

# XML and Databases

Prof. Dr. Marc H. Scholl

`Marc.Scholl@uni-konstanz.de`

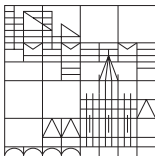
University of Konstanz

Dept. of Computer & Information Science

Databases and Information Systems Group

Winter 2005/06

(Most of the slides of this presentation have been prepared by Torsten Grust, now at TU Munich)



# Part I

## Preliminaries

# Outline of this part

## 1 Welcome

## 2 Overview

- XML
- XML and Databases

## 3 Organization

# Welcome

... to this course introducing you to the world of XML and the challenges of dealing with XML in a DBMS.

As a coarse outline, we will proceed as follows:

- 1 Introduction to XML
- 2 XML processing in general
- 3 Query languages for XML data
- 4 Mapping XML to databases
- 5 Database-aware implementation of XML query languages

# About XML

- XML is the World Wide Web Consortium's (W3C, <http://www.w3.org/>) **E**xtensible **M**arkup **L**anguage.
- We hope to convince you that XML is not yet another hyped TLA but useful technology.
- You will become best friends with one of the most important data structures in Computing Science, the **tree**. XML is all about tree-shaped data.
- You will learn how to apply a number of closely related **XML standards**:
  - ▶ Representing data: **XML** itself, **DTD**, **XMLSchema**, **XML** dialects.
  - ▶ Interfaces to connect programming languages to XML: **DOM**, **SAX**.
  - ▶ Languages to query and transform XML: **XPath**, **XQuery**, **XSLT**.

## More about XML

- We will talk about **algorithms and programming techniques** to efficiently manipulate XML data:
  - ▶ **Regular expressions** can be used to **validate** XML data,
  - ▶ **finite state machines** lie at the heart of highly efficient **XPath implementations**,
  - ▶ **tree traversals** may be used to preprocess XML trees in order to support **XPath evaluation**, to **store XML trees in databases**, *etc.*
- In the end you should be able to digest the thick pile of related W3C Xfoo<sup>1</sup> standards.

### What this course is not about:

- Hacking CGI scripts,
- HTML,
- Java (but see below).

---

<sup>1</sup>..., XQuery, XPointer, XLink, XHTML, XInclude, XML Schema, XML Base, ...

# XML and databases

We assume you are ...

- familiar with the general concepts & ideas behind relational databases,
- (somewhat) fluent in SQL,
- interested in systems' issues (such as, architecture & performance).

We'll try to achieve that you're familiar with ...

- the challenges in extending DB technology to deal with XML structured data,
- some of current research results in that area,
- possible application areas.

# Why database-supported XML?

- The structure implied by XML is less rigid than the traditional relational format.
  - ▶ We speak of **semi-structured** data.
- Several application domains can be modeled easier in XML.
  - ▶ E. g. content management systems, library databases
- Growing amounts of data are readily available in the XML format.
  - ▶ Think of current text processing or spreadsheet software.



# Problems

- Databases can handle huge amounts of data stored in **relations** easily.
  - ▶ Storage management, index structures, join or sort algorithms, ...
- The data model behind XML is the **tree**.
  - ▶ While we trivially represent relations with trees, the opposite is challenging.
- Structure is part of the data, implying novel tree operations.
  - ▶ We **navigate** through the XML tree, following a **path**.

## Example (XQuery)

```
for $x in fn:doc("bib.xml")/bib/books/book[author = "John Doe"]
where @price >= 42
return <expensive-book> { $x/title/text() } </expensive-book>
```

# Some of the challenges

- Existing technology cannot directly be applied to XML data.
  - ▶ How do we store trees?
  - ▶ Can we benefit from index structures?
  - ▶ How can we implement tree navigation?
- The W3C XQuery proposal poses additional challenges:
  - ▶ a notion of **order**,
  - ▶ a complex **type system**, and
  - ▶ the possibility to **construct new tree nodes** on the fly.

# Some solutions to be discussed

- Tree representation in relational databases
  - ▶ “Schema-based” methods, if we have regular data and know its structure
  - ▶ “Schema-oblivious” methods that can handle arbitrary XML trees
- Evaluation techniques for path queries
  - ▶ Step-by-step evaluation
  - ▶ Pattern based techniques that treat paths as a whole
- Index structures for XML
- XQuery evaluation
  - ▶ Support for the remaining features of XQuery
- Other database techniques
  - ▶ Streaming query evaluation
  - ▶ Query rewriting

# Organizational matters

- **Lectures:**

Monday, 16<sup>15</sup>–17<sup>45</sup> (C 252, lecture)

Tuesday, 14<sup>15</sup>–15<sup>45</sup> (C 252, lecture)

Thursday, 10<sup>15</sup>–11<sup>45</sup> (C 252, tutorial)

- **Office hours:**

Whenever our office doors (E211/E217) are open, you may want to drop an e-mail note before.

- **Course homepage:**

[www.inf.uni-konstanz.de/dbis/teaching/ws0506/database-xml/](http://www.inf.uni-konstanz.de/dbis/teaching/ws0506/database-xml/)

Download these slides, assignments, and various other good stuff from there.

- **Read your e-mail!**

Become a member of Unix group `xmlldb_W05` (→ account tool<sup>2</sup>).

---

<sup>2</sup>[www.inf.uni-konstanz.de/system/service/accounts/accounttool.html](http://www.inf.uni-konstanz.de/system/service/accounts/accounttool.html)

# How you will benefit most from this course

- Use the material provided on the course website to prepare for the lectures.
- Actively **participate in and work on the “paper-and-pencil” as well as the C/C++/Java programming assignments** scattered throughout the semester (→ Christian).
- Pass the (oral, unless you are a too big crowd) examination at the end of the semester.
- Have a look at various XML files that come across your way!
- Don't hesitate to ask questions; let us know if we can improve the lecture material and/or its presentation.
- Have fun!

# Questions?

- Questions ... ?
- Comments ... ?
- Suggestions ... ?

# Part II

## XML Basics

# Outline of this part

## 4 Markup Languages

- Early Markup
- An Application of Markup: A Comic Strip Finder



# Early markup languages

- The term **markup** has been coined by the **typesetting** community, *not* by computer scientists:
- With the advent of the printing press, writers and editors used (often marginal) notes to instruct printers to
  - ▶ select certain fonts,
  - ▶ let passages of text stand out,
  - ▶ indent a line of text, *etc.*
- Proofreaders use a special set of symbols, their special **markup language**, to identify typos, formatting glitches, and similar erroneous fragments of text.


**N.B.** The markup language is designed to be easily recognizable in the actual flow of text.

# Example



Reproduced from the “Duden”, 21st edition (1996), © Brockhaus AG.

81

## Korrekturvorschriften

11. **Überflüssige Buchstaben** oder **Wörter** werden durchgestrichen und auf dem Rand durch  (für: deletatur, d. h. „es werde getilgt“) angezeigt.




  

12. **Fehlende** oder **überflüssige Satzzeichen** werden wie fehlende oder überflüssige Buchstaben angezeigt  

13. **Verstellte Buchstaben** werden durchgestrichen und auf dem Rand in der richtigen Reihenfolge angegeben.


 

**Verstellte Wörter** durch    das Umstellungszeichen gekennzeichnet.

Die Wörter werden bei größeren Umstellungen beziffert.

**Verstellte Zahlen** sind immer ganz durchzustreichen und in der richtigen Ziffernfolge auf den Rand zu schreiben, z. B.  1684.

 1864.

14. Für **unleserliche** oder **zweifelhafte Manuskriptstellen**, die noch nicht blockiert sind, sowie für noch **zu ergänzenden Text** wird

- Computing Scientists adopted the markup idea—originally to **annotate program source code**:
  - ▶ Design the markup language such that its constructs are **easily recognizable** by a machine.
  - ▶ **Approaches**:
    - ① Markup is written using a **special set of characters**, disjoint from the set of characters that form the tokens of the program.
    - ② Markup occurs in places in the source file where program code may not appear (**program layout**).
- **Example** of ②: Fortran 77 fixed form source:
  - ▶ Fortran statements start in column 7 and do not exceed column 72,
  - ▶ a Fortran statement longer than 66 characters may be continued on the next line if a character  $\notin \{0, !, \_ \}$  is placed in column 6 of the continuing line,
  - ▶ comment lines start with a C or \* in column 1,
  - ▶ numeric labels (DO, FORMAT statements) have to be placed in columns 1–5.

## Fortran 77 source, fixed form, space characters made explicit ( )

## Fortran 77

```

1  C THIS PROGRAM CALCULATES THE CIRCUMFERENCE AND AREA OF A CIRCLE WITH
2  C RADIUS R.
3  C
4  C DEFINE VARIABLE NAMES:
5  C R = RADIUS OF CIRCLE
6  C PI = VALUE OF PI = 3.14159
7  C CIRCUM = CIRCUMFERENCE = 2 * PI * R
8  C AREA = AREA OF THE CIRCLE = PI * R * R
9  *****
10 C
11 C REAL R, CIRCUM, AREA
12 C
13 C PI = 3.14159
14 C
15 C SET VALUE OF R:
16 C R = 4.0
17 C
18 C CALCULATIONS:
19 C CIRCUM = 2. * PI * R
20 C AREA = PI * R * R
21 C
22 C WRITE RESULTS:
23 C WRITE(6,*) ' FOR A CIRCLE OF RADIUS ', R,
24 C ' THE CIRCUMFERENCE IS ', CIRCUM,
25 C ' AND THE AREA IS ', AREA
26 C
27 C END

```

- Increased computing power and more sophisticated parsing technology made fixed form source obsolete.
- Markup, however, is still being used on different levels in today's programming languages and systems:
  - ▶ ASCII defines a set of **non-printable characters** (the C0 control characters, code range 0x00–0x1f):

code	name	description
0x01	STX	start of heading
0x02	SOT	start of text
0x04	EOT	end of transmission
0x0a	LF	line feed
0x0d	CR	carriage return

- ▶ **Blocks** (containers) are defined using various form of matching **delimiters**:
  - ★ `begin ... end, \begin{foo} ... \end{foo}`
  - ★ `/* ... */ , { ... } , // ... LF`
  - ★ `do ... done, if ... fi, case ... esac, $[ ... ]`

# An Application of Markup: A Comic Strip Finder

## Problem:

- **Query a database of comic strips by content.** We want to approach the system with queries like:
  - 1 Find all strips featuring Dilbert but not Dogbert.
  - 2 Find all strips with Wally being angry with Dilbert.
  - 3 Show me all strips featuring characters talking about XML.

## Approach:

- Unless we have next<sup>n</sup> generation image recognition software available, we obviously have to **annotate** the comic strips to be able to process the queries above:

strips	bitmap	annotation
	⋮	⋮
		...Dilbert...Dogbert Wally...
	⋮	⋮

# Stage 1: ASCII-Level Markup



Copyright © 2000 United Feature Syndicate, Inc.  
Redistribution in whole or in part prohibited

## ASCII-Level Markup

```
1 Pointy-Haired Boss: >>Speed is the key to success.<<
2 Dilbert: >>Is it okay to do things wrong if we're really, really fast?<<
3 Pointy-Haired Boss: >>Um... No.<<
4 Wally: >>Now I'm all confused. Thank you very much.<<
```

- ASCII C0 character sequence 0x0d, 0x0a (CR, LF) divides lines,
- each line contains a character name, then a colon (:), then a line of speech (comic-speak: *bubble*),
- the contents of each bubble are delimited by >> and <<.

✍ Which kind of queries may we ask now?

And what kind of software do we need to complete the comic strip finder?

## Stage 2: HTML-Style Physical Markup

dilbert.html

```
1 <h1>Dilbert</h1>
2 <h2>Panel 1</h2>
3 <ul>
4   <li> <b>Pointy-Haired Boss</b> <em>Speed is the key
5     to success.</em>
6 </ul>
7 <h2>Panel 2</h2>
8 <ul>
9   <li> <b>Dilbert</b> <em>Is it okay to do things wrong
10    if we're really really fast?</em>
11 </ul>
12 <h2>Panel 3</h2>
13 <ul>
14   <li> <b>Pointy-Haired Boss</b> <em>Um... No.</em>
15   <li> <b>Wally</b> <em>Now I'm all confused.
16     Thank you very much.</em>
17 </ul>
```



# HTML: Observations

- HTML defines a number of markup **tags**, some of which are required to match (`<t>...</t>`).
- Note that HTML tags primarily describe **physical markup** (font size, font weight, indentation, ...)
- Physical markup is of limited use for the comic strip finder (the **tags do not reflect the structure of the comic content**).

## Stage 3: XML-Style Logical Markup

- We create a **set of tags that is customized** to represent the content of comics, *e.g.*:

```
<character>Dilbert </character>
```

```
<bubble>Speed is the key to success.</bubble>
```

- New types of queries may require new tags: No problem for XML!
  - ▶ Resulting set of tags forms a new markup language (**XML dialect**).
- **All** tags need to appear in **properly nested** pairs (*e.g.*,  
`<t>...<s>...</s>...</t>`).
- Tags can be freely nested to reflect the **logical structure** of the comic content.

### ✍ Parsing XML?

In comparison to the stage 1 ASCII-level markup parsing, how difficult do you rate the construction of an XML parser?

# In our example

dilbert.xml

```
1 <strip>
2   <panel>
3     <speech>
4       <character>Pointy-Haired Boss</character>
5       <bubble>Speed is the key to success.</bubble>
6     </speech>
7   </panel>
8   <panel>
9     <speech>
10      <character>Dilbert</character>
11      <bubble>Is it okay to do things wrong
12        if we're really, really fast?</bubble>
13    </speech>
14  </panel>
15  <panel>
16    <speech>
17      <character>Pointy-Haired Boss</character>
18      <bubble>Um... No.</bubble>
19    </speech>
20    <speech>
21      <character>Wally</character>
22      <bubble>Now I'm all confused.
23        Thank you very much.</bubble>
24    </speech>
25  </panel>
26 </strip>
```

## Stage 4: Full-Featured XML Markup

- Although fairly simplistic, the previous stage clearly constitutes an improvement.
- XML comes with a number of additional constructs which allow us to convey even more useful information, *e.g.*:
  - ▶ **Attributes** may be used to qualify tags (avoid the so-called *tag soup*). Instead of
    - ★ `<question>Is it okay ...?</question>`  
`<angry>Now I'm ...</angry>`use
    - ★ `<bubble tone="question">Is it okay ...?</bubble>`  
`<bubble tone="angry">Now I'm ...</bubble>`
  - ▶ **References** establish links internal to an XML document:  
Establish link target:
    - ★ `<character id="phb">The Pointy-Haired Boss</character>`Reference the target:
    - ★ `<bubble speaker="phb">Speed is the key to success.</bubble>`

## dilbert.xml

```
1 <?xml version="1.0" encoding="iso-8859-1"?>
2 <strip copyright="United Feature Syndicate" year="2000">
3   <prolog>
4     <series href="http://www.dilbert.com/">Dilbert</series>
5     <author>Scott Adams</author>
6     <characters>
7       <character id="phb">The Pointy-Haired Boss</character>
8       <character id="dilbert">Dilbert, The Engineer</character>
9       <character id="wally">Wally</character>
10      <character id="alice">Alice, The Technical Writer</character>
11    </characters>
12  </prolog>
13  <panels length="3">
14    <panel no="1">
15      <scene visible="phb">
16        Pointy-Haired Boss pointing to presentation slide.
17      </scene>
18      <bubbles>
19        <bubble speaker="phb">Speed is the key to success.</bubble>
20      </bubbles>
21    </panel>
22    <panel no="2">
23      <scene visible="wally dilbert alice">
24        Wally, Dilbert, and Alice sitting at conference table.
25      </scene>
26      <bubbles>
27        <bubble speaker="dilbert" to="phb" tone="question">
28          Is it ok to do things wrong if we're really, really fast?
29        </bubble>
30      </bubbles>
31    </panel>
32    <panel no="3">
33      <scene visible="wally dilbert">Wally turning to Dilbert, angrily.
34      </scene>
35      <bubbles>
36        <bubble speaker="phb" to="dilbert">Um... No.</bubble>
```

## Part III

### Well-Formed XML

# Outline of this part

## 5 Formalization of XML

- Elements
- Attributes
- Entities

## 6 Well-Formedness

- Context-free Properties
- Context-dependent Properties

## 7 XML Text Declarations

- XML Documents and Character Encoding
- Unicode
- XML and Unicode

## 8 The XML Processing Model

- The XML Information Set
- More XML Node Types

# Formalization of XML

- We will now try to approach XML in a slightly more formal way. The nuts and bolts of XML are pleasingly easy to grasp.
- This discussion will be based on *the* central XML technical specification:
  - ▶ **Extensible Markup Language (XML) 1.0 (Second Edition)**  
**W3C Recommendation 6 October 2000**  
(<http://www.w3.org/TR/REC-xml>)



## Visit the W3C site

This lecture does *not* try to be a “guided tour” through the XML-related W3C technical documents (*boring!*).

