We aim at a **truly (or purely) relational approach** here:

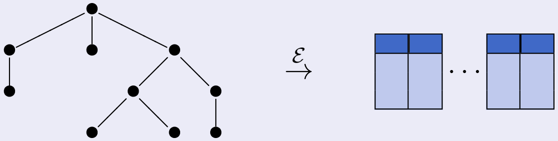
- Re-use existing relational database infrastructure—table storage layer and indexes (*e.g.*, B-trees), SQL or algebraic query engine and optimizer—and invade the database kernel in a very limited fashion (or, ideally, not at all).

Relational XML processors (2)

Our approach to **relational XQuery processing:**

- The XQuery data model—ordered, unranked trees and ordered item sequences—is, in a sense, alien to a relational database kernel.
- A **relational tree encoding** $\mathcal{E}$ is required to map trees into the relational domain, *i.e.*, tables.

Relational tree encoding $\mathcal{E}$



What makes a good (relational) (XML) tree encoding?

Hard requirements:

1. $\mathcal{E}$ is required to reflect **document order** and **node identity.**
   - *Otherwise*: cannot enforce XPath semantics, cannot support << and `is`, cannot support node construction.
2. $\mathcal{E}$ is required to encode the **XQuery DM node properties.**
   - *Otherwise*: cannot support XPath axes, cannot support XPath node tests, cannot support atomization, cannot support validation.
3. $\mathcal{E}$ is able to encode any well-formed **schema-less** XML fragment (*i.e.*, $\mathcal{E}$ is "**schema-oblivious**", see below).
   - *Otherwise*: cannot process non-validated XML documents, cannot support arbitrary node construction.

What makes a good (relational) (XML) tree encoding?

Soft requirements (primarily motivated by performance concerns):

4. **Data-bound operations** on trees (potentially delivering/copying lots of nodes) should map into efficient database operations.
   - *XPath location steps* (12 axes)
5. Principal, recurring **operations imposed by the XQuery semantics** should map into efficient database operations.
   - *Subtree traversal* (atomization, element construction, serialization).

For a relational encoding, "database operations" always mean "table operations" . . .

Dead end #1: Large object blocks

- Import **serialized XML fragment as-is** into tuple fields of type `CLOB` or `BLOB`:

<u>uri</u>	xml	
"foo.xml"	foo···	···

  - The `CLOB` column content is monolithic and **opaque with respect to the relational query engine**: a relational query cannot inspect the fragment (but extract and reproduce it).
  - The database kernel needs to incorporate (or communicate with) an **extra XML/XPath/XQuery processor** ⇒ frequent re-parsing will occur.
  - This is *not* a relational encoding in our sense.
  - But: see SQL/XML functionality mentioned earlier!

Dead end #2: Schema-based encoding

XML address database (excerpt)

```
<person>
  <name><first>John</first><last>Foo</last></name>
  <address><street>13 Main St</street>
           <zip>12345</zip><city>Miami</city>
  </address>
</person>
<person>
  <name><first>Erik</first><last>Bar</last></name>
  <address><street>42 Kings Rd</street>
           <zip>54321</zip><city>New York</city>
  </address>
</person>
```

Schema-based relational encoding: table person

id	first	last	street	zip	city
0	John	Foo	13 Main St	12345	Miami
1	Erik	Bar	42 Kings Rd	54321	New York

Dead end #2: Schema-based encoding

- Note that the schema of the "encoding" relation assumes a quite regular element nesting in the source XML fragment.
    - This regularity either needs to be discovered (during XML encoding) or read off a **DTD** or **XML Schema description.**
    - Relation `person` is **tailored to capture the specific regularities** found in the fragment.
- **Further issues:**
    - This encodes **element-only content** only (*i.e.*, content of type `element(*)*` or `text()`) and fails for **mixed content.**
    - Lack of any support for the XPath horizontal axes (*e.g.*, `following`, `preceding-sibling`).

Dead end #2: Schema-based encoding

Irregular hierarchy

```
<a no="0">
  <b><c>X</c><c/></b>
</a>
<a no="1">
  <b><c>Y</c></b>
</a>
<a><b/></a>
<a no="3"/>
```

A relational encoding

id	@no	b
0	0	α
3	1	β
5	NULL^a	γ
6	3	NULL^b

id	b	c
1	α	X
2	α	NULL^c
4	β	Y

Issues:

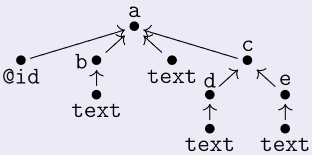
- Number of encoding tables depends on nesting depth.
- Empty element c encoded by NULL^c, empty element b encoded by absence of γ (will need *outer join* on column b).
- NULL^a encodes absence of attribute, NULL^b encodes absence of element.
- Document order/identity of b elements only implicit.

Dead end #3: Adjacency-based encoding

Adjacency-based encoding of XML fragments

```
<a id="0">
   <b>fo</b>o
   <c>
     <d>b</d><e>ar</e>
   </c>
</a>
```

≡



Resulting relational encoding

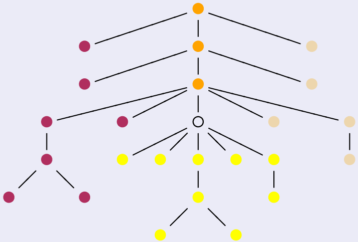
id	parent	tag	text	val
0	NULL	a	NULL	NULL
1	0	@id	NULL	"0"
2	0	b	NULL	NULL
3	2	NULL	"fo"	NULL
4	0	NULL	"o"	NULL
5	0	c	NULL	NULL
⋮				

Dead end #3: Adjacency-based encoding

- **Pro:**
  - Since this captures all adjacency, kind, and content information, we can—in principle—**serialize the original XML fragment.**
  - **Node identity** and **document order** is adequately represented.
- **Contra:**
  - The XQuery processing model is not well-supported: subtree traversals require **extra-relational** queries (**recursion**).
  - This is completely `parent–child` centric. How to support `descendant`, `ancestor`, `following`, or `preceding`?

Tree partitions and XPath axes

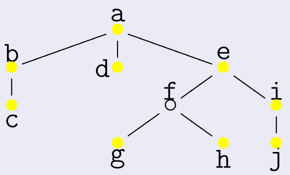
Axes: `descendant`, `ancestor`, `preceding`, `following`



Given an **arbitrary context node** ∘, the XPath axes `descendant`, `ancestor`, `preceding`, `following` **cover** and **partition the tree** containing ∘.

Tree partitions and XPath axes

Context node (here: f) is arbitrary



$$\{\mathtt{a}\ldots\mathtt{j}\} \quad = \quad \{\mathtt{f}\} \cup \bigcup_{\substack{\alpha \in \{\mathtt{preceding}, \mathtt{descendant},\\ \mathtt{ancestor}, \mathtt{following}\}}} \mathtt{f}/\alpha\mathtt{::node()}$$

NB: Here we assume that no node is an attribute node. Attributes treated separately (recall the XPath semantics).

The XPath Accelerator tree encoding

We will now introduce the **XPath Accelerator**, a relational tree encoding based on this observation.

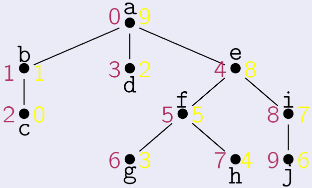
- If we can exploit the partitioning property, the encoding will represent each tree node exactly once.
- In a sense, the semantics of the XPath axes `descendant`, `ancestor`, `preceding`, and `following` will be "built into" the encoding ⇒ **"XPath awareness"**.
- XPath accelerator is **schema-oblivious** and **node-based:** each node maps into a row in the relational encoding.

Pre-order and post-order traversal ranks

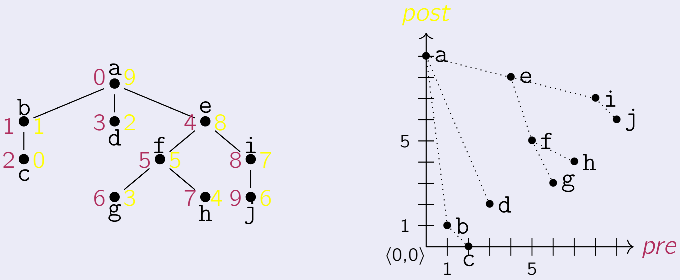
Pre-order/post-order traversal

(During a single scan through the document:) To each node v, assign its **pre-order** and **post-order** traversal ranks $\langle pre(v), post(v)\rangle$.

Pre-order/post-order traversal rank assignment



Pre-order/post-order: Tree isomorphism

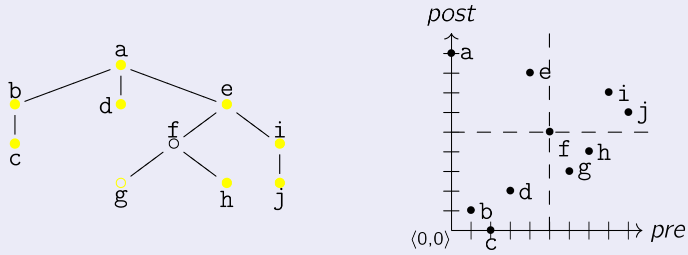


$pre(v)$ encodes document order and node identity

$v_1 \ll v_2 \iff pre(v_1) < pre(v_2)$ | v_1 `is` $v_2 \iff pre(v_1) = pre(v_2)$

XPath axes in the pre/post plane

Plane partitions ≡ XPath axes, ∘ is arbitrary!



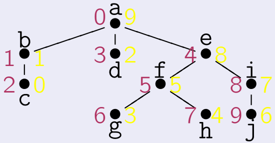
Pre/post plane regions ≡ major XPath axes

The **major XPath axes** `descendant`, `ancestor`, `following`, `preceding` correspond to rectangular **pre/post plane windows.**

XPath Accelerator encoding

XML fragment *f* and its skeleton tree

```
<a>
  <b>c</b>
  <!--d-->
  <e><f><g/><?h?></f>
     <i>j</i>
  </e>
</a>
```



Pre/post encoding of *f*: table accel

pre	*post*	*par*	*kind*	*tag*	*text*
0	9	NULL	elem	a	NULL
1	1	0	elem	b	NULL
2	0	1	text	NULL	c
3	2	0	com	NULL	d
4	8	0	elem	e	NULL
5	5	4	elem	f	NULL
6	3	5	elem	g	NULL
7	4	5	pi	NULL	h
8	7	4	elem	i	NULL
9	6	8	text	NULL	j

Relational evaluation of XPath location steps

Evaluate an XPath location step by means of a **window query** on the *pre*/*post* plane.

1. Table `accel` encodes an XML fragment,
2. table `context` encodes the **context node sequence** (in XPath accelerator encoding).

XPath location step (axis α) $\Rightarrow$ SQL window query

$$
\begin{array}{rl}
\texttt{SELECT} & \texttt{DISTINCT}\ v'.* \\
\texttt{FROM} & \texttt{context}\ v, \texttt{accel}\ v' \\
\texttt{WHERE} & v'\ \texttt{INSIDE}\ \mathit{window}(\alpha, v) \\
\texttt{ORDER BY} & v'.\mathit{pre}
\end{array}
$$

10 XPath axes[41] and *pre*/*post* plane windows

Window def's for axis α, name test t ($* =$ *don't care*)

Axis α	Query window $window(\alpha::t, v)$				
	pre	*post*	*par*	*kind*	*tag*
`child`	$\langle\, (v.pre, *)$	$(*, v.post)$	$v.pre$	$elem$	$t\,\rangle$
`descendant`	$\langle\, (v.pre, *)$	$(*, v.post)$	$*$	$elem$	$t\,\rangle$
`descendant-or-self`	$\langle\, [v.pre, *)$	$(*, v.post]$	$*$	$elem$	$t\,\rangle$
`parent`	$\langle\, v.par$	$(v.post, *)$	$*$	$elem$	$t\,\rangle$
`ancestor`	$\langle\, (*, v.pre)$	$(v.post, *)$	$*$	$elem$	$t\,\rangle$
`ancestor-or-self`	$\langle\, (*, v.pre]$	$[v.post, *)$	$*$	$elem$	$t\,\rangle$
`following`	$\langle\, (v.pre, *)$	$(v.post, *)$	$*$	$elem$	$t\,\rangle$
`preceding`	$\langle\, (*, v.pre)$	$(*, v.post)$	$*$	$elem$	$t\,\rangle$
`following-sibling`	$\langle\, (v.pre, *)$	$(v.post, *)$	$v.par$	$elem$	$t\,\rangle$
`preceding-sibling`	$\langle\, (*, v.pre)$	$(*, v.post)$	$v.par$	$elem$	$t\,\rangle$

[41] Missing axes in this definition: `self` and `attribute`.

Pre/*post* plane window ⇒ SQL predicate

`descendant::foo`, context node v

$$v' \ \texttt{INSIDE} \ \langle (v.pre, *), (*, v.post), *, elem, \texttt{foo} \rangle$$

$$\equiv$$

$$v'.pre > v.pre \ \texttt{AND} \ v'.post < v.post \ \texttt{AND}$$
$$v'.kind = elem \ \texttt{AND} \ v'.tag = \texttt{foo}$$

`ancestor-or-self::*`, context node v

$$v' \ \texttt{INSIDE} \ \langle (*, v.pre], [v.post, *), *, elem, * \rangle$$

$$\equiv$$

$$v'.pre <= v.pre \ \texttt{AND} \ v'.post >= v.post \ \texttt{AND}$$
$$v'.kind = elem$$

(e,f)/descendant::node()

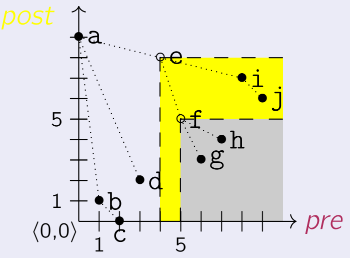
Context & frag. encodings

context

pre	*post*	···
5	5	
4	8	

accel

pre	*post*	···
0	9	
1	1	
2	0	
3	2	
4	8	
5	5	
6	3	
7	4	
8	7	
9	6	



SQL query with expanded *window*() predicate

```
SELECT    DISTINCT v1.*
FROM      context v, accel v1
WHERE     v1.pre > v.pre AND v1.post < v.post
ORDER BY  v1.pre
```

Compiling XPath into SQL

path: an XPath to SQL compilation scheme (sketch)

$$path(\texttt{fn:root()}) = \begin{array}{rl} \texttt{SELECT} & v'.* \\ \texttt{FROM} & accel\ v' \\ \texttt{WHERE} & v'.pre = 0 \end{array}$$

$$path(c/\alpha) = \begin{array}{rl} \texttt{SELECT} & \texttt{DISTINCT}\ v'.* \\ \texttt{FROM} & path(c)\ v, accel\ v' \\ \texttt{WHERE} & v'\ \texttt{INSIDE}\ window(\alpha, v) \\ \texttt{ORDER BY} & v'.pre \end{array}$$

$$path(c[\alpha]) = \begin{array}{rl} \texttt{SELECT} & \texttt{DISTINCT}\ v.* \\ \texttt{FROM} & path(c)\ v, accel\ v' \\ \texttt{WHERE} & v'\ \texttt{INSIDE}\ window(\alpha, v) \\ \texttt{ORDER BY} & v.pre \end{array}$$

An example: Compiling XPath into SQL

Compile `fn:root()/descendant::a/child::text()`

$$\mathit{path}(\texttt{fn:root()/descendant::a/child::text()})$$

$$=$$

$$\begin{array}{rl} \texttt{SELECT DISTINCT} & v_1.* \\ \texttt{FROM} & \mathit{path}(\texttt{fn:root/descendant::a})\ v, \mathit{accel}\ v_1 \\ \texttt{WHERE} & v_1\ \texttt{INSIDE}\ \mathit{window}(\texttt{child::text()}, v) \\ \texttt{ORDER BY} & v_1.\mathit{pre} \end{array}$$

$$=$$

$$\begin{array}{rl} \texttt{SELECT DISTINCT} & v_1.* \\ \texttt{FROM} & \left(\begin{array}{rl} \texttt{SELECT DISTINCT} & v_2.* \\ \texttt{FROM} & \mathit{path}(\texttt{fn:root})\ v, \mathit{accel}\ v_2 \\ \texttt{WHERE} & v_2\ \texttt{INSIDE}\ \mathit{window}(\texttt{descendant::a}, v) \\ \texttt{ORDER BY} & v_2.\mathit{pre} \end{array} \right) v, \\ & \mathit{accel}\ v_1 \\ \texttt{WHERE} & v_1\ \texttt{INSIDE}\ \mathit{window}(\texttt{child::text()}, v) \\ \texttt{ORDER BY} & v_1.\mathit{pre} \end{array}$$

Does this lead to efficient SQL? Yes!

- Compilation scheme *path*$(\cdot)$ yields an SQL query of nesting depth n for an XPath location path of n steps.
  - On each nesting level, apply `ORDER BY` and `DISTINCT`.
- **Observations:**
  1. All but the outermost `ORDER BY` and `DISTINCT` clauses may be safely **removed.**
  2. The nested `SELECT-FROM-WHERE` blocks may be **unnested** without any effect on the query semantics.

Result of *path*(·) simplified and unnested

path(`fn:root()/descendant::a/child::text()`)

SELECT	DISTINCT v_1.*
FROM	*accel* v_3, *accel* v_2, *accel* v_1
WHERE	v_1 INSIDE *window*(child::text(), v_2)
AND	v_2 INSIDE *window*(descendant::a, v_3)
AND	v_3.*pre* = 0
ORDER BY	v_1.*pre*

- An XPath location path of *n* steps leads to an *n*-fold **self join** of encoding table *accel*.
- The join conditions are
  - **conjunctions** ✓ of
  - **range** or **equality predicates** ✓.

  } **multi-dimensional window!**