Why map relational database contents to XML?

- **Interoperability:** we may want to use (parts of) our RDB contents in many different application contexts (XML as data interchange format).
- **Reconstruction:** we might have stored (parts of) our XML documents in an RDBMS in the first place (RDBMS as XML store).
- **Dynamic XML contents:** we may use RDBMS queries to retrieve dynamic XML contents (*cf.* dynamic Web sites).
- **Wrapping:** everybody likes XML . . . , so why don't we give it to them?

Why do we look at that mapping?

What we're *really* interested in is the mapping in the opposite direction: *How to get XML into a database!*

- **Yes, but ...**
  - this one is easier to start with,
  - we do get some insight for the other mapping,
  - we can see some of the problems,
  - we'll see some of the "standard" XML benchmark data,
  - we'll see in what respect XML supports semi-structured data,
  - we'll learn more about SQL as well.

Representing relational tables in XML

. . . is easy, since they have such a simple structure:

- In a straightforward mapping, we generate elements for the relation, for the tuples, and for the attribute values.

Example

Consider a relational schema *Employees*(*eno*, *name*, *salary*, *phone*), and a corresponding table

Employees

eno	*name*	*salary*	*phone*
007	James	1,000,000	123 456

$\Rightarrow$

```
<Employees>
  ...
  <Employee>
    <eno>007</eno>
    <name>James</name>
    <salary>1,000,000</salary>
    <phone>123 456</phone>
  </Employee>
  ...
</Employees>
```

This is but *one* possible representation! There are many more . . .

Schemas of relational tables

- In the XML representation just shown, every `<Employee>` element "*carries the relational schema*" of the *Employees* relation.
- This can be considered some kind of "self-descriptive" representation.
  - As such, it incurs quite some (space) overhead—"attribute" names are "stored" *twice* with each value!
  - On the other hand, missing (`NULL`) values are easily represented *by leaving them out*.
  - Also, **deviations** from the given schema, such as extra attributes, would be covered easily ($\rightarrow$ *semi*-structured data).
- Even more self-descriptive representations can be chosen ...

Fully self-descriptive table representation

Completely generic XML "table" representation

```
<relation name="Employees">
  ...
  <tuple>
    <attribute name="eno">007</attribute>
    <attribute name="name">James</attribute>
    <attribute name="salary">1,000,000</attribute>
    <attribute name="phone">123 456</attribute>
  </tuple>
  ...
</relation>
```

Obviously, we could also represent table and attribute names using additional XML elements instead of XML attributes.

Deriving DTDs for relational schemas

Given the schema of a relational table, we can generate a DTD that describes our chosen XML representation.

DTD for the (first) XML representation of the *Employees* relation

```
<!DOCTYPE Employees [
  <!ELEMENT Employees (Employee*) >
  <!ELEMENT Employee  (eno, name, salary, phone) >
  <!ELEMENT eno       (#PCDATA) >
  <!ELEMENT name      (#PCDATA) >
  <!ELEMENT salary    (#PCDATA) >
  <!ELEMENT phone     (#PCDATA) >
]>
```

- Optional attributes (`NULL` allowed) can be characterized as such in the element specification for `Employee`, *e.g.*, "`... phone? ...`"
- All representations (and DTDs) can easily be extended to capture whole relational databases (as a collections of tables).

Beyond flat relational tables

Example: Nested Relation

A bibliography referring to journal articles might be described as a "**Nested Relation**" *Articles*, where each tuple has atomic attributes, *e.g.*, for *title*, *journal*, *year*, *pages*, as well as relation-valued attributes (*aka.* sub-relations), *e.g.*, *Authors* with a set of (*firstname*, *lastname*)-tuples and *Keywords*: (*keyword*, *weight*)-tuples:

$$Artcls(\ tit, jnl, yr, pp, Auths(fn, ln), Kwds(kw, wt)\ )$$

One tuple in that table might look like this:

Artcls

tit	*jnl*	*yr*	*pp*	*Auths*		*Kwds*	
				fn	*ln*	*kw*	*wt*
bla	jacm	2000	30–57	J.	Doe	java	0.9
				S.	Shoe	object	0.5
						pgmg	0.7

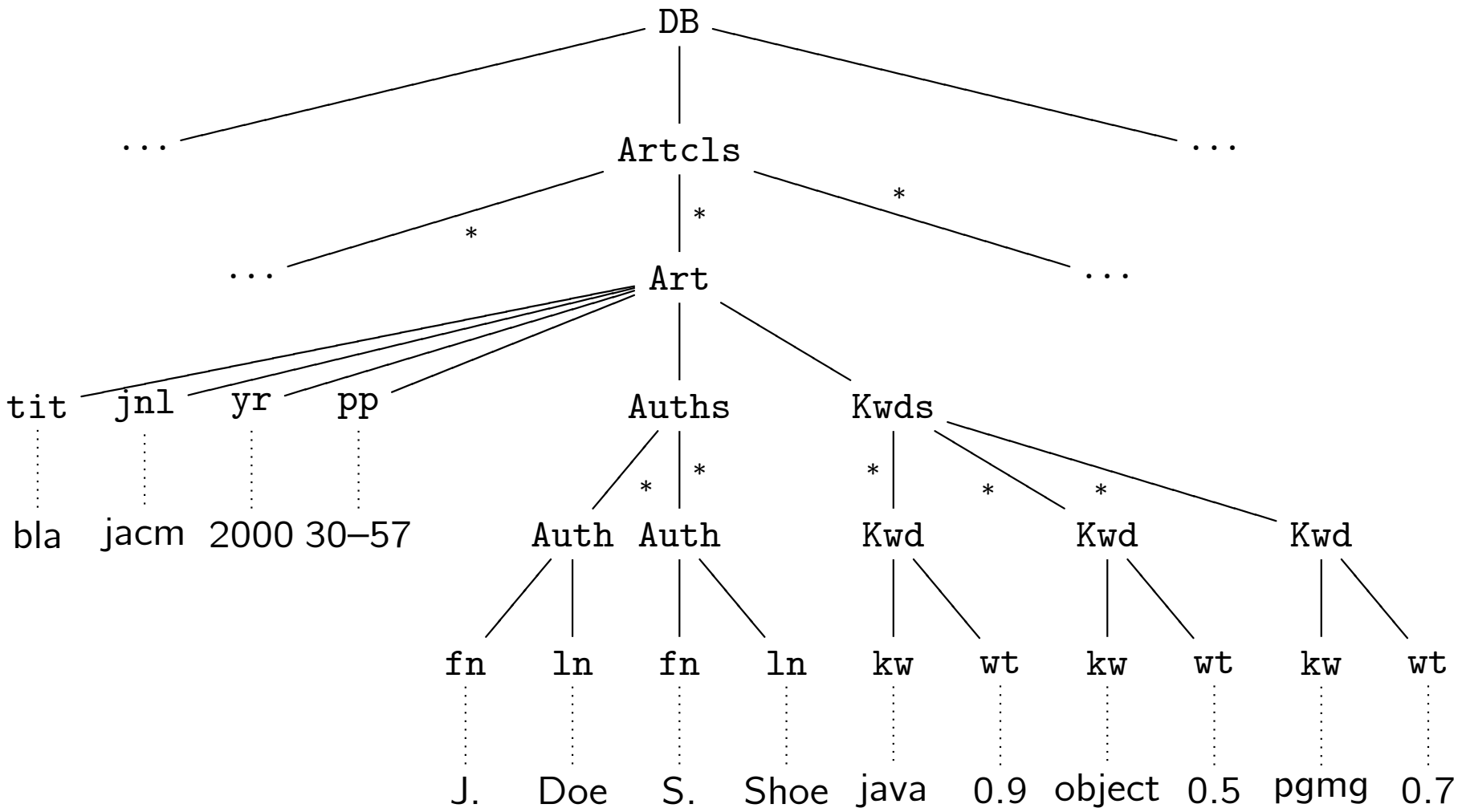
SQL-3 tables

SQL-3 offers a number of extensions beyond 1NF (flat) relations. For example, attributes may now be record-, array-, or (multi-)set-valued. Nested relations are thus part of the SQL standard!

Nested table *Artcls* can be described by the following DTD:

```
<!DOCTYPE Artcls [
  <!ELEMENT Artcls (Art*) >
  <!ELEMENT Art    ( tit, jnl, yr, pp, Auths, Kwds ) >
  <!ELEMENT tit    (#PCDATA) >
  <!ELEMENT jnl    (#PCDATA) >
  <!ELEMENT yr     (#PCDATA) >
  <!ELEMENT pp     (#PCDATA) >
  <!ELEMENT Auths  (Auth*) >
  <!ELEMENT Auth   ( fn, ln ) >
  <!ELEMENT fn     (#PCDATA) >
  <!ELEMENT ln     (#PCDATA) >
  <!ELEMENT Kwds   (Kwd*) >
  <!ELEMENT Kwd    ( kw, wt ) >
  <!ELEMENT kw     (#PCDATA) >
  <!ELEMENT wt     (#PCDATA) >
]>
```

XML tree of the example (including database node)



"*"-edges indicate possible repetition (set-valued elements).

Generating XML from within SQL

SQL/XML, a part of SQL:2003, allows the construction of XML fragments within a `SELECT–FROM–WHERE` query.

SQL/XML example 1: generate XML from (1NF) *Employees*-tuple

```
SELECT XMLELEMENT(NAME "Employee",
                  XMLATTRIBUTES(eno),
                  name) AS element
FROM   Employees
```

$\Downarrow$

element
⋮
`<Employee ENO="007">James</Employee>`
⋮

Generating XML from within SQL

SQL/XML example 2: generate XML from (1NF) *Employees*-tuple

```
SELECT XMLGEN('<Employee Name="{$name}">
                 <salary>{$salary/13}</salary>
               </Employee>') AS Empls
FROM   Employees
```

⇓

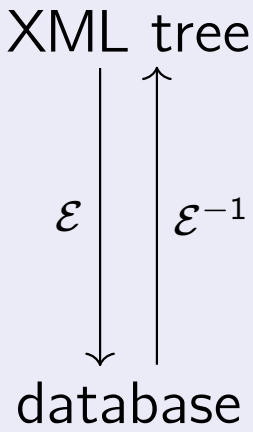
Empls
⋮
<Employee Name="James"> <salary>76923.077</salary> </Employee>
⋮

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)



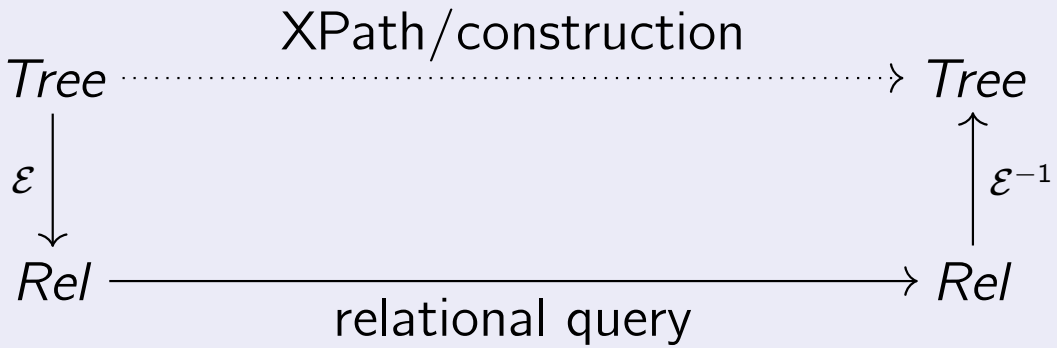
We will call the mapping $\mathcal{E}$ an *encoding* in the sequel.

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:

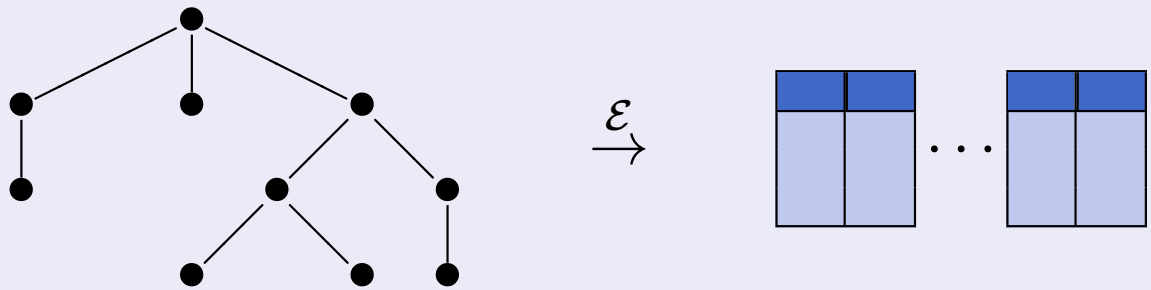
- Re-use existing relational database infrastructure—table storage layer and indexes (*e.g.*, B-trees), SQL or algebraic query engine and optimizer—and invade the database kernel in a very limited fashion (or, ideally, not at all).

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`:

uri	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** $\Rightarrow$ 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:

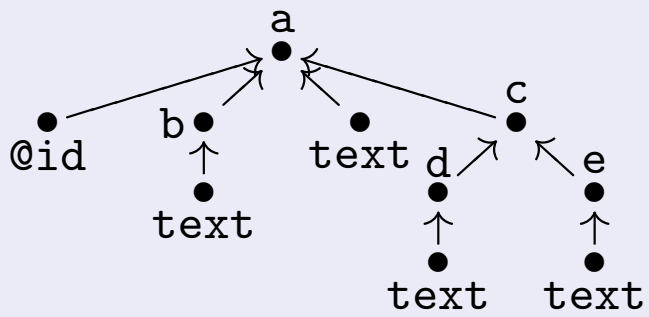
- Number of encoding tables depends on nesting depth.
- Empty element `c` encoded by NULL^c, empty element `b` encoded by absence of γ (will need *outer join* on column `b`).
- NULL^a encodes absence of attribute, NULL^b encodes absence of element.
- Document order/identity of `b` elements only implicit.

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`?

Node-based encoding

Several encoding schemes are based on an (appropriate) mapping of XML *nodes* onto relational tuples. Key questions are:

- How to represent *node IDs*, and
- how to represent XML-*structure*, in particular, *document order*.

Obviously, both questions are related, and—since we deal we *tree* structures—we might as well think of an *edge-based* representation scheme (in a tree, each non-root node has exactly one incoming edge!)

Most representations encode *document order* into node IDs by chosing an appropriately ordered ID domain.

Node IDs

Two very common approaches can be distinguished:

- XML nodes are numbered **sequentially** (in document order).
- XML nodes are numbered **hierarchically** (reflecting tree structure).

Observations:

- In both cases, node ID numbers are assigned automatically by the encoding scheme.
- Sequential numbering necessarily requires additional encoding means for capturing the tree structure.
- Both schemes represent document order by a (suitable) numeric order on the node ID numbers.
- Both schemes envisage problems when the document structure dynamically changes (due to updates to the document), since node ID numbers and document structure/order are related! (*see later*)

Sequential node ID numbering

Typically, XML nodes are numbered sequentially *in document order*.

- For an example, see the adjacency-based encoding above (`id`-attribute).
- IDs may be assigned *globally* (unique across the document) or *locally* (unique within the same parent node.)

Document structure needs to be represented separately, *e.g.*, by means of a "parent node ID" attribute (`par`).

In the most simple case (ignoring everything but "pure structure"), the resulting binary relational table

id	parent
⋮	⋮

could be considered a *node-based* (1 tuple per node ID) as well as an *edge-based* (1 tuple per edge) representation.[39]

[39] The edge-based representation would typically *not* include a tuple for the root node ID.

Hierarchical node ID numbering

While sequential numbering assigns *globally unique* IDs to all nodes, hierarchical numbering assigns node IDs that are *relative to* a node's parent node's ID.

Globally unique node IDs can then be obtained by (recursively) prepending parent node IDs to local node IDs. Typically, "dot notation" is used to separate the parts of those globally unique IDs:

$$\langle \texttt{rootID} \rangle.\langle \texttt{rootchildID} \rangle.\cdots.\langle \texttt{parentID} \rangle.\langle \texttt{nodeID} \rangle$$

Observations:

- In general, a node on level i of the tree (root = level 0) will have a global node ID with $i+1$ "components": $\langle \texttt{ID}_0 \rangle.\langle \texttt{ID}_1 \rangle.\cdots.\langle \texttt{ID}_i \rangle$
- Such IDs represent *tree structure* as well.
- (Local) node IDs need not be globally unique.
- This could also be considered a *path-based* representation.

Working with node-based encodings

Obviously, relational representations based on node-based encoding (traditionally called "edge table encodings") provide support for (bi-directional) parent-child traversal, name tests, and value-based predicates using the following kind of table:

edgetable

nodeID	parentID	elemname	value
⋮	⋮	⋮	⋮

As mentioned before, this table wastes space due to repetition of element names. Furthermore, to support certain kinds of path expressions, it may be beneficial to:

- store paths instead of element names, so as to
  - support path queries, while
  - introduce even more storage redundancy; thus
- use a separate ("path table") to store the paths together with path IDs.

Path table representation

Element names (or rather paths) can now be represented via path IDs in the edge table, pointing (as foreign keys) to the separate path table:

`edgetable`

nodeID	parentID	pathID	value
⋮	⋮	⋮	⋮

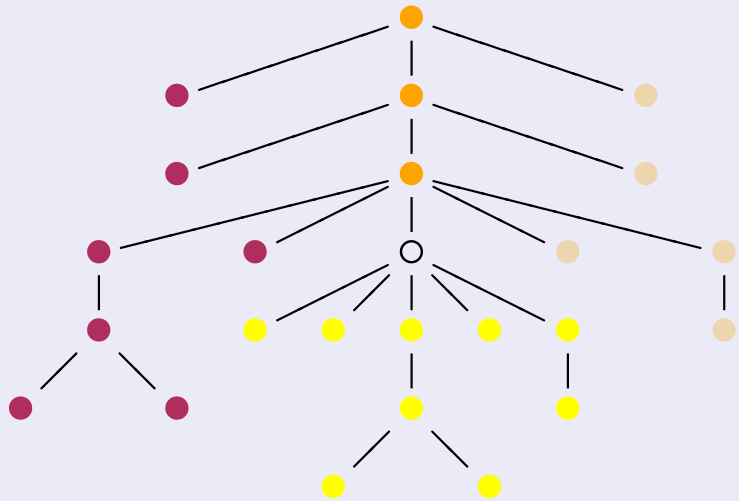
`pathtable`

pathID	path
⋮	⋮

Notice that the path table entries represent paths of the form `/bib/doc/author/name`, *i.e.*, they record paths that end in element names, not values. Hence, they are type- and not instance-specific: all document nodes that have identical root-to-element paths are represented by a *single* entry in the path table!

Tree partitions and XPath axes

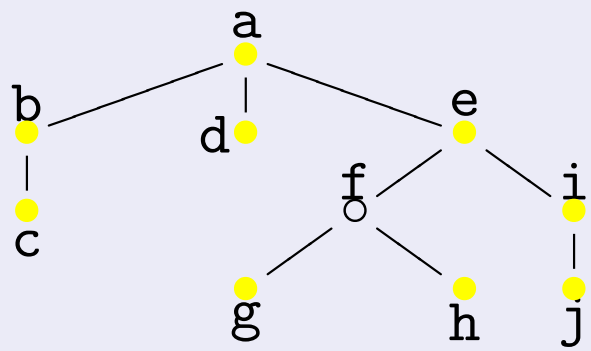
Axes: descendant, ancestor, preceding, following



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



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

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.

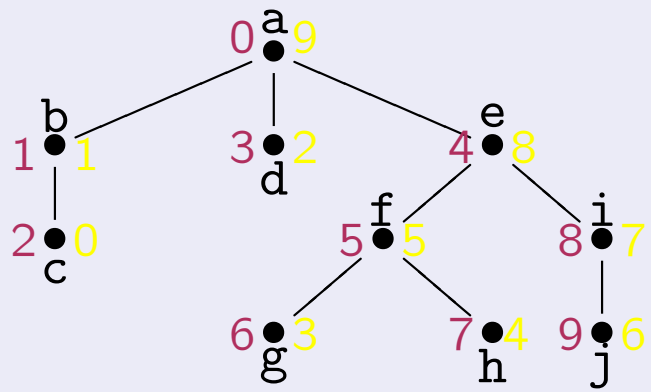
- If we can exploit the partitioning property, the encoding will represent each tree node exactly once.
- In a sense, the semantics of the XPath axes `descendant`, `ancestor`, `preceding`, and `following` will be "built into" the encoding ⇒ **"XPath awareness"**.
- XPath accelerator is **schema-oblivious**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

Pre-order and post-order traversal ranks

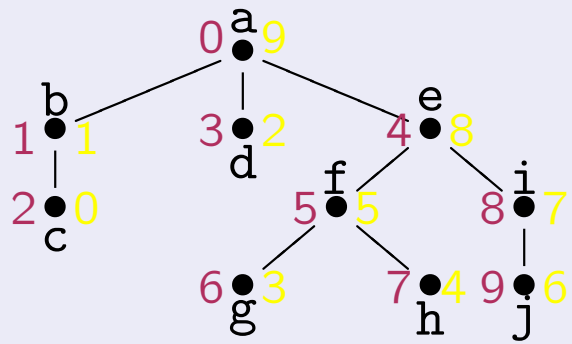
Pre-order/post-order traversal

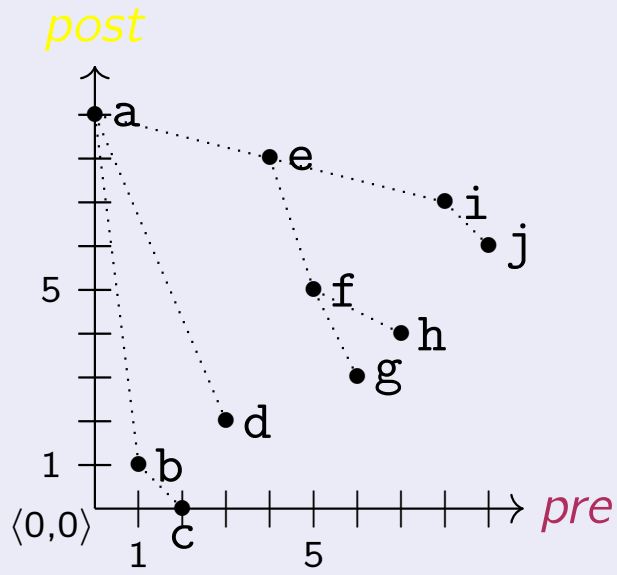
(During a single scan through the document:) To each node v, assign its **pre-order** and **post-order** traversal ranks $\langle pre(v), post(v) \rangle$.

Pre-order/post-order traversal rank assignment



Pre-order/post-order: Tree isomorphism



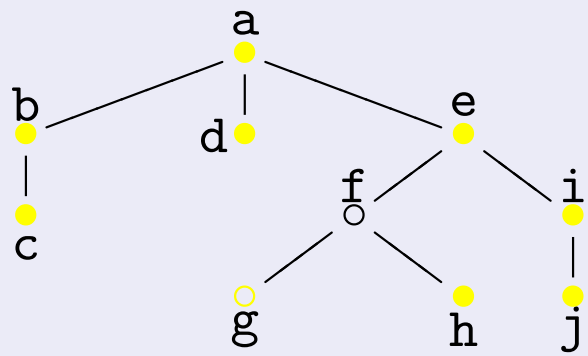


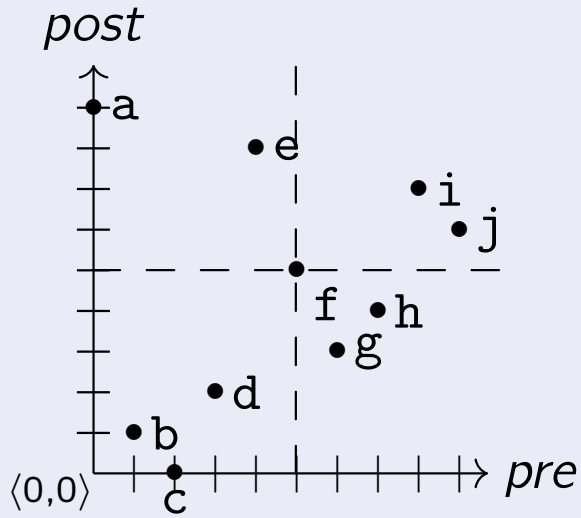
$pre(v)$ encodes document order and node identity

$$v_1 << v_2 \iff pre(v_1) < pre(v_2) \quad \Big| \quad v_1 \texttt{ is } v_2 \iff pre(v_1) = pre(v_2)$$

XPath axes in the pre/post plane

Plane partitions ≡ XPath axes, ∘ is arbitrary!





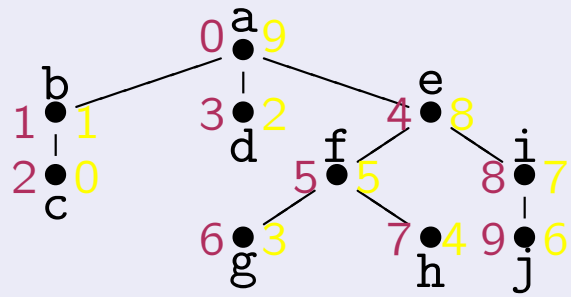
Pre/post plane regions ≡ major XPath axes

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[40] 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*	$t \rangle$

[40] 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()

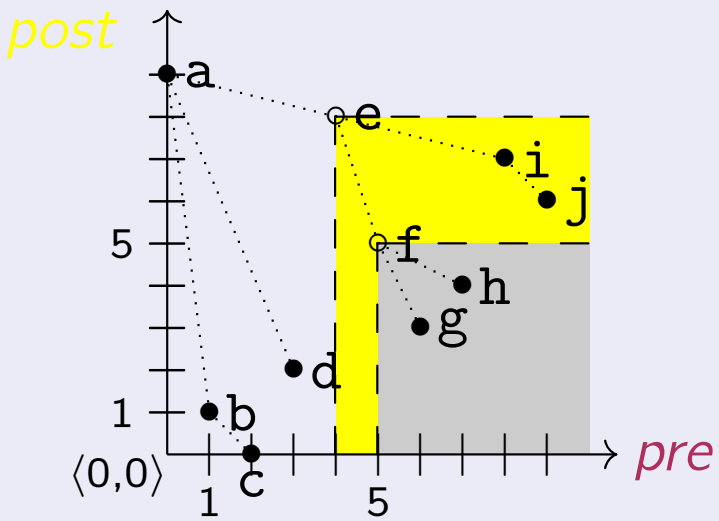
Context & frag. encodings

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()`

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

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 v1.*
  FROM   accel v3, accel v2, accel v1
 WHERE   v1 INSIDE window(child::text(), v2)
   AND   v2 INSIDE window(descendant::a, v3)
   AND   v3.pre = 0
ORDER BY v1.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!**