Instead we will cover the basic principles and most interesting ideas. Visit the W3C site and use the original W3C documents to get a full grasp of their contents.



# Elements

- The **element** is the main markup construct provided by XML.
  - ▶ Marked up document region (**element content**) enclosed in matching **start** end **closing (end) tags**:
    - ★ **start tag**: `<t>` (*t* is the **tag name**),
    - ★ matching **closing tag**: `</t>`

## Well-formed XML (fragments)

```
1 <foo> okay </foo>
2 <This-is-a-well-formed-XML-tag.> okay
3 </This-is-a-well-formed-XML-tag.>
4 <foo>okay</foo>
```

## Non-well-formed XML

```
1 <foo> oops </bar>
2 <foo> oops </Foo>
3 <foo> oops ... <EOT>
```

- Element content may contain document characters as well as **properly nested elements** so-called **mixed content**):

#### Well-formed XML

```
1 <foo><bar>
2     <baz> okay </baz>
3     </bar>
4     <ok> okay </ok> still okay
5 </foo>
```

#### Non-well-formed XML

```
1 <foo><bar> oops </foo></bar>
2 <foo><bar> oops </bar><bar> oops </foo></bar>
```

#### Check for proper nesting

Which data structure would you use to straightforwardly implement the check for proper nesting in an XML parser?

- Element content may be **empty**:
  - ▶ The fragments `<t> </t>` and `<t/>` are well-formed XML and considered equivalent.
- Element nesting establishes a **parent–child relationship between elements**:
  - ▶ In the XML fragment `<p><c>...</c>...<c'>...</c'></p>`,
    - ★ element *p* is *the* **parent** of elements *c*, *c'*,
    - ★ elements *c*, *c'* are **children** of element *p*,
    - ★ elements *c*, *c'* are **siblings**.
- There is exactly one element that encloses the **whole** XML content: the **root element**.

### Non-well-formed XML

```
1 <one>
2   one eins un
3 </one>
4 <two> two zwei deux </two>
```

# Attributes

- Elements may further be classified using **attributes**:  
(It is common practice to denote an attribute named *a* by @*a* in written text (**attribute** *a*).)

`<t a="..." a='...' ...> ...</t>`

- ▶ An attribute **value** is restricted to character data (attributes may *not* be nested),
- ▶ attributes are *not* considered to be children of the containing element (instead they are **owned** by the containing element).

## Well-formed XML (fragment)

```
1 <price currency="US$" multiplier='1'>
2   23.45
3 </price>
4 <price>
5   <currency>US$</currency>
6   <multiplier>1</multiplier>
7   23.45
8 </price>
```

# Entities

- In XML, document **content** and **markup** are specified using a *single set of characters*.
- The characters { <, >, &, ", ' } form pieces of XML markup and may instead be denoted by **predefined entities** if they actually represent content:

Character	Entity
<	&lt;
>	&gt;
&	&amp;
"	&quot;
'	&apos;

## Well-formed XML

1

```
<operators>Valid comparison operators are &lt;=, &gt;, &lt;.</operators>
```

- The XML entity facility is actually a versatile recursive **macro** expansion machinery (more on that later).

# Well-Formedness

- The W3C XML recommendation is actually more formal and rigid in defining the syntactical structure of XML:

*“A textual object is **well-formed** XML if,*

- ① *Taken as a whole, it **matches the production** labeled document.*
- ② *It meets all the **well-formedness constraints** given in this [the W3C XML Recommendation] specification. ...”*

# Well-formedness #1: Context-free Properties

- 1 All **context-free** properties of well-formed XML documents are concisely captured by a **grammar** (using an EBNF-style notation).
  - **Grammar**: system of **production (rule)s** of the form

$$lhs ::= rhs$$

# Excerpt of the XML grammar

[1]	<i>document</i>	::=	<i>prolog element Misc</i> *
[2]	<i>Char</i>	::=	⟨a Unicode character⟩
[3]	<i>S</i>	::=	('␣'   '\t'   '\n'   '\r') <sup>+</sup>
[4]	<i>NameChar</i>	::=	<i>Letter</i>   <i>Digit</i>   '.'   '-'   '_'   ':'
[5]	<i>Name</i>	::=	( <i>Letter</i>   '-'   ':') ( <i>NameChar</i> ) <sup>*</sup>
[10]	<i>AttValue</i>	::=	'" ([^<&" ]   <i>Reference</i> ) <sup>*</sup> "'
			' ' ([^<&' ]   <i>Reference</i> ) <sup>*</sup> ' '
[14]	<i>CharData</i>	::=	[^<&] <sup>*</sup>
[22]	<i>prolog</i>	::=	<i>XMLDecl?</i> <i>Misc</i> *
[23]	<i>XMLDecl</i>	::=	'<?xml' <i>VersionInfo</i> <i>EncodingDecl?</i> <i>S?</i> '?>'
[24]	<i>VersionInfo</i>	::=	<i>S</i> 'version' <i>Eq</i> (''' <i>VersionNum</i> '''   '"' <i>VersionNum</i> '"')
[25]	<i>Eq</i>	::=	<i>S?</i> '=' <i>S?</i>
[26]	<i>VersionNum</i>	::=	([a-zA-Z0-9_.'-]) <sup>+</sup>
[27]	<i>Misc</i>	::=	<i>S</i>
[39]	<i>element</i>	::=	<i>EmptyElemTag</i>
			<i>S</i> <i>Tag</i> <i>content</i> <i>E</i> <i>Tag</i>
[40]	<i>S</i> <i>Tag</i>	::=	'<' <i>Name</i> ( <i>S</i> <i>Attribute</i> ) <sup>*</sup> <i>S?</i> '>'
[41]	<i>Attribute</i>	::=	<i>Name</i> <i>Eq</i> <i>AttValue</i>
[42]	<i>E</i> <i>Tag</i>	::=	'</' <i>Name</i> <i>S?</i> '>'
[43]	<i>content</i>	::=	( <i>element</i>   <i>CharData</i>   <i>Reference</i> ) <sup>*</sup>
[44]	<i>EmptyElemTag</i>	::=	'<' <i>Name</i> ( <i>S</i> <i>Attribute</i> ) <sup>*</sup> <i>S?</i> '/>'
[67]	<i>Reference</i>	::=	<i>EntityRef</i>
[68]	<i>EntityRef</i>	::=	'&' <i>Name</i> ';' ;
[84]	<i>Letter</i>	::=	[a-zA-Z]
[88]	<i>Digit</i>	::=	[0-9]



## N.B.

- The numbers in  $[\cdot]$  refer to the corresponding productions in the W3C XML Recommendation.

Expression...	... denotes	
$r^*$	$\epsilon, r, rr, rrr, \dots$	zero or more repetitions of $r$
$r^+$	$rr^*$	one or more repetitions of $r$
$r?$	$r \mid \epsilon$	optional $r$
$[abc]$	$a \mid b \mid c$	character class
$[\^abc]$		inverted character class

# Remarks

---

Rule...	...implements this characteristic of XML:
[1]	an XML document contains exactly one <b>root element</b>
[10]	attribute values are enclosed in " or '
[22]	XML documents may include an optional <b>declaration prolog</b>
[14]	characters < and & may <i>not</i> appear literally in element content
[43]	element content may contain character data and entity references as well as nested elements
[68]	entity references may contain arbitrary entity names (other than lt, amp, ...)
⋮	⋮

---

- As usual, the XML grammar may systematically be transformed into a program, an **XML parser**, to be used to check the syntax of XML input.

# Parsing XML

- 1 Starting with the symbol *document*, the parser uses the *lhs ::= rhs* rules to expand symbols, constructing a **parse tree**.
- 2 The leaves of the parse tree are characters which have no further expansion.
- 3 The XML input is **parsed** successfully if it perfectly matches the parse tree's **front** (concatenate the parse tree leaves from left to right<sup>3</sup>).

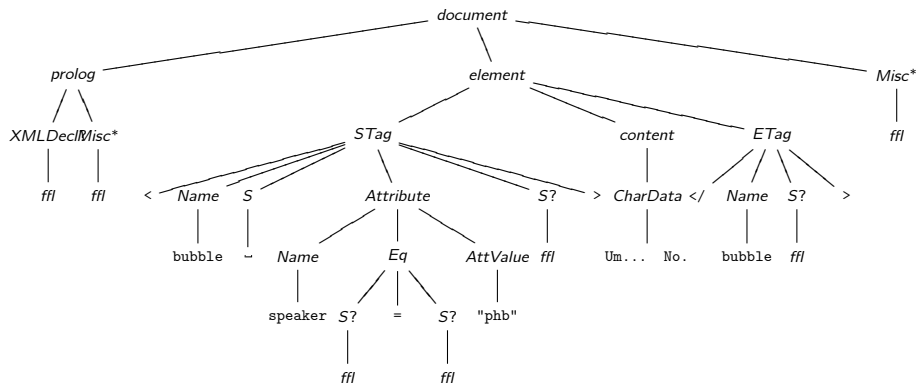
---

<sup>3</sup>**N.B.:**  $x\epsilon y = xy$ .

# Example 1

Parse tree for XML input

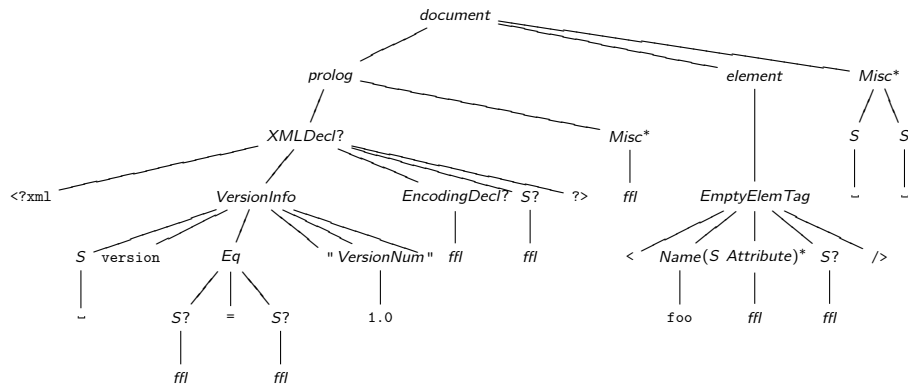
<bubble speaker="phb">Um... No.</bubble> :



## Example 2

Parse tree for the “minimal” XML document

`<?xml version="1.0"?><foo/>`



## Well-formedness #2: Context-dependent Properties

- The XML grammar cannot enforce *all* XML **well-formedness constraints (WFCs)**.
- Some XML WFCs depend on
  - ① what the XML parser has **seen before** in its input, or
  - ② on a **global state**, *e.g.*, the definitions of user-declared entities.
- These WFCs cannot be checked by simply comparing the parse tree front against the XML input (**context-dependent WFCs**).

# Sample WFCs

WFC	Comment
<b>(2) Element Type Match</b>	The <i>Name</i> in an element's end tag must match the element name in the start tag.
<b>(3) Unique Att Spec</b>	No attribute name may appear more than once in the same start tag or empty element tag.
<b>(5) No &lt; in Attribute Values</b>	The replacement text of any entity referred to directly or indirectly in an attribute value (other than &lt;) must not contain a <.
<b>(9) No Recursion</b>	A parsed entity must not contain a recursive reference to itself, either directly or indirectly.

All 10 XML WFCs are given in <http://www.w3.org/TR/REC-xml>.

## How to implement the XML WFC checks?

Devise methods—besides parse tree construction—that an XML parser could use to check the XML WFCs listed above.

Specify *when* during the parsing process you would apply each method.

# The XML Text Declaration `<?xml...?>`

- Remember that a well-formed XML document may start off with an optional header, the **text declaration** (grammar rule [23]).
  - **N.B.** Rule [23] says, *if* the declaration is actually there, no character (whitespace, etc.) may precede the leading `<?xml`.

## The leading `<?xml`

Can you imagine why the XML standard is so rigid with respect to the placement of the `<?xml` leader of the text declaration?

- An XML document whose text declaration carries a *VersionInfo* of `version="1.0"` is required to conform to W3C's XML Recommendation posted on October 6, 2000 (see <http://www.w3.org/TR/REC-xml>).



# XML Documents and Character Encoding

- For a computer, a character like X is nothing but an 8 (16/32) bit **number** whose value is *interpreted* as the character X when needed (e.g., to drive a display).
- Trouble is, a large number of such *number*  $\rightarrow$  *character* mapping tables, the so-called **encodings**, are in parallel use today.
- Due to the huge amount of characters needed by the global computing community today (Latin, Hebrew, Arabic, Greek, Japanese, Chinese ... languages), **conflicting intersections** between encodings are common.

## Example:

0xa4 0xcb 0xe4 0xd3	<u>iso-8859-7</u> $\rightarrow$	☐ Λ δ Σ
0xa4 0xcb 0xe4 0xd3	<u>iso-8859-15</u> $\rightarrow$	€ Ë ä Ó

# Unicode

- The **Unicode** (<http://www.unicode.org/>) Initiative aims to define a new encoding that tries to embrace all character needs.
- The Unicode encoding contains characters of “all” languages of the world, plus scientific, mathematical, technical, box drawing, ... symbols (see <http://www.unicode.org/charts/>).
- Range of the Unicode encoding: 0x0000–0x10FFFF (16 × 65536 characters).
  - ▶ Codes that fit into the first 16 bits (denoted U+0000–U+FFFF) have been assigned to encode the most widely used languages and their characters (**Basic Multilingual Plane, BMP**).
  - ▶ Codes U+0000–U+007F have been assigned to match the 7-bit ASCII encoding which is pervasive today.

# UTF-32

- Current CPUs operate most efficiently on **32-bit words (16-bit words, 8-bit bytes)**.
- Unicode thus developed **Unicode Transformation Formats (UTF)** which define how a Unicode character code between U+0000–U+10FFFF is to be mapped into a 32-bit word (16-bit words, 8-bit bytes).

## UTF-32 (map a Unicode character into a 32-bit word)

- 1 Map any Unicode character in the range U+0000–U+10FFFF to the corresponding 32-bit value 0x00000000–0x0010FFFF.
- 2 **N.B.** For each Unicode character encoded in UTF-32 we waste at least 11 zero bits.

# UTF-16

... map a Unicode character into one or two 16-bit words

- 1 Apply the following mapping scheme:

Unicode range	Word sequence
U+000000–U+00FFFF	□□□□□□□□□□□□□□□□
U+010000–U+10FFFF	110110□□□□□□□□□□ 110111□□□□□□□□□□

- 2 For the range U+000000–U+00FFFF, simply fill the □ positions with the 16 bit of the character code.  
(Code ranges U+D800–U+DBFF and U+DC00–U+DFFF are unassigned!)
- 3 For the U+010000–U+10FFFF range, subtract 0x010000 from the character code and fill the □ positions using the resulting 20-bit value.

## Example

Unicode character U+012345 ( $0x012345 - 0x010000 = 0x02345$ ):

UTF-16: 11011000000001000 1101111101000101

# UTF-8

**N.B.** UTF-16 is designed to facilitate efficient and robust decoding:

- If we see a leading 11011 bit pattern in a 16-bit word, we know it is the first **or** second word in a UTF-16 multi-word sequence.
- The sixth bit of the word then tells us if we actually look at the first or second word.

**UTF-8** (map a Unicode character into a sequence of 8-bit bytes)

- UTF-8 is of special importance because
  - (a) a stream of 8 bit bytes (*octets*) is what flows over an IP network connection,
  - (b) text-processing software today is built to deal with 8 bit character encodings (*iso-8859-x*, *ASCII*, *etc.*).

# UTF-8 encoding

- 1 Apply the following mapping scheme:

Unicode range	Byte sequence
U+000000–U+00007F	0□□□□□□
U+000080–U+0007FF	110□□□□□ 10□□□□□□
U+000800–U+00FFFF	1110□□□□ 10□□□□□□ 10□□□□□□
U+010000–U+10FFFF	11110□□□ 10□□□□□□ 10□□□□□□ 10□□□□□□

- 2 The spare bits (□) are filled with the bits of the character code to be represented (rightmost □ is least significant bit, pad to the left with 0-bits).

## Examples:

- ▶ Unicode character U+00A9 (© sign):  
UTF-8: 11000010 10101001 (0xC2 0xA9)
- ▶ Unicode character U+2260 (math relation symbol ≠):  
UTF-8: 11100010 10001001 10100000 (0xE2 0x89 0xA0)

# Advantages of UTF-8 encoding

**N.B.** UTF-8 enjoys a number of highly desirable properties:

- For a UTF-8 multi-byte sequence, the **length of the sequence** is equal to the number of leading 1-bits (in the first byte), *e.g.*:

11100010 10001001 10100000

(Only single-byte UTF-8 encodings have a leading 0-bit.)

- **Character boundaries** are simple to detect (even when placed at some arbitrary position in a UTF-8 byte stream).
- UTF-8 encoding does not affect (binary) sort order.
- Text processing software which was originally developed to work with the pervasive 7-bit ASCII encoding remains functional.  
This is especially true for the C programming language and its string (`char []`) representation.

## C and UTF-8

Can you explain the last points made?

# XML and Unicode

- A conforming XML parser is required to correctly process UTF-8 and UTF-16 encoded documents (The W3C XML Recommendation predates the UTF-32 definition).
- Documents that use a different encoding *must* announce so using the XML text declaration, *e.g.*

```
<?xml encoding="iso-8859-15"?>
or <?xml encoding="utf-32"?>
```

- Otherwise, an XML parser is encouraged to **guess** the encoding while reading the very first bytes of the input XML document:

Head of doc (bytes)	Encoding guess
0x00 0x3C 0x00 0x3F	UTF-16 ( <i>big-endian</i> )
0x3C 0x00 0x3F 0x00	UTF-16 ( <i>little-endian</i> )
0x3C 0x3F 0x78 0x6D	UTF-8 (or ASCII, iso-8859-?: <i>erroneous</i> )

(Notice: < = U+003C, ? = U+003F, x = U+0078, m = U+006D)

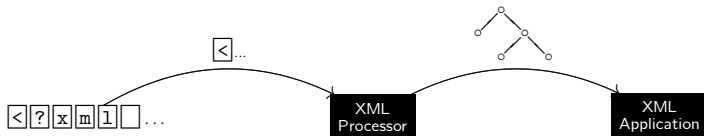


# The XML Processing Model

- On the **physical** side, XML defines nothing but a **flat text format**, *i.e.*, defines a set of (UTF-8/16) character sequences being well-formed XML.
- Applications that want to analyse and transform XML data in any meaningful manner will find processing flat character sequences hard and inefficient.
- The **nesting** of XML elements and attributes, however, defines a **logical tree-like structure**.

# XML Processors

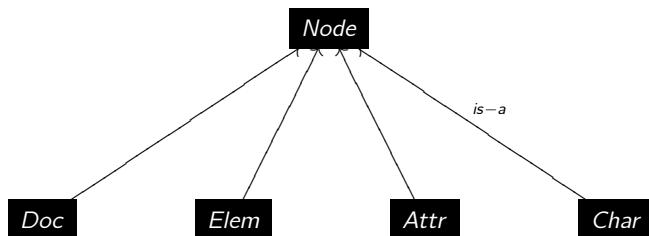
- Virtually all **XML applications operate on the logical tree view** which is provided to them through an **XML Processor** (*i.e.*, the XML parser):



- XML processors are widely available (*e.g.*, Apache's Xerces see <http://xml.apache.org/>).
- How is the XML processor supposed to **communicate the XML tree structure** to the application ... ?

# The XML Information Set

- Once an XML processor has checked its XML input document to be well-formed, it offers its application a set of document **properties** (functions).
- The application calls property functions and thus explores the input XML tree as needed.
- An XML document tree is built of different types of **node objects**:



- The set of properties of all document nodes is the document's **Information Set** (see <http://www.w3.org/TR/xml-infoset/>).

# Node properties

Node Object Type	Property		Comment
<i>Doc</i>	<i>children</i>	$:: Doc \rightarrow Elem$	root element
	<i>base-uri</i>	$:: Doc \rightarrow String$	
	<i>version</i>	$:: Doc \rightarrow String$	<code>&lt;?xml version="1.0"?&gt;</code>
<i>Elem</i>	<i>localname</i>	$:: Elem \rightarrow String$	
	<i>children</i>	$:: Elem \rightarrow (Node)$	$*^1$
	<i>attributes</i>	$:: Elem \rightarrow (Attr)$	
	<i>parent</i>	$:: Elem \rightarrow Node$	$*^2$
<i>Attr</i>	<i>localname</i>	$:: Attr \rightarrow String$	
	<i>value</i>	$:: Attr \rightarrow String$	
	<i>owner</i>	$:: Attr \rightarrow Elem$	
<i>Char</i>	<i>code</i>	$:: Char \rightarrow Unicode$	a single character
	<i>parent</i>	$:: Char \rightarrow Elem$	

- Read symbol  $::$  as “has type”.
- For any node type  $\tau$ ,  $(\tau)$  denotes an ordered **sequence** of type  $\tau$ .

✎ Make sense of the types of the *Elem* properties *children* ( $*^1$ ) and *parent* ( $*^2$ )!

# Information set of a sample document

1 \_\_\_\_\_ Document  $\delta_0$  (weather forecast) \_\_\_\_\_

```

2 <?xml version="1.0"?>
3 <forecast date="Thu, May 16">
4   <condition>sunny</condition>
5   <temperature unit="Celsius">23</temperature>
6 </forecast>

```

<i>children</i> ( $\delta_0$ )	=	$\varepsilon_1$	<i>code</i> ( $\gamma_4$ )	=	U+0073 's'
<i>base-uri</i> ( $\delta_0$ )	=	"file:/..."	<i>parent</i> ( $\gamma_4$ )	=	$\varepsilon_3$
					:
<i>version</i> ( $\delta_0$ )	=	"1.0"			:
<i>localname</i> ( $\varepsilon_1$ )	=	"forecast"	<i>code</i> ( $\gamma_8$ )	=	U+0079 'y'
<i>children</i> ( $\varepsilon_1$ )	=	( $\varepsilon_3, \varepsilon_9$ )	<i>parent</i> ( $\gamma_8$ )	=	$\varepsilon_3$
<i>attributes</i> ( $\varepsilon_1$ )	=	( $\alpha_2$ )	<i>localname</i> ( $\varepsilon_9$ )	=	"temperature"
<i>parent</i> ( $\varepsilon_1$ )	=	$\delta_0$	<i>children</i> ( $\varepsilon_9$ )	=	( $\gamma_{11}, \gamma_{12}$ )
<i>localname</i> ( $\alpha_2$ )	=	"date"	<i>attributes</i> ( $\varepsilon_9$ )	=	( $\alpha_{10}$ )
<i>value</i> ( $\alpha_2$ )	=	"Thu, May 16"	<i>parent</i> ( $\varepsilon_9$ )	=	$\varepsilon_1$
					:
<i>localname</i> ( $\varepsilon_3$ )	=	"condition"			:
<i>children</i> ( $\varepsilon_3$ )	=	( $\gamma_4, \gamma_5, \gamma_6, \gamma_7, \gamma_8$ )			
<i>attributes</i> ( $\varepsilon_3$ )	=	()			
<i>parent</i> ( $\varepsilon_3$ )	=	$\varepsilon_1$			

**N.B.** Node objects of type *Doc*, *Elem*, *Attr*, *Char* are denoted by  $\delta_i$ ,  $\varepsilon_i$ ,  $\alpha_i$ ,  $\gamma_i$ , respectively (subscript  $i$  makes object identifiers unique).

## Working with the Information Set

- The W3C has introduced the XML Information Set to aid the specification of further XML standards.
- We can nevertheless use it to write simple “programs” that explore the XML tree structure. The resulting code looks fairly similar to code we would program using the DOM (Document Object Model, see next chapter).

**Example:** Compute the list of **sibling** *Elem* nodes of given *Elem*  $\epsilon$  (including  $\epsilon$ ):

*siblings* ( $\epsilon$ ) :: *Elem*  $\rightarrow$  (*Elem*)

*Node*  $\nu$ ;

$\nu \leftarrow \text{parent}(\epsilon)$ ;

if  $\nu = \delta_{\square}$  then

    //  $\nu$  is the Doc node, i.e.,  $\epsilon$  is the root element  
    return ( $\epsilon$ );

else

    return *children* ( $\nu$ );

## Another Example

Return the text **content** of a given *Doc*  $\delta$  (the sequence of all Unicode characters  $\delta$  contains):

$content(\delta) :: Doc \rightarrow (Unicode)$

return collect ((children ( $\delta$ )));

$collect(\nu s) :: (Node) \rightarrow (Unicode)$

Node  $\nu$ ;

(Unicode)  $\gamma s$ ;

$\gamma s \leftarrow ()$ ;

foreach  $\nu \in \nu s$  do

    if  $\nu = \gamma_{\square}$  then

        // we have found a Char node ...

$\gamma s \leftarrow \gamma s + (code(\nu))$ ;

    else

        // otherwise  $\nu$  must be an Elem  
node

$\gamma s \leftarrow \gamma s + collect(children(\nu))$ ;

return  $\gamma s$ ;

- Example run:  $content(\delta_0) = ('s', 'u', 'n', 'n', 'y', '2', '3')$ .

## “Querying” using the Information Set

- Having the XML Information Set in hand, we can analyse a given XML document in arbitrary ways, *e.g.*
  - ① In a given document (comic strip), find **all** *Elem* nodes with local name *bubble* owning an *Attr* node with local name *speaker* and value "Dilbert".
  - ② List **all** scene *Elem* nodes **containing** a bubble spoken by "Dogbert" (*Attr* *speaker*).
  - ③ Starting in panel number 2 (no *Attr*), find **all** bubbles **following** those spoken by "Alice" (*Attr* *speaker*).
- Queries like these are quite common in XML applications. An XML standard exists (**XPath**) which allows to specify such **document path traversals** in a declarative manner:
  - ① `//bubble[./@speaker = "Dilbert"]`
  - ② `//bubble[@speaker = "Dogbert"]/../..`
  - ③ `//panel[@no = "2"]//bubble[@speaker = "Alice"]/following::bubble`



## More XML node types ...

- The XML standard defines a number of additional node types that may occur in well-formed documents (and thus in their XML Information Set).
- **CDATA** nodes (embed *unparsed* non-binary character data)

### CDATA

```
1 <source>
2   <![CDATA[ May use <, >, and & and
3     anything else freely here ]]>
4 </source>
```

- **Comment** nodes (place comments in XML documents)

### Comment

```
1 <proof>
2   <!-- Beware! This has not been properly
3     checked yet... -->
4   ...
5 </proof>
```

## ... more XML node types

- **PI** nodes (embed *processing instructions* in XML documents)

**PI**

```
1 <em>
2   <b>Result:</b>
3   <?php sql ("SELECT * FROM ...") ...?>
4 </em>
```

- For a complete list of node types see the W3C XML Recommendation (<http://www.w3.org/TR/REC-xml>).

## Part IV

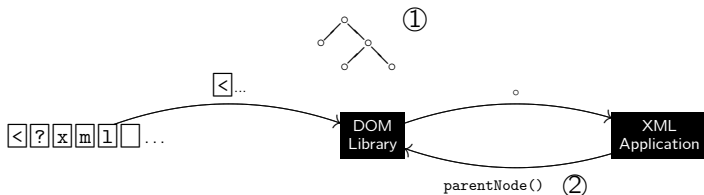
### DOM—Document Object Model

# Outline of this part

- 9 DOM Level 1 (Core)
- 10 DOM Example Code
- 11 DOM—A Memory Bottleneck

# DOM—Document Object Model

- With **DOM**, W3C has defined a **language-** and **platform-neutral** view of XML documents much like the XML Information Set.
- DOM APIs exist for a wide variety of—predominantly object-oriented—programming languages (Java, C++, C, Perl, Python, ...).
- The DOM design rests on two major concepts:
  - ① An **XML Processor** offering a DOM interface parses the XML input document, and constructs the **complete XML document tree** (in-memory).
  - ② The **XML application** then issues DOM library calls to **explore** and **manipulate** the XML document, or **generate** new XML documents.



- The DOM approach has some obvious advantages:
  - ▶ Once DOM has build the XML tree structure, (tricky) issues of XML grammar and syntactical specifics are void.
  - ▶ **Constructing** an XML document using the DOM instead of serializing an XML document manually (using some variation of `print`), ensures **correctness** and **well-formedness**.
    - ★ No missing/non-matching tags, attributes never owned by attributes,  
...
  - ▶ The DOM can simplify document **manipulation** considerably.
    - ★ Consider transforming

#### Weather forecast (English)

```
1 <?xml version="1.0"?>
2 <forecast date="Thu, May 16">
3   <condition>sunny</condition>
4   <temperature unit="Celsius">23</temperature>
5 </forecast>
```

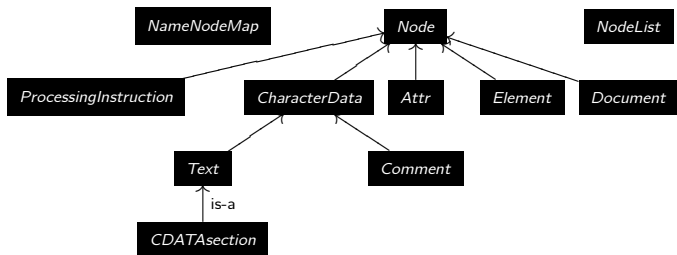
into

#### Weather forecast (German)

```
1 <?xml version="1.0"?>
2 <vorhersage datum="Do, 16. Mai">
3   <wetterlage>sonnig</wetterlage>
4   <temperatur skala="Celsius">23</temperatur>
5 </vorhersage>
```

# DOM Level 1 (Core)

- To operate on XML document trees, DOM Level 1<sup>4</sup> defines an inheritance hierarchy of node objects—and methods to operate on these—as follows (excerpt):



- Character strings (DOM type *DOMString*) are defined to be encoded using UTF-16 (e.g., Java DOM represents type *DOMString* using its `String` type).

<sup>4</sup><http://www.w3.org/TR/REC-DOM-Level-1/>

- (The complete DOM interface is too large to list here.) Some methods of the principal DOM types *Node* and *Document*:

DOM Type	Method	Comment
<i>Node</i>	<i>nodeName</i> :: <i>DOMString</i>	redefined in subclasses, e.g., tag name for <i>Element</i> , "#text" for <i>Text</i> nodes, ...
	<i>parentNode</i> :: <i>Node</i>	
	<i>firstChild</i> :: <i>Node</i>	leftmost child node
	<i>nextSibling</i> :: <i>Node</i>	returns NULL for root element or last child or attributes
	<i>childNodes</i> :: <i>NodeList</i>	see below
	<i>attributes</i> :: <i>NameNodeMap</i>	see below
	<i>ownerDocument</i> :: <i>Document</i>	
	<i>replaceChild</i> :: <i>Node</i>	replace new for old node, returns old
<i>Document</i>	<i>createElement</i> :: <i>Element</i>	creates element with given tag name
	<i>createComment</i> :: <i>Comment</i>	creates comment with given content
	<i>getElementsByTagName</i> :: <i>NodeList</i>	list of all <i>Elem</i> nodes in document order

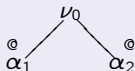


## Some DOM Details

- Creating an element (or attribute) using *createElement* (*createAttribute*) does *not* wire the new node with the XML tree structure yet.  
Call *insertBefore*, *replaceChild*, ... to wire a node at an explicit position.
- DOM type *NodeList* (node sequence) makes up for the lack of collection datatypes in most programming languages.  
Methods: *length*, *item* (node at specific index position).
- DOM type *NamedNodeMap* represents an *association table* (nodes may be accessed by name).

### Example:

bubble



Apply method *attributes* to  
Element object  $\nu_0$  to obtain  
this *NamedNodeMap*:

speaker      to

name		node
"speaker"	$\mapsto$	$\alpha_1$
"to"	$\mapsto$	$\alpha_2$

Methods: *getNamedItem*, *setNamedItem*, ...

# DOM Example Code

- The following slide shows C++ code written against the Xerces C++ DOM API<sup>5</sup>.
- The code implements a variant of the  $content :: Doc \rightarrow (Char)$ :
  - ▶ Function `collect ()` decodes the UTF-16 text content returned by the DOM and prints it to standard output directly (`transcode ()`, `cout`).

## N.B.

- A W3C DOM node type named  $\tau$  is referred to as `DOM_τ` in the Xerces C++ DOM API.
- A W3C DOM property named *foo* is—in line with common object-oriented programming practice—called `getFoo()` here.

---

<sup>5</sup><http://xml.apache.org/>

# Example: C++/DOM Code

```

1 // Xerces C++ DOM API support
2 #include <dom/DOM.hpp>
3 #include <parsers/DOMParser.hpp>
4
5 void collect (DOM_NodeList ns)
6 {
7     DOM_Node n;
8
9     for ( unsigned long i = 0;
10          i < ns.getLength ();
11          i++){
12         n = ns.item (i);
13
14         switch (n.getNodeType ()) {
15             case DOM_Node::TEXT_NODE:
16                 cout << n.getNodeValue ().transcode ();
17                 break;
18             case DOM_Node::ELEMENT_NODE:
19                 collect (n.getChildNodes ());
20         }
21     }
22 }

```

```

23
24 void content (DOM_Document d)
25 {
26     collect (d.getChildNodes ());
27 }
28
29 int main (void)
30 {
31     XMLPlatformUtils::Initialize ();
32
33     DOMParser parser;
34     DOM_Document doc;
35
36     parser.parse ("foo.xml");
37     doc = parser.getDocument ();
38
39     content (doc);
40
41     return 0;
42 }

```

**Now:** Find all occurrences of Dogbert speaking (attribute speaker of element bubble) ...

## dogbert.cc (1)

```
1 // Xerces C++ DOM API support
2 #include <dom/DOM.hpp>
3 #include <parsers/DOMParser.hpp>
4
5 void dogbert (DOM_Document d)
6 {
7     DOM_NodeList      bubbles;
8     DOM_Node          bubble, speaker;
9     DOM_NamedNodeMap  attrs;
10
11     bubbles = d.getElementsByTagName ("bubble");
12
13     for (unsigned long i = 0; i < bubbles.getLength (); i++) {
14         bubble = bubbles.item (i);
15
16         attrs = bubble.getAttributes ();
17         if (attrs != 0)
18             if ((speaker = attrs.getNamedItem ("speaker")) != 0)
19                 if (speaker.getNodeValue ().
20                     compareString (DOMString ("Dogbert")) == 0)
21                     cout << "Found Dogbert speaking." << endl;
22     }
23 }
```

## dogbert.cc (2)

```
24
25 int main (void)
26 {
27     XMLPlatformUtils::Initialize ();
28
29     DOMParser parser;
30     DOM_Document doc;
31
32     parser.parse ("foo.xml");
33     doc = parser.getDocument ();
34
35     dogbert (doc);
36
37     return 0;
38 }
```

# DOM—A Memory Bottleneck

- The two-step processing approach (① parse and construct XML tree, ② respond to DOM property function calls) enables the DOM to be “**random access**”:

The XML application may explore and update any portion of the XML tree at any time.

- The inherent memory hunger of the DOM may lead to
  - ① heavy **swapping** activity  
(partly due to unpredictable memory access patterns, `madvise()` less helpful)  
or
  - ② even “out-of-memory” failures.  
(The application has to be extremely careful with its own memory management, the very least.)

# Numbers





## DOM and random node access

Even if the application touches a single element node only, the DOM API has to maintain a data structure that represents the **whole XML input document** (all sizes in kB):<sup>6</sup>

XML size	DOM process size DSIZ	$\frac{\text{DSIZ}}{\text{XML size}}$	Comment
7480	47476	6.3	(Shakespeare's works) many elements containing small text fragments
113904	552104	4.8	(Synthetic eBay data) elements containing relatively large text fragments

<sup>6</sup>The random access nature of the DOM makes it hard to provide a truly “lazy” API implementation.

## To remedy the memory hunger of DOM-based processing . . .

- Try to **preprocess** (*i.e.*, filter) the input XML document to reduce its overall size.
  - ▶ Use an XPath/XSLT processor to preselect *interesting* document regions,
  - ▶  *no updates* to the input XML document are possible then,
  - ▶  make sure the XPath/XSLT processor is *not* implemented on top of the DOM.

Or

- Use a **completely different** approach to XML processing (→ **SAX**).



## Part V

### SAX—Simple API for XML

# Outline of this part

- 12 SAX Events
- 13 SAX Callbacks
- 14 SAX and the XML Tree Structure
- 15 SAX and Path Queries
  - Path Query Evaluation
- 16 Final Remarks on SAX

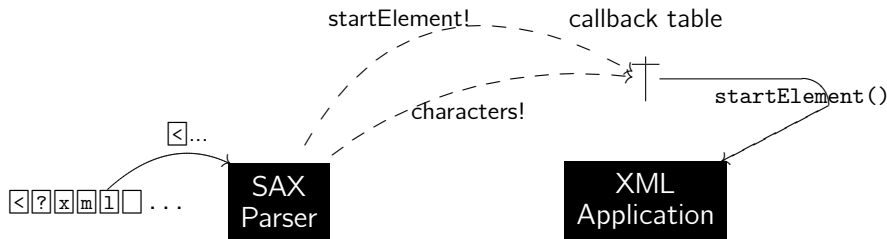
# SAX—Simple API for XML


- **SAX<sup>7</sup> (Simple API for XML)** is, unlike DOM, *not* a W3C standard, but has been developed jointly by members of the XML-DEV mailing list (*ca.* 1998).
- SAX processors use **constant space**, regardless of the XML input document size.
  - ▶ Communication between the SAX processor and the backend XML application does *not* involve an intermediate tree data structure.
  - ▶ Instead, the **SAX parser sends events** to the application whenever a certain piece of XML text has been recognized (*i.e.*, parsed).
  - ▶ The **backend acts on/ignores events** by populating a **callback function table**.

---

<sup>7</sup><http://www.saxproject.org/>

# Sketch of SAX's mode of operations



- A SAX processor reads its input document **sequentially** and **once** only.
- No memory of what the parser has seen so far is retained while parsing. As soon as a  *significant bit of XML text* has been recognized, an **event** is sent.
- The application is able to act on events **in parallel** with the parsing progress.

# SAX Events

- To meet the constant memory space requirement, SAX reports **fine-grained parsing events** for a document:

Event	...reported when seen	Parameters sent
<i>startDocument</i>	<code>&lt;?xml...?&gt;</code> <sup>8</sup>	
<i>endDocument</i>	<code>&lt;EOF&gt;</code>	
<i>startElement</i>	<code>&lt;t a<sub>1</sub>=v<sub>1</sub> ... a<sub>n</sub>=v<sub>n</sub>&gt;</code>	<i>t</i> , ( <i>a</i> <sub>1</sub> , <i>v</i> <sub>1</sub> ), ..., ( <i>a</i> <sub><i>n</i></sub> , <i>v</i> <sub><i>n</i></sub> )
<i>endElement</i>	<code>&lt;/t&gt;</code>	<i>t</i>
<i>characters</i>	<i>text content</i>	Unicode buffer ptr, length
<i>comment</i>	<code>&lt;!--c--&gt;</code>	<i>c</i>
<i>processingInstruction</i>	<code>&lt;?t pi?&gt;</code>	<i>t</i> , <i>pi</i>
	⋮	

<sup>8</sup>**N.B.:** Event *startDocument* is sent even if the optional XML text declaration should be missing.

## dilbert.xml

```

1 <?xml encoding="utf-8"?> *1
2 <bubbles> *2
3   <!-- Dilbert looks stunned --> *3
4   <bubble speaker="phb" to="dilbert"> *4
5     Tell the truth, but do it in your usual engineering way
6     so that no one understands you. *5
7   </bubble> *6
8 </bubbles> *7 *8

```

Event <sup>9</sup> <sup>10</sup>		Parameters sent
* <sub>1</sub>	<i>startDocument</i>	
* <sub>2</sub>	<i>startElement</i>	<i>t</i> = "bubbles"
* <sub>3</sub>	<i>comment</i>	<i>c</i> = "_Dilbert looks stunned_"
* <sub>4</sub>	<i>startElement</i>	<i>t</i> = "bubble", ("speaker","phb"), ("to","dilbert")
* <sub>5</sub>	<i>characters</i>	<i>buf</i> = "Tell the...understands you.", <i>len</i> = 99
* <sub>6</sub>	<i>endElement</i>	<i>t</i> = "bubble"
* <sub>7</sub>	<i>endElement</i>	<i>t</i> = "bubbles"
* <sub>8</sub>	<i>endDocument</i>	

<sup>9</sup>Events are reported in **document reading order** \*<sub>1</sub>, \*<sub>2</sub>, ..., \*<sub>8</sub>.

<sup>10</sup>**N.B.:** Some events suppressed (white space).

# SAX Callbacks

- To provide an efficient and tight **coupling** between the SAX **frontend** and the application **backend**, the SAX API employs **function callbacks**.<sup>11</sup>
  - Before parsing starts, the application **registers function references** in a table in which each event has its own slot:

Event	Callback		Event	Callback
⋮			⋮	
<i>startElement</i>	?	— →	<i>startElement</i>	<i>startElement ()</i>
<i>endElement</i>	?	<i>SAXregister(startElement,</i> <i>startElement ())</i>	<i>endElement</i>	<i>endElement ()</i>
⋮		<i>SAXregister(endElement,</i>	⋮	
⋮		<i>endElement ())</i>	⋮	

- The application alone decides on the implementation of the functions it registers with the SAX parser.
- Reporting an event**  $\star_i$  then amounts to call the function (with parameters) registered in the appropriate table slot.

<sup>11</sup>Much like in event-based GUI libraries.



## Java SAX API

In Java, populating the callback table is done via implementation of the SAX `ContentHandler` interface: a `ContentHandler` object represents the callback table, its methods (e.g., `public void endDocument ()`) represent the table slots.

**Example:** Reimplement *content.cc* shown earlier for DOM (find all XML text nodes and print their content) using SAX (pseudo code):

*content (File f)*

```
// register the callback,
// we ignore all other events
SAXregister (characters, printText);
SAXparse (f);
return;
```

*printText ((Unicode) buf, Int len)*

```
Int i;
foreach i ∈ 1 . . . len do
    | print (buf[i]);
return;
```



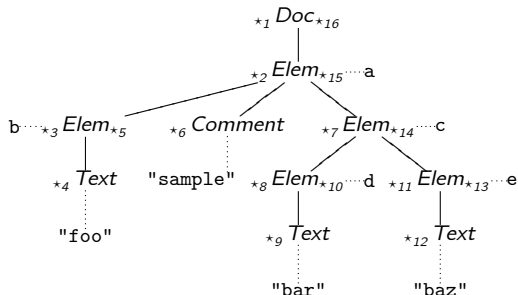
# SAX and the XML Tree Structure

- Looking closer, the **order** of SAX events reported for a document is determined by a **preorder traversal** of its document tree<sup>12</sup>:

Sample XML document

```

1  *1
2  <a>*2
3    <b>*3 foo*4 </b>*5
4    <!--sample-->*6
5    <c>*7
6      <d>*8 bar*9 </d>*10
7      <e>*11 baz*12 </e>*13
8    </c>*14
9  </a>*15 *16
  
```



**N.B.:** An *Elem* [*Doc*] node is associated with two SAX events, namely *startElement* and *endElement* [*startDocument*, *endDocument*].

<sup>12</sup>Sequences of sibling *Char* nodes have been collapsed into a single *Text* node.

## Challenge

- This **left-first depth-first** order of SAX events is well-defined, but appears to make it hard to answer certain queries about an XML document tree.

 Collect all direct children nodes of an *Elem* node.

In the example on the previous slide, suppose your application has just received the *startElement*(*t* = "a") event  $\star_2$  (i.e., the parser has just parsed the opening element tag <a>).

With the remaining events  $\star_3 \dots \star_{16}$  still to arrive, can your code detect all the immediate children of *Elem* node a (i.e., *Elem* nodes b and c as well as the *Comment* node)?

The previous question can be answered more generally:

*SAX events are sufficient to **rebuild the complete XML document tree** inside the application. (Even if we most likely don't want to.)*

## SAX-based tree rebuilding strategy (sketch):

- ① *[startDocument]*  
Initialize a **stack** *S* of **node IDs** (e.g.  $\in \mathbb{Z}$ ). **Push** first ID for this node.
- ② *[startElement]*  
Assign a **new ID** for this node. **Push** the ID onto *S*.<sup>13</sup>
- ③ *[characters, comment, ...]*  
Simply assign a new node ID.
- ④ *[endElement, endDocument]*  
**Pop** *S* (no new node created).



Invariant: The **top** of *S* holds the identifier of the current **parent node**.

<sup>13</sup>In callbacks ② and ③ we might wish to store further node details in a table or similar summary data structure.

# SAX Callbacks

**SAX callbacks** to rebuild XML document tree:

- We maintain a summary table of the form

<u>ID</u>	<u>  </u>	<u>NodeType</u>	<u> </u>	<u>Tag</u>	<u> </u>	<u>Content</u>	<u> </u>	<u>ParentID</u>
-----------	-----------	-----------------	----------	------------	----------	----------------	----------	-----------------

- *insert* (*id*, *type*, *t*, *c*, *pid*) inserts a row into this table.
- Maintain stack *S* of node IDs, with operations *push(id)*, *pop()*, *top()*, and *empty()*.

*startDocument* ()

```

id ← 0;
S.empty();
insert (id, Doc, □, □, □);
S.push(id);
return ;

```

*endDocument* ()

```

S.pop();
return ;

```

# SAX Callbacks

*startElement* ( $t, (a_1, v_1), \dots$ )

```

 $id \leftarrow id + 1$ ;
insert ( $id, Elem, t, \square, S.top()$ );
 $S.push(id)$ ;
return ;

```

*characters* ( $buf, len$ )

```

 $id \leftarrow id + 1$ ;
insert ( $id, Text, \square, buf[1 \dots len], S.top()$ );
return ;

```

*endElement* ( $t$ )

```

 $S.pop()$ ;
return ;

```

*comment* ( $c$ )

```

 $id \leftarrow id + 1$ ;
insert ( $id, Comment, \square, c, S.top()$ );
return ;

```

- Run against the example given above, we end up with the following summary table:

ID	NodeType	Tag	Content	ParentID
0	<i>Doc</i>	□	□	□
1	<i>Elem</i>	a	□	0
2	<i>Elem</i>	b	□	1
3	<i>Text</i>	□	"foo"	2
4	<i>Comment</i>	□	"sample"	1
5	<i>Elem</i>	c	□	1
6	<i>Elem</i>	d	□	5
7	<i>Text</i>	□	"bar"	6
8	<i>Elem</i>	e	□	5
9	<i>Text</i>	□	"baz"	8

- Since XML defines tree structures only, the ParentID column is all we need to recover the complete node hierarchy of the input document.

### Walking the XML node hierarchy?

Explain how we may use the summary table to find the (a) *children*, (b) *siblings*, (c) *ancestors*, (d) *descendants* of a given node (identified by its ID).


# SAX and Path Queries

- **Path queries** are *the* core language feature of virtually all XML query languages proposed so far (e.g., XPath, XQuery, XSLT, ...).
- To keep things simple for now, let a path query take one of two forms (the  $t_i$  represent *tag names*):

$//t_1/t_2/\dots/t_n$       or       $//t_1/t_2/\dots/t_{n-1}/\text{text}()$

## Semantics:

A **path query selects a set of *Elem* nodes** [with `text()`: *Text* nodes] from a given XML document:

- 1 The selected nodes have tag name  $t_n$  [are *Text* nodes].
- 2 Selected nodes have a parent *Elem* node with tag name  $t_{n-1}$ , which in turn has a parent node with tag name  $t_{n-2}$ , which ... has a parent node with tag name  $t_1$  ( not necessarily the document root element).

## Examples:

- 1 Retrieve all **scene nodes** from a DilbertML document:  
`//panels/panel/scene`
- 2 Retrieve all character **names** from a DilbertML document:  
`//strip/characters/character/text()`

## Path Query Evaluation

- The summary table discussed in the previous section obviously includes all necessary information to evaluate both types of path queries.

### Evaluating path queries using the materialized tree structure.

Sketch a summary table based algorithm that can evaluate a path query. (Use `//a/c/d/text()` as an example.)

- Note that, although based on SAX, such a path query evaluator would probably **consume as much memory as a DOM-based implementation**.



## SAX-based path query evaluation (sketch):

- 1 Preparation:
  - ▶ Represent path query  $//t_1/t_2/\dots/t_{n-1}/\text{text}()$  via the *step array*  $\text{path}[0] = t_1, \text{path}[1] = t_2, \dots, \text{path}[n-1] = \text{text}()$ .
  - ▶ Maintain an array index  $i = 0 \dots n$ , the *current step* in the path.
  - ▶ Maintain a stack  $S$  of index positions.
- 2 `[startDocument]`  
Empty stack  $S$ . We start with the first step.
- 3 `[startElement]`  
If the current step's tag name  $\text{path}[i]$  and the reported tag name match, proceed to next step. Otherwise make a failure transition<sup>14</sup>. Remember how far we have come already: **push** the current step  $i$  onto  $S$ .
- 4 `[endElement]`  
The parser ascended to a parent element. Resume path traversal from where we have left earlier: **pop** old  $i$  from  $S$ .
- 5 `[characters]`  
If the current step  $\text{path}[i] = \text{text}()$  we have found a match. Otherwise do nothing.

<sup>14</sup>This “Knuth-Morris-Pratt failure function” *fail[]* is to be explained in the tutorial.

# SAX-based path query evaluation (given step array $path[0 \dots n - 1]$ ):

$startElement(t, (a_1, v_1), \dots)$

$S.push(i);$

while true do

    if  $path[i] = t$  then

$i \leftarrow i + 1;$

        if  $i = n$  then

            ★Match★;

$i \leftarrow fail[i];$

        break;

    if  $i = 0$  then

        break;

$i \leftarrow fail[i];$

return ;

$startDocument()$

$i \leftarrow 0;$

$S.empty();$

return ;

$characters(buf, len)$

$endElement(t)$

$i \leftarrow S.pop();$

return ;

if  $path[i] = text()$

then

    ★Match★;

return ;

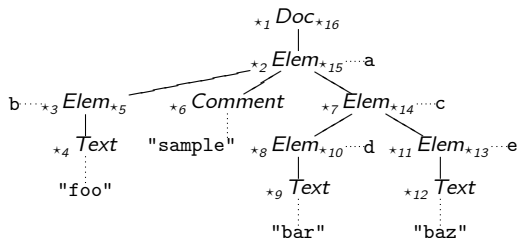
**N.B.:**

These SAX callbacks

- 1 evaluate a path query while we receive events (**stream processing**), and
- 2 operate **without building a summary data structure** and can thus evaluate path queries on documents of **arbitrary size**.

## Tracing SAX Events ...

Is there a bound on the stack depth we need during the path query execution?



Path Query (length  $n = 4$ ): `//a/c/d/text()`  
`path[0]=a, path[1]=c, path[2]=d, path[3]=text()`

\*<sub>1</sub> `startDocument()`  
 $i = 0$   
 $S = \square\square\square\square$

\*<sub>2</sub> `startElement(t = a)`  
 $i = 1$   
 $S = 0\square\square\square$

\*<sub>3</sub> `startElement(t = b)`  
 $i = 0$   
 $S = 10\square\square$

\*<sub>4</sub> `characters("foo", 3)`  
 $i = 0$   
 $S = 10\square\square$

\*<sub>5</sub> `endElement(b)`  
 $i = 1$   
 $S = 0\square\square\square$

\*<sub>6</sub> `comment("sample")`  
 $i = 1$   
 $S = 0\square\square\square$

\*<sub>7</sub> `startElement(c)`  
 $i = 2$   
 $S = 10\square\square$

\*<sub>8</sub> `startElement(d)`  
 $i = 3$   
 $S = 210\square$

\*<sub>9</sub> `characters("bar", 3)`  
 $i = 3$   
 $S = 210\square$  ★Match★

\*<sub>10</sub> `endElement(d)`  
 $i = 2$   
 $S = 10\square\square$

\*<sub>11</sub> `startElement(e)`  
 $i = 0$   
 $S = 210\square$

\*<sub>12</sub> `characters("baz", 3)`  
 $i = 0$   
 $S = 210\square$

\*<sub>13</sub> `endElement(e)`  
 $i = 2$   
 $S = 10\square\square$

\*<sub>14</sub> `endElement(c)`  
 $i = 1$   
 $S = 0\square\square\square$

\*<sub>15</sub> `endElement(a)`  
 $i = 0$   
 $S = \square\square\square\square$

\*<sub>16</sub> `endDocument()`

## Final Remarks on SAX

- For an XML document fragment shown on the left, SAX might actually report the events indicated on the right:

XML fragment	XML + SAX events
<pre> 1 &lt;affiliation&gt; 2   AT&amp;T Labs 3 &lt;/affiliation&gt; </pre>	<pre> 1 &lt;affiliation&gt;*<sub>1</sub> 2   AT*<sub>2</sub>&amp;*<sub>3</sub>T Labs 3 *<sub>4</sub>&lt;/affiliation&gt;*<sub>5</sub> </pre>
<hr/> <div data-bbox="473 474 974 681"> <pre> *<sub>1</sub> startElement(affiliation) *<sub>2</sub> characters("\n...AT", 5) *<sub>3</sub> characters("&amp;", 1) *<sub>4</sub> characters("T Labs\n", 7) *<sub>5</sub> endElement(affiliation) </pre> </div> <hr/>	



**White space** is reported.

**Multiple characters events** may be sent for text content (although adjacent).

(Often SAX parsers break text on entities, but may even report each character on its own.)

# Part VI

## Valid XML—DTDs

# Outline of this part

## 17 Valid XML

## 18 DTDs—Document Type Definitions

- Element Declaration
- Attribute Declaration
- Crossreferencing via ID and IDREF
- Other DTD Features
- A “Real Life” DTD—GraphML
- Concluding remarks on DTDs

## 19 XML Schema

- Some XML Schema Constructs
- Other XML Schema Concepts

## 20 Validating XML Documents Against DTDs

- Regular Expressions
- Evaluating Regular Expressions (Matching)
- Plugging It All Together

# Valid XML

- More often than not, applications that operate on XML data require the XML input data to conform to a **specific XML dialect**.



- This requirement is more strict than just XML well-formedness.
- The (hard-coded) application logic relies on, *e.g.*,
  - ▶ the **presence** or absence of specifically named elements [attributes],
  - ▶ the **order** of child elements within an enclosing element,
  - ▶ attributes having exactly one of several **expected values**, ...
- If the input data fails to meet the requirements, results are often disastrous.

**Example:** Transform element `amount` into attribute:

```
<bet gambler="doe"><amount>7</amount>...</bet>
```



```
<bet gambler="doe" amount="7">...</bet>
```

## Stumbling Code

```
1  $ java foo input.xml
2  Calculate gambling results.....
3  Exception in thread "main" java.lang.NumberFormat
4      at java.lang.Integer.parseInt(Integer.java:394)
5      at java.lang.Integer.parseInt(Integer.java:476)
6      at foo.getResult(foo.java:169)
7      at foo.main(foo.java:214)
8
9  $ java bar input.xml
10 Exception in thread "main" java.lang.NullPointerException
11     at bar.printGamblers(bar.java:186)
12     at bar.main(bar.java:52)
13
14 $ java baz input.xml
15 Gambler John Doe lost 0.
16 Gambler Johnny Average lost 0.
17 Gambler Betty Bet lost 0.
18 Gambler Linda Loser lost 0.
19 Gambler Robert Johnson lost 0.
20
21 $ █
```



# DTDs–Document Type Definitions

- The XML Recommendation<sup>15</sup> includes technology that enables applications to rigidly specify the XML **dialect** (the **document type**) they expect to see: **DTD** s (Document Type Definitions).
- XML parsers use the DTD to ensure that input data is not only well-formed but also conforms to the DTD (XML speak: input data is **valid**).



Valid XML documents  $\subset$  Well-formed XML documents

- **Document validation** is critical, if
  - ▶ distinct organizations (B2B) need to share XML data: also **share the DTDs**,
  - ▶ applications need to discover and explore yet unknown XML dialects,
  - ▶ high-speed XML throughput is required (once the input is validated, we can **abandon** a lot of **runtime checks**).

<sup>15</sup><http://www.w3.org/TR/REC-xml>

- A document's DTD is directly attached to its XML text using a DOCTYPE declaration:

	DOCTYPE Declaration
1	<code>&lt;?xml version="1.0"?&gt;</code>
2	<code>&lt;!DOCTYPE t d<sub>e</sub> d<sub>i</sub>&gt;</code>
3	<code>&lt;t&gt;</code>
4	<code>...</code>
5	<code>&lt;/t&gt;</code>

- ▶ The DOCTYPE declaration follows the text declaration (`<?xml...?>`) (comments `<!--...-->`, processing instructions `<?...?>` in between are OK).
- ▶ The first parameter  $t$  of the DOCTYPE declaration is required to match the document's root element tag.
- ▶ The document type definition itself consists of an **external subset** ( $d_e \equiv \text{SYSTEM "uri"}$ ) as well as an **internal subset** ( $d_i \equiv [...]$ ), i.e., embedded in the document itself).
- ▶ Both subsets are optional. Should clashes occur, declarations in the internal subset override those in the external subset.

## Example:


```
<!DOCTYPE strip
  SYSTEM "file:///DilbertML.dtd" [ <!ENTITY phb "Pointy-Haired Boss"> ] >
```

external subset                      internal subset

## The ELEMENT Declaration

- The DTD ELEMENT declaration, in some sense, defines the *vocabulary* available in an XML dialect.
- Any XML element  $t$  to be used in the dialect needs to be introduced via

`<!ELEMENT t cm>`

- ▶ The **content model**  $cm$  of the element defines which **element content** is considered valid.
- ▶ Whenever an application encounters a  $t$  element  anywhere in a valid document, it may assume that  $t$ 's content conforms to  $cm$ .

Content model	Valid content
ANY	arbitrary well-formed XML content
EMPTY	no child elements allowed (attributes OK)
regular expression over tag names, #PCDATA, and constructors , ,   , + , * , ?	order and occurrence of child elements and text content must <i>match</i> the regular expression

## N.B.

- A DTD with `<!ELEMENT t ANY >` gives the application no clue about *t*'s content. Use judiciously.
- A `<!ELEMENT t EMPTY >` forbids any content for *t* elements.

**Example:** (X)HTML `img`, `br` tags:

### XHTML 1.0 Strict DTD

```
1      <!ELEMENT img EMPTY>
2      ...
3      <!ELEMENT br  EMPTY>
```

- **Regular expression** content models provide control over the exact order and occurrence of children nodes below an element node:

Reg. exp.	Semantics
$t$ (tag name)	child element with tag $t$
#PCDATA	text content (parsed character data)
$c_1$ , $c_2$	$c_1$ followed by $c_2$
$c_1$   $c_2$	$c_1$ or, alternatively, $c_2$
$c^+$	$c$ , one or more times
$c^*$	$c$ , zero or more times
$c?$	optional $c$

### Example (DilbertML):

#### DilbertML.dtd

```
1  <!ELEMENT panel (scene, bubbles*) >
2  <!ELEMENT scene (#PCDATA) >
3  <!ELEMENT bubbles (bubble+) >
4  <!ELEMENT bubble (#PCDATA) >
```

**Example** (modify bubble element so that we can use <loud>...</loud> and <whisper>...</whisper> to markup speech more accurately):

```
<bubble>E-mail <loud>two copies</loud> to me when you're done.</bubble>
```

#### DilbertML.dtd

```
1  <!ELEMENT panel    (scene, bubbles*) >
2  <!ELEMENT scene    (#PCDATA) >
3  <!ELEMENT bubbles  (bubble+) >
4  <!ELEMENT bubble    (#PCDATA | loud | whisper)* >
5  <!ELEMENT loud      (#PCDATA) >
6  <!ELEMENT whisper   (#PCDATA) >
```

- Element bubble is said to allow **mixed content** (text and element nodes), while panel and bubbles allow **element content** only. Elements scene, loud, whisper have **text content**.
- DTD restriction:  
The above example shows the only acceptable placements of #PCDATA in content models.



## Element Content vs. Mixed Content

Element `bubbles` has *element content*: an XML parser will *not* report white space contained in a `bubbles` element to its underlying application.

Element `bubble` has *mixed content*: white space (`#PCDATA`) is regarded essential and thus reported to the application.

### Dilbert.xml

```

1 <bubbles> ␣
2   ...<bubble> ␣
3     ...<loud>No coffee</loud> ␣
4     ...no research ... ␣
5   ...</bubble> ␣
6 </bubbles>
  
```

---

### SAX events

---

```

startElement (t="bubbles")
startElement (t="bubble")
characters (buf = "... ␣", len = 4)
startElement (t="loud")
      ⋮
endElement (t="bubble")
endElement (t="bubbles")
  
```

---

# Ex.: DTD and valid XML encoding academic titles

## Academic.xml

```
1 <?xml version="1.0"?>
2 <!DOCTYPE academic [
3     <!ELEMENT academic (Prof?, (Dr, (rernat|emer|phil)*)?,
4         Firstname, Middlename*, Lastname) >
5     <!ELEMENT Prof      EMPTY >
6     <!ELEMENT Dr        EMPTY >
7     <!ELEMENT rernat    EMPTY >
8     <!ELEMENT emer      EMPTY >
9     <!ELEMENT phil      EMPTY >
10    <!ELEMENT Firstname (#PCDATA) >
11    <!ELEMENT Middlename (#PCDATA) >
12    <!ELEMENT Lastname  (#PCDATA) >
13 ]>
14
15 <academic>
16     <Prof/> <Dr/> <emer/>
17     <Firstname>Don</Firstname>
18     <Middlename>E</Middlename>
19     <Lastname>Knuth</Lastname>
20 </academic>
```



# The ATTLIST Declaration

- Using the DTD ATTLIST declaration, validation of XML documents is extended to attributes.
- The ATTLIST declaration associates a list of attribute names  $a_i$  with their owning element named  $t$ :

## ATTLIST Declaration

```
1  <!ATTLIST t
2       $a_1$   $\tau_1$   $d_1$ 
3      ...
4       $a_n$   $\tau_n$   $d_n$ 
5  >
```

- ▶ The **attribute types**  $\tau_i$  define which values are valid for attribute  $a_i$ .
- ▶ The **defaults**  $d_i$  indicate if  $a_i$  is required or optional (and, if absent, if a default value should be assumed for  $a_i$ ).
- ▶ In XML, the attributes of an element are *unordered*. The ATTLIST declaration prescribes *no order* of attribute usage.

- Via **attribute types**, control over the valid attribute values can be exercised:

Attribute Type $\tau_i$	Semantics
CDATA	character data (no <, but &lt;;, ...)
$(v_1   v_2   \dots   v_m)$	enumerated literal values
ID	value is document-wide unique identifier for owner element
IDREF	references an element via its ID attribute

### Example:

#### Academic.xml (fragment)

```
1  <!ELEMENT academic (Firstname, Middlename*, Lastname) >
2  <!ATTLIST academic
3      title (Prof|Dr) #REQUIRED
4      type  CDATA #IMPLIED >
5  >
6
7  <academic title="Dr" type="rer.nat."> ... </academic>
```

- Attribute defaulting in DTDs:

Attribute Default $d_i$	Semantics
#REQUIRED	element must have attribute $a_i$
#IMPLIED	attribute $a_i$ is optional
$v$ (a value)	attribute $a_i$ is optional, if absent, default value $v$ for $a_i$ is assumed
#FIXED $v$	attribute $a_i$ is optional, if present, must have value $v$

### Example:

#### DilbertML.dtd (fragment)

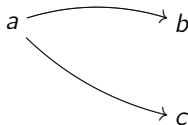
```
1 <!DOCTYPE strip [  
2   ...  
3   <!ELEMENT characters (character+) >  
4   <!ATTLIST characters  
5     alphabetical (yes|no) "no" >   <!-- play safe -->  
6   <!ELEMENT character (#PCDATA) >  
7 ]>
```

## Crossreferencing via ID and IDREF

- Well-formed XML documents essentially describe tree-structured data.
- Attributes of type ID and IDREF may be used to encode **graph structures** in XML. A validating XML parser can check such a graph encoding for consistent connectivity.
- To establish a directed edge between two XML document nodes  $a$  and  $b$



- attach a unique **identifier** to node  $b$  (using an ID attribute), then
- refer** to  $b$  from  $a$  via this identifier (using an IDREF attribute),
- for an outdegree  $> 1$  (see below), use an IDREFS attribute.

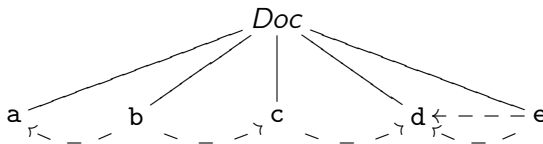


**Graph.xml**

```

1  <?xml version="1.0"?>
2  <!DOCTYPE graph [
3      <!ELEMENT graph (node+) >
4      <!ELEMENT node ANY >      <!-- attach arbitrary data to a node -->
5      <!ATTLIST node
6          id      ID      #REQUIRED
7          edges   IDREFS #IMPLIED >  <!-- we may have nodes with outdegree 0 -->
8  ]>
9
10 <graph>
11   <node id="A">a</node>
12   <node id="B" edges="A C">b</node>
13   <node id="C" edges="D">c</node>
14   <node id="D">d</node>
15   <node id="E" edges="D D">e</node>
16 </graph>

```



## Example (Character references in DilbertML)

### DilbertML.dtd (fragment)

```
1  <!DOCTYPE strip [  
2      ...  
3      <!ELEMENT character (#PCDATA) >  
4      <!ATTLIST character  
5          id          ID                      #REQUIRED >  
6      <!ELEMENT bubble      (#PCDATA) >  
7      <!ATTLIST bubble  
8          speaker IDREF                      #REQUIRED  
9          to       IDREFS                    #IMPLIED  
10         tone     (angry|question|...) #IMPLIED >  
11  ]>
```

### Validation results (messages generated by Apache's Xerces):

- Setting attribute to to some random non-existent character identifier:  
ID attribute 'yoda' was referenced but never declared
- Using a non-enumerated value for attribute tone:  
Attribute 'tone' does not match its defined enumeration list

DilbertML.dtd

```

1 <!DOCTYPE strip [
2   <!ELEMENT strip      (prolog, panels) >
3   <!ATTLIST strip
4     copyright CDATA #IMPLIED
5     year      CDATA #IMPLIED >
6
7   <!ELEMENT prolog      (series, author, characters) >
8
9   <!ELEMENT series      (#PCDATA) >
10  <!ATTLIST series
11    href CDATA #IMPLIED >
12
13  <!ELEMENT author      (#PCDATA) >
14
15  <!ELEMENT characters (character+) >
16  <!ATTLIST characters alphabetical (yes|no) 'no' >
17
18  <!ELEMENT character   (#PCDATA) >
19  <!ATTLIST character   id ID #REQUIRED >
20
21  <!ELEMENT panels      (panel+) >
22  <!ATTLIST panels
23    length CDATA #IMPLIED >
24
25  <!ELEMENT panel       (scene, bubbles*) >
26  <!ATTLIST panel
27    no CDATA #IMPLIED >
28
29  <!ELEMENT scene       (#PCDATA) >
30  <!ATTLIST scene
31    visible IDREFS #IMPLIED >
32
33  <!ELEMENT bubbles     (bubble+) >
34
35  <!ELEMENT bubble      (#PCDATA) >
36  <!ATTLIST bubble
37    speaker IDREF #REQUIRED
38    to      IDREFS #IMPLIED
39    tone     (question|angry|screaming) #IMPLIED >
40 ]>

```

# Other DTD features

- **User-defined entities** via `<!ENTITY e d>` declarations (usage: `&e;`)

```
<!ENTITY phb "The Pointy-Haired Boss">
```

- **Parameter entities** (“DTD macros”) via `<!ENTITY % e d>` (usage: `%e;`)

```
<!ENTITY ident "ID #REQUIRED">
```

```
...
```

```
<!ATTLIST character
```

```
  id %ident; >
```

- **Conditional sections** in DTDs via `<![INCLUDE[...]]>` and `<![IGNORE[...]]>`

```
<!ENTITY % withCharacterIDs "INCLUDE" >
```

```
<!ATTLIST bubble
```

```
  <![%withCharacterIDs;
```

```
    speaker %ident;
```

```
    to      %ident;
```

```
  ]]>
```

```
  tone      (angry|question|...) #IMPLIED >
```



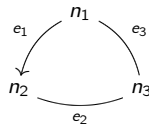
# A “Real Life” DTD—GraphML

- GraphML<sup>16</sup> has been designed to provide a powerful and easy-to-use file format to represent arbitrary graphs.
  - ① Graphs (element graph) are specified as lists of nodes and edges. Edges point from source to target.
  - ② Nodes and edges may be annotated using arbitrary descriptions and data.
  - ③ Edges may be directed (and attribute edgedefault of graph).
  - ④ Edges may be attached to nodes at specific ports (north, west, ...).

## Example:

GraphML.xml

```
1 <graphml>
2   <graph edgedefault="undirected">
3     <node id="n1"/>
4     <node id="n2"/>
5     <node id="n3"/>
6     <edge id="e1" source="n1" target="n2" directed="true"/>
7     <edge id="e2" source="n2" target="n3" directed="false"/>
8     <edge id="e3" source="n3" target="n1"/>
9   </graph>
10 </graphml>
```



<sup>16</sup><http://www.graphdrawing.org/>

## GraphML.dtd

```

1 <!-- ===== -->
2 <!-- GRAPHML DTD (flat version) ===== -->
3 <!-- file: graphml.dtd
4
5     SYSTEM "http://www.graphdrawing.org/dtds/graphml.dtd"
6
7     xmlns="http://www.graphdrawing.org/xmlns/graphml"
8     (consider these urls as examples)
9
10    ===== -->
11
12
13 <!--=====-->
14 <!--elements of GRAPHML-->
15 <!--=====-->
16
17
18 <!ELEMENT graphml  ((desc)?,(key)*,((data)|(graph))*)>
19
20
21 <!ELEMENT locator EMPTY>
22 <!ATTLIST locator
23     xmlns:xlink    CDATA    #FIXED

```

```

24      "http://www.w3.org/TR/2000/PR-xlink-20001220/"
25      xlink:href      CDATA      #REQUIRED
26      xlink:type      (simple) #FIXED      "simple"
27 >
28
29 <!ELEMENT desc (#PCDATA)>
30
31
32 <!ELEMENT graph      ((desc?),((((data)|(node)|
33                        (edge)|(hyperedge))*|(locator)))>
34 <!ATTLIST graph
35      id              ID              #IMPLIED
36      edgedefault     (directed|undirected) #REQUIRED
37 >
38
39 <!ELEMENT node      (desc?,(((data|port)*,graph?)|locator))>
40 <!ATTLIST node
41      id              ID              #REQUIRED
42 >
43
44 <!ELEMENT port      ((desc?),((data)|(port))*>
45 <!ATTLIST port
46      name            NMTOKEN      #REQUIRED

```

```
47 >
48
49
50 <!ELEMENT edge ((desc)?,(data)*,(graph)?)>
51 <!ATTLIST edge
52         id          ID          #IMPLIED
53         source       IDREF       #REQUIRED
54         sourceport   NMTOKEN     #IMPLIED
55         target       IDREF       #REQUIRED
56         targetport   NMTOKEN     #IMPLIED
57         directed     (true|false) #IMPLIED
58 >
59
60
61 <!ELEMENT hyperedge ((desc)?,((data)|(endpoint))*,(graph)?)>
62 <!ATTLIST hyperedge
63         id          ID          #IMPLIED
64 >
65
66 <!ELEMENT endpoint ((desc)?)>
67 <!ATTLIST endpoint
68         id          ID          #IMPLIED
69         node        IDREF       #REQUIRED
```

```
70         port  NMTOKEN          #IMPLIED
71         type   (in|out|undir) "undir"
72     >
73
74
75 <!--ELEMENT key (#PCDATA)>
76 <!--ATTLIST key
77         id ID                                #REQUIRED
78         for (graph|node|edge|hyperedge|port|endpoint|all) "all"
79     >
80
81 <!--ELEMENT data (#PCDATA)>
82 <!--ATTLIST data
83         key IDREF                        #REQUIRED
84         id ID                            #IMPLIED
85     >
86
87 <!--=====
88         end of graphml.dtd
89     =====>
```

## XMLns: XML name spaces

- ... provide a means of importing a couple of predefined element and attribute declarations (from different DTDs),
- are used to resolve name clashes when importing several DTDs
- are declared as an **attribute of the top-level document element**:

### Name space declaration

```
1 <elementname xmlns:name_space_ID = "name_space_URI">
```

For example:

### Importing several DTD (name spaces)

```
1 <touristinformation
2   xmlns:hotelinfo="http://www.hotels.de"
3   xmlns:eventinfo="http://www.events.de">
4   <hotelinfo:ort>Konstanz</hotelinfo:ort>
5   <eventinfo:ort>Zuerich</eventinfo:ort>
6 </touristinformation>
```

# Concluding remarks

- DTD syntax:
  - Pro:** compact, easy to understand
  - Con:** not in XML
- DTD functionality:
  - ▶ no distinguishable types (everything is character data)
  - ▶ no further value constraints (e.g., cardinality of sequences)
  - ▶ no built-in scoping (but: use XMLNs for name spaces)
- **From a database perspective, DTDs are a poor schema definition language.** (but: see XMLSchema below...)

# XML Schema

- With **XML Schema**<sup>17</sup>, the W3C provides a **schema description language** for XML documents that goes way beyond the capabilities of the “native” DTD concept.

Specifically:

- ① XML Schema **descriptions are valid XML documents** themselves.
- ② XML Schema provides a **rich set of built-in data types**.  
(Modelled after the SQL and Java type systems.)
- ③ **Far-reaching control over the values** a data type can assume (**facets**).
- ④ Users can extend this type system via **user-defined types**.
- ⑤ XML element (and attribute) types may even be derived by **inheritance**.

## XML Schema vs. DTDs

Ad ①: Why would you consider this an advantage?

Ad ②: What are the data types supported by DTDs?

<sup>17</sup><http://www.w3.org/TR/xmlschema-0/>



# Some XML Schema Constructs

## Declaring an element

❶ `<xsd:element name="author"/>`

No further typing specified: the author element may contain string values only.

## Declaring an element with bounded occurrence

❶ `<xsd:element name="character" minOccurs="0" maxOccurs="unbounded"/>`

Absence of `minOccurs`/`maxOccurs` implies *exactly once*.

## Declaring a typed element

❶ `<xsd:element name="year" type="xsd:date"/>`

Content of year takes the format YYYY-MM-DD. Other **simple types**: string, boolean, number, float, duration, time, base64Binary, AnyURI, ...

- **Simple types** are considered **atomic** with respect to XML Schema (e.g., the YYYY part of an `xsd:date` value has to be extracted by the XML application itself).

- Non-atomic **complex types** are built from simple types using **type constructors**.

#### Declaring sequenced content

```
1  <xsd:complexType name="Characters">
2    <xsd:sequence>
3      <xsd:element name="character" minOccurs="1" maxOccurs="unbounded"/>
4    </xsd:sequence>
5  </xsd:complexType>
6  <xsd:complexType name="Prolog">
7    <xsd:sequence>
8      <xsd:element name="series"/>
9      <xsd:element name="author"/>
10     <xsd:element name="characters" type="Characters"/>
11   </xsd:sequence>
12 </xsd:complexType>
13 <xsd:element name="prolog" type="Prolog"/>
```

An `xsd:complexType` may be used anonymously (no name attribute).

- With attribute `mixed="true"`, an `xsd:complexType` admits **mixed content**.

- New complex types may be **derived** from an existing (base) type.

#### Deriving a new complex type

```
1 <xsd:element name="newprolog">
2   <xsd:complexType>
3     <xsd:complexContent>
4       <xsd:extension base="Prolog">
5         <xsd:element name="colored" type="xsd:boolean"/>
6       </xsd:extension>
7     </xsd:complexContent>
8   </xsd:complexType>
9 </xsd:element>
```

- **Attributes** are declared within their owner element.

#### Declaring attributes

```
1 <xsd:element name="strip">
2   <xsd:attribute name="copyright"/>
3   <xsd:attribute name="year" type="xsd:gYear"/> ...
4 </xsd:element>
```

Other xsd:attribute modifiers: use (required, optional, prohibited), fixed, default.

- The validation of an XML document against an XML Schema declaration goes as far as peeking into the **lexical representation** of simple typed values.

#### Restricting the value space of a simple type (enumeration)

```
1 <xsd:simpleType name="Tone">
2   <xsd:restriction base="xsd:string">
3     <xsd:enumeration value="question"/>
4     <xsd:enumeration value="angry"/>
5     <xsd:enumeration value="screaming"/>
6   </xsd:restriction>
7 </xsd:simpleType>
```

#### Restricting the value space of a simple type (regular expression)

```
1 <xsd:simpleType name="AreaCode">
2   <xsd:restriction base="xsd:string">
3     <xsd:pattern value="0[0-9]+"/>
4     <xsd:minLength value="3"/>
5     <xsd:maxLength value="5"/>
6   </xsd:restriction>
7 </xsd:simpleType>
```

- Other **facets**: length, maxInclusive, minExclusive, ...

# Other XML Schema Concepts

- **Fixed** and **default** element content,
- support for **null values**,
- uniqueness constraints, arbitrary **keys** (specified via XPath), local keys, key references, and referential integrity,
- ...

# Validating XML Documents Against DTDs

- To validate against this DTD ...

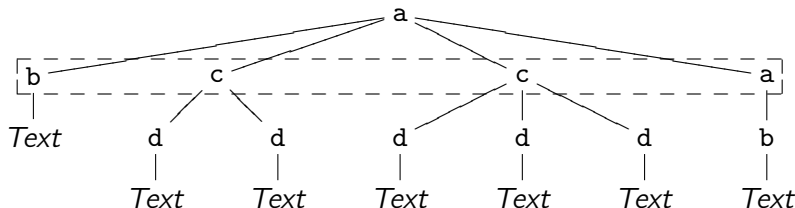
## DTD featuring regular expression (RE) content models

```

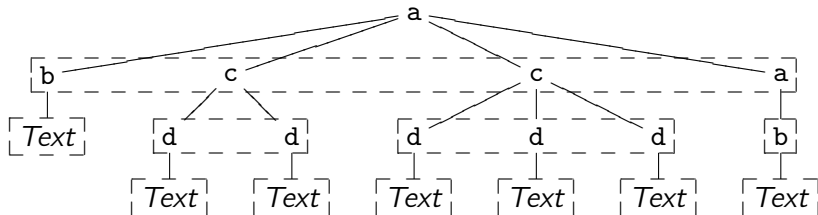
1  <!DOCTYPE a [
2    <!ELEMENT a (b, c*, a?)>
3    <!ELEMENT b (#PCDATA) >
4    <!ELEMENT c (d, d+) >
5    <!ELEMENT d (#PCDATA) >
6  ]>

```

... means to check that the **sequence of child nodes** for each element **matches** its RE content model:



- When, during RE matching, we encounter a child element  $t$ , we need to **recursively check  $t$ 's content model  $cm(t)$**  in the same fashion:



$$cm(a) = b, c^*, a?$$

$$cm(b) = \#PCDATA$$

$$cm(c) = d, d^+$$

$$cm(d) = \#PCDATA$$

### SAX and DTD validation?

- Can we use SAX to drive this validation (= RE matching) process?
- If so, which SAX events do we need to catch to implement this?

# Regular Expressions

- To provide adequate support for SAX-based XML validation, we assume REs of the following structure:

$RE = \emptyset$	matches <b>nothing</b>
$\varepsilon$	matches <b>empty</b> sequence of SAX events
$\#PCDATA$	matches <i>characters</i> (.)
$t$	matches <i>startElement</i> ( $t, \cdot$ )
$RE, RE$	<b>concatenation</b>
$RE^+$	<b>one-or-more repetitions</b>
$RE^*$	<b>zero-or-more repetitions</b>
$RE?$	<b>option</b>
$RE \mid RE$	<b>alternative</b>
$(RE)$	



- $\emptyset$  and  $\varepsilon$  are *not* the same thing.
- In the *startElement*( $t, \cdot$ ) callback we can process `<!ATTLIST t ...>` declarations (not discussed here).



- Associated with each RE is the **regular language**  $L(RE)$  (here: sets of sequences of SAX events) this RE **accepts**:

$$L(\emptyset) = \emptyset$$

$$L(\varepsilon) = \{\varepsilon\}$$

$$L(\#PCDATA) = \{characters(\cdot)\}$$

$$L(t) = \{startElement(t, \cdot)\}^{18}$$

$$L(RE_1, RE_2) = \{s_1 s_2 \mid s_1 \in L(RE_1), s_2 \in L(RE_2)\}$$

$$L(RE^+) = \bigcup_{i=1}^{\infty} L(RE^i)$$

$$L(RE^*) = \bigcup_{i=0}^{\infty} L(RE^i)$$

$$L(RE?) = \{\varepsilon\} \cup L(RE)$$

$$L(RE_1 \mid RE_2) = L(RE_1) \cup L(RE_2)$$

- N.B.:**  $RE^0 = \varepsilon$  and  $RE^i = RE, RE^{i-1}$ .

<sup>18</sup>To save trees, we will abbreviate this as  $\{t\}$  from now on.

## Example

- Which sequence of SAX events is matched by the RE  $\#PCDATA \mid b^*$ ?

$$L(\#PCDATA \mid b^*)$$

$$= L(\#PCDATA) \cup L(b^*)$$

$$= L(\#PCDATA) \cup \bigcup_{i=0}^{\infty} L(b^i)$$

$$= L(\#PCDATA) \cup L(b^0) \cup \bigcup_{i=1}^{\infty} L(b^i)$$

$$= L(\#PCDATA) \cup L(b^0) \cup L(b^1) \cup \bigcup_{i=2}^{\infty} L(b^i)$$

$$= L(\#PCDATA) \cup L(b^0) \cup L(b^1) \cup L(b^2) \cup \bigcup_{i=3}^{\infty} L(b^i)$$

$$= L(\#PCDATA) \cup L(\varepsilon) \cup L(b) \cup L(b, b^1) \cup \dots$$

$$= L(\#PCDATA) \cup L(\varepsilon) \cup L(b) \cup \{s_1 s_2 \mid s_1 \in L(b), s_2 \in L(b^1)\} \cup \dots$$

$$= \{characters(\cdot), \varepsilon, b, bb, \dots\}$$

$$\pencil L(d, d^+) = ?$$

# Evaluating Regular Expressions (Matching)

- Now that we are this far, we know that matching a sequence of SAX events  $s$  against the content model of element  $t$  means to carry out the test

$$s \stackrel{?}{\in} L(cm(t)) .$$

- $L(cm(t))$ , however, might be infinite or otherwise too costly to construct inside our DTD validator.
- We thus follow a different path that avoids to enumerate  $L(cm(t))$  at all.
- Instead, we will use the **derivative**  $s \backslash RE$  of  $RE$  with respect to input event  $s$ :

$$L(s \backslash RE) = \{s' \mid s s' \in L(RE)\}$$

“ $s \backslash RE$  matches everything matched by  $RE$ , with head  $s$  cut off.”

- We can use the derivate operator  $\backslash$  to develop a simple **RE matching procedure**.

Suppose we are to match the SAX event sequence  $s_1s_2s_3$  against  $RE$ :

$$\begin{aligned}
 s_1s_2s_3 \in L(RE) &\Leftrightarrow s_1s_2s_3\varepsilon \in L(RE) \\
 &\Leftrightarrow s_2s_3\varepsilon \in L(s_1 \backslash RE) \\
 &\Leftrightarrow s_3\varepsilon \in L(s_2 \backslash (s_1 \backslash RE)) \\
 &\Leftrightarrow \varepsilon \in L(s_3 \backslash (s_2 \backslash (s_1 \backslash RE))) \quad .
 \end{aligned}$$

- We thus have solved our matching problem if
  - ① we can efficiently **test for  $\varepsilon$ -containment** for a given RE, and
  - ② we are able to **compute**  $L(s \backslash RE)$  for any given input event  $s$  and any  $RE$ .

✎ Ad ①: Testing for  $\varepsilon$ 's presence in a regular language.

Define a predicate (boolean function)  $nullable(RE)$  such that

$$nullable(RE) \Leftrightarrow \varepsilon \in L(RE) .$$

$$nullable(\emptyset) = false$$

$$nullable(\varepsilon) = true$$

$$nullable(\#PCDATA) = false$$

$$nullable(t) =$$

$$nullable(RE_1, RE_2) =$$

$$nullable(RE^+) =$$

$$nullable(RE^*) =$$

$$nullable(RE?) =$$

$$nullable(RE_1 \mid RE_2) =$$

## Example

Does  $L(\#PCDATA \mid b^*)$  contain the empty SAX event sequence  $\varepsilon$ ?

$$\begin{aligned} nullable(\#PCDATA \mid b^*) &= nullable(\#PCDATA) \vee nullable(b^*) \\ &= false \vee true \\ &= true . \end{aligned}$$

✎  $nullable(Prof?, Dr, (rernat \mid emer \mid phil)^+) = ?$

Ad ②: Note that the **derivative**  $s\backslash$  is an operator on REs (to REs). We define it like follows and justify this definition on the next slides.

$$s\backslash\emptyset = \emptyset$$

$$s\backslash\varepsilon = \emptyset$$

$$s\backslash\#PCDATA = \begin{cases} \varepsilon & \text{if } s = \text{characters}(\cdot) \\ \emptyset & \text{otherwise} \end{cases}$$

$$s\backslash t = \begin{cases} \varepsilon & \text{if } s = \text{startElement}(t, \cdot) \quad // \star \text{ recursively match } cm(t) \\ \emptyset & \text{otherwise} \end{cases}$$

$$s\backslash(RE_1, RE_2) = \begin{cases} ((s\backslash RE_1), RE_2) \mid (s\backslash RE_2) & \text{if } nullable(RE_1) \\ (s\backslash RE_1), RE_2 & \text{otherwise} \end{cases}$$

$$s\backslash RE^+ = (s\backslash RE), RE^*$$

$$s\backslash RE^* = (s\backslash RE), RE^*$$

$$s\backslash RE? = s\backslash RE$$

$$s\backslash(RE_1 \mid RE_2) = (s\backslash RE_1) \mid (s\backslash RE_2)$$

## Correctness: Case Analysis I

To assess the correctness of this derivative construction  $s \backslash RE = RE'$  we can systematically check all 9 cases for **language equivalence**, *i.e.*

$$L(s \backslash RE) = L(RE') .$$

①  $RE = \emptyset$ :

$$\begin{aligned} L(s \backslash \emptyset) &= \{s' \mid s s' \in L(\emptyset)\} \\ &= \{s' \mid s s' \in \emptyset\} \\ &= \emptyset \\ &= L(\emptyset). \end{aligned}$$



## Correctness: Case Analysis II

②  $RE = \varepsilon$ :

$$\begin{aligned}L(s \setminus \varepsilon) &= \{s' \mid s s' \in L(\varepsilon)\} \\&= \{s' \mid s s' \in \{\varepsilon\}\} \\&= \emptyset \\&= L(\emptyset).\end{aligned}$$

③  $RE = \#PCDATA, s = \text{characters}(\cdot)$ :

$$\begin{aligned}L(\text{characters}(\cdot) \setminus \#PCDATA) &= \{s' \mid \text{characters}(\cdot) s' \in L(\#PCDATA)\} \\&= \{s' \mid \text{characters}(\cdot) s' \in \{\text{characters}(\cdot)\}\} \\&= \{\varepsilon\} \\&= L(\varepsilon).\end{aligned}$$

## Correctness: Case Analysis III

$RE = \#PCDATA, s \neq characters(\cdot)$ :

$$\begin{aligned}
 L(s \setminus \#PCDATA) &= \{s' \mid s s' \in L(\#PCDATA)\} \\
 &= \{s' \mid s s' \in \{characters(\cdot)\}\} \\
 &= \emptyset \\
 &= L(\emptyset).
 \end{aligned}$$

④  $RE = t$ . Analogous to ③.

⑤  $RE = RE_1, RE_2, nullable(RE_1) = false$ :

$$\begin{aligned}
 L(s \setminus (RE_1, RE_2)) &= \{s' \mid s s' \in L(RE_1, RE_2)\} \\
 &= \{s' \mid s' \in L((s \setminus RE_1), RE_2)\} \\
 &= L((s \setminus RE_1), RE_2).
 \end{aligned}$$

## Correctness: Case Analysis IV

$RE = RE_1, RE_2$ ,  $nullable(RE_1) = true$ :

$$\begin{aligned} L(s \setminus (RE_1, RE_2)) &= \{s' \mid s s' \in L(RE_1, RE_2)\} \\ &= \{s' \mid s s' \in L(RE_2) \vee s s' \in L(RE_1, RE_2)\} \\ &= \{s' \mid s' \in L(s \setminus RE_2) \vee s' \in L((s \setminus RE_1), RE_2)\} \\ &= \{s' \mid s' \in L(s \setminus RE_2)\} \cup \{s' \mid s' \in L((s \setminus RE_1), RE_2)\} \\ &= L(s \setminus RE_2) \cup L((s \setminus RE_1), RE_2) \\ &= L((s \setminus RE_2) \mid ((s \setminus RE_1), RE_2)). \end{aligned}$$

# Correctness: Case Analysis V

⑥  $RE = RE_1 \mid RE_2$ :

$$\begin{aligned}L(s \setminus (RE_1 \mid RE_2)) &= \{s' \mid s s' \in L(RE_1 \mid RE_2)\} \\&= \{s' \mid s s' \in L(RE_1) \cup L(RE_2)\} \\&= \{s' \mid s s' \in L(RE_1)\} \cup \{s' \mid s s' \in L(RE_2)\} \\&= \{s' \mid s' \in L(s \setminus RE_1)\} \cup \{s' \mid s' \in L(s \setminus RE_2)\} \\&= L(s \setminus RE_1) \cup L(s \setminus RE_2) \\&= L((s \setminus RE_1) \mid (s \setminus RE_2)).\end{aligned}$$

## Correctness: Case Analysis VI

7  $RE = RE^*$ ,  $nullable(RE) = false$ :

$$\begin{aligned}
 L(s \setminus RE^*) &= L(s \setminus (\varepsilon \mid (RE, RE^*))) \\
 &= L(s \setminus \varepsilon) \cup L(s \setminus (RE, RE^*)) \\
 &= L(s \setminus (RE, RE^*)) \\
 &= L((s \setminus RE), RE^*).
 \end{aligned}$$

$RE = RE^*$ ,  $nullable(RE) = true$ :

$$\begin{aligned}
 L(s \setminus RE^*) &= L(s \setminus (\varepsilon \mid (RE, RE^*))) \\
 &= L((s \setminus \varepsilon) \mid (s \setminus (RE, RE^*))) \\
 &= L(\emptyset \mid (s \setminus (RE, RE^*))) \\
 &= L(s \setminus (RE, RE^*)) \\
 &= L((s \setminus RE^*) \mid ((s \setminus RE), RE^*)) \\
 &= L(s \setminus RE^*) \cup L((s \setminus RE), RE^*) \\
 &= L((s \setminus RE), RE^*).
 \end{aligned}$$

## Correctness: Case Analysis VII

- ⑧  $RE = RE^+$ . Follows from  $RE^+ = RE \mid RE^*$ .
- ⑨  $RE = RE^?$ . Follows from  $RE^? = \varepsilon \mid RE$ .

## ✎ Matching SAX events against an RE

Assume the RE content model  $b, c^*, a?$  is to be matched against the SAX events  $bcca$ .<sup>19</sup>

To validate,

- ① construct the corresponding derivative  $RE' = a \setminus (c \setminus (c \setminus (b \setminus (b, c^*, a?))))$ ,
- ② then test  $nullable(RE')$ .

**Hint:** To simplify phase ①, use the following **laws**, valid for REs in general:

$\epsilon^*$	$=$	$\epsilon$	$\epsilon, RE$	$=$	$RE$
$\emptyset^*$	$=$	$\epsilon$	$\emptyset, RE$	$=$	$\emptyset$
$\epsilon^+$	$=$	$\epsilon$	$RE, \epsilon$	$=$	$RE$
$\emptyset^+$	$=$	$\emptyset$	$RE, \emptyset$	$=$	$\emptyset$
$\epsilon?$	$=$	$\epsilon$	$\emptyset \mid RE$	$=$	$RE$
$\emptyset?$	$=$	$\epsilon$	$RE \mid \emptyset$	$=$	$RE$

<sup>19</sup>Actual event sequence:

$startElement(b, \cdot), startElement(c, \cdot), startElement(c, \cdot), startElement(a, \cdot)$ .

# Plugging It All Together

The following SAX callbacks use the aforementioned RE matching techniques to (partially) implement DTD validation **while parsing** the input XML document:

The input DTD (declaring the content models  $cm(\cdot)$ ) is

<!DOCTYPE r [ ... ]>

startDocument()

```
S.empty();
RE ← cm(r);
return ;
```

characters(·)

```
RE ←
#PCDATA\RE;
return ;
```

startElement(t, ·)

```
RE ← t\RE;
S.push(RE);
RE ← cm(t);
return ;
```

endElement(t)

```
if nullable(RE) then
| RE ← S.pop();
else
| ★ FAIL ★;
return ;
```

endDocument()

```
★ OK ★;
```

**N.B.** Stack  $S$  is used to suspend [resume] the RE matching for a specific element node whenever SAX descends [ascends] the XML document tree.



## Part VII

# Querying XML—The XQuery Data Model

# Outline of this part

## 21 Querying XML Documents

- Overview

## 22 The XQuery Data Model

- The XQuery Type System
- Node Properties
- Items and Sequences
- Atomic Types
- Automatic Type Assignment (Atomization)
- Node Types
- Node Identity
- Document Order

# Querying XML Documents

- “**Querying** XML data” essentially means to
  - ▶ **identify (or address) nodes**,
  - ▶ to **test** certain further **properties** of these nodes,
  - ▶ then to **operate on** the matches,
  - ▶ and finally, to **construct result** XML documents as answers.
- In the XML context, the language **XQuery** plays the role that SQL has in relational databases.
- XQuery can express all of the above constituents of XML querying:
  - ▶ XPath, as an **embedded sublanguage**, expresses the **locate** and **test** parts;
  - ▶ XQuery can then iterate over selected parts, operate on and construct answers from these.
  - ▶ There are more XML languages that make use of XPath as embedded sublanguages.
- We will first look into the (XML-based) **data model** used by XQuery and XPath ...

## Motivating example

Recall DilbertML and the comic strip finder:

- 1 *"Find all bubbles with Wally being angry with Dilbert."*

**Query:** Starting from the root, locate all `bubble` elements somewhere below the `panel` element. Select those `bubble` elements with attributes `@tone = "angry"`, `@speaker = "Wally"`, and `@to = "Dilbert"`.

- 2 *"Find all strips featuring Dogbert."*

**Query:** Starting from the root, step down to the element `prolog`, then take a step down to element `characters`. Inside the latter, step down to all `character` elements and check for contents being equal to `Dogbert`.

Note the **locate, then test** pattern in both queries.

- An XML parser (with DOM/SAX backend) is all we need to implement such queries.

⇒ **Tedious!** ⇐

# XPath as an embedded sublanguage

- **XPath**<sup>20</sup> is a declarative, expression-based language to **locate and test** doc nodes (with lots of syntactic sugar to make querying sufficiently sweet).
- Addressing document nodes is a core task in the XML world. XPath occurs as an **embedded sub-language** in
  - ▶ **XSLT**<sup>21</sup> (extract and transform XML document [fragments] into XML, XHTML, PDF, ...)
  - ▶ **XQuery**<sup>22</sup> (compute with XML document nodes and contents, compute new docs, ...)
  - ▶ **XPointer**<sup>23</sup> (representation of the address of one or more doc nodes in a given XML document)

---

<sup>20</sup><http://www.w3.org/TR/xpath20/>

<sup>21</sup><http://www.w3.org/TR/xslt/>

<sup>22</sup><http://www.w3.org/TR/xquery/>

<sup>23</sup><http://www.w3.org/TR/xptr/>

# The XQuery Data Model

Like for any other database query language, before we talk about the **operators** of the language, we have to specify exactly **what** it is that **these operate on** ...

- XQuery (and the other languages) use an abstract view of the XML data, the so-called **XQuery data model**.

## Data Model (DM)

The XQuery DM determines which aspects of an XML document may be inspected and manipulated by an XQuery query.

- What exactly should the XQuery DM look like...?



A simple sequence of characters or other lexical tokens certainly seems inappropriate (too fine-grained)!

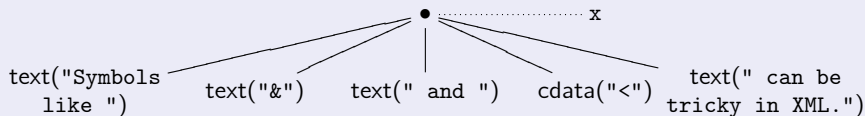
# XQuery data model (1)

## ✍ Which aspects of XML data are relevant to queries?

`<x>` Symbols like `&amp;`; and `<![CDATA[<]]>` can be tricky in XML.`</x>`

- What is an adequate representation of XML element `x`?

## DOM style...?




- Faithfully preserves entities and CDATA sections, paying the price of creating more DM nodes during parsing.

## XQuery data model (2)

### ✎ Which aspects of XML data are relevant to queries?

`<x>`Symbols like `&amp;` and `<![CDATA[<]]>` can be tricky in XML.`</x>`

### XQuery style...



text(" Symbols like  
& and < can be  
tricky in XML.")

- Do not distinguish between ordinary text, entities, and CDATA sections (the latter two are merely requirements of XML syntax).



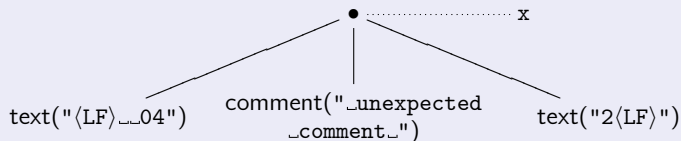
## XQuery data model (3): untyped vs. typed

An XML element containing an integer

```
<x>  
  04<!-- unexpected comment -->2  
</x>
```



Untyped view ...



## XQuery data model (3): untyped vs. typed

An XML element containing an integer

```
<x>  
  04<!-- unexpected comment -->2  
</x>
```



Typed view ...

integer(42)

- XQuery can work with the typed view, if the input XML document has been **validated** against an XML Schema description.

# XQuery DM: Node properties (1)

- A separate W3C document<sup>24</sup> describes the XQuery DM in detail.

In the XQuery DM, a tag in an XML document—an element—exhibits a number of properties, including:

<b>node-name</b>	tag name of this element
<b>parent</b>	parent element, may be empty
<b>children</b>	children lists, may be empty
<b>attributes</b>	set of attributes of this element, may be empty
<b>string-value</b>	concatenation of all string values in content
<b>typed-value</b>	element value (after validation only)
<b>type-name</b>	type name assigned by validation

---

<sup>24</sup><http://www.w3.org/TR/xpath-datamodel/>

## XQuery DM: Node properties (2)

An XML element containing an integer

```
<x>  
  04<!-- unexpected comment -->2  
</x>
```



### Node properties of unvalidated element x

<b>node-name</b>	x
<b>parent</b>	()
<b>children</b>	( $t_1, c, t_2$ )
<b>attributes</b>	$\emptyset$
<b>string-value</b>	" $\langle \text{LF} \rangle \_042 \langle \text{LF} \rangle$ "
<b>typed-value</b>	" $\langle \text{LF} \rangle \_042 \langle \text{LF} \rangle$ "
<b>type-name</b>	untypedAtomic

## XQuery DM: Node properties (3)

An XML element containing an integer

```
<x>  
  04<!-- unexpected comment -->2  
</x>
```



Node properties of **validated** element **x**

<b>node-name</b>	<b>x</b>
<b>parent</b>	<b>()</b>
<b>children</b>	<b>(<math>t_1, c, t_2</math>)</b>
<b>attributes</b>	<b><math>\emptyset</math></b>
<b>string-value</b>	<b>"042"</b>
<b>typed-value</b>	<b>42</b>
<b>type-name</b>	<b>integer</b>

## XQuery: Access to the DM in a query

XQuery provides various ways to access properties of nodes in a query.  
For example:

access node-name

```
name(<x>content here</x>) ⇒ "x"
```

access parent element (this is actually XPath functionality)

```
<x>content here</x>/parent::* ⇒ ()
```

access string value:

```
string(<x>content here</x>) ⇒ "content here"
```

# Items and sequences (1)

Two data structures are pervasive in the XQuery DM:

- 1 **Ordered, unranked trees of nodes** (XML elements, attributes, text nodes, comments, processing instructions) and
- 2 **ordered sequences of** zero or more **items**.

## Item

An XQuery **item** either is

- ▶ a **node** (of one of the kinds listed above), or
- ▶ an **atomic value** of one of the 50+ atomic types built into the XQuery DM.

## Items and sequences (2)

- A sequence of  $n$  items  $x_1$  is written in parentheses, comma-separated

### ✎ Sequence of length $n$ and empty sequence

$$(x_1, x_2, \dots, x_n) \qquad ()$$

- A single item  $x$  and the singleton sequence  $(x)$  are equivalent!
- Sequences *cannot* contain other sequences (*i.e.*, *nested sequences* are implicitly **flattened**):

### ✎ Flattening, order

$$\begin{aligned} (0, (), (1, 2), (3)) &\equiv (0, 1, 2, 3) \\ (0, 1) &\not\equiv (1, 0) \end{aligned}$$



# Sequence types (1)

XQuery uses **sequence types** to describe the type of sequences:

## Sequence types $t$ (simplified)

```
 $t$  ::= empty-sequence()  
      |  $item\ occ$   
 $occ$  ::= + | * | ? |  $\epsilon$   
 $item$  ::=  $atomic$  |  $node$  |  $item()$   
 $node$  ::=  $element(name)$  |  $text()$  |  $node()$  | ...  
 $name$  ::= * |  $QName$   
 $atomic$  ::= integer | string | double | ...
```

- A  $QName$  (qualified name) denotes an element or attribute name, possibly with a name space prefix (e.g.,  $ns:x$ ).

## Sequence types (2)

### Sequence type examples

Value	Sequence type
42	<code>integer, item()</code>
<code>&lt;x&gt;foo&lt;/x&gt;</code>	<code>element(x), item()</code>
<code>()</code>	<code>empty-sequence(), integer*</code>
<code>("foo", "bar")</code>	<code>string+, item()*</code>
<code>(&lt;x/&gt;, &lt;y/&gt;)</code>	<code>element(*)+, node()*</code>

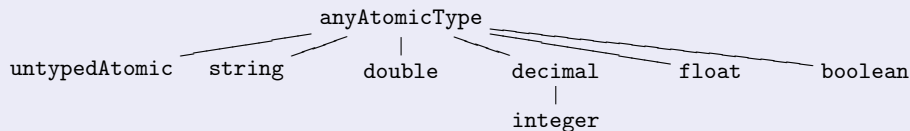
- In the table above, the most specific type is listed first.

## Items: atomic values

- XQuery, other than XPath 1.0 or XSLT which exclusively manipulate nodes, can also compute with **atomic values** (numbers, Boolean values, strings of characters, ...).
  - ▶ XQuery knows a rich collection of atomic types (*i.e.*, a versatile hierarchy of number types like fixed and arbitrary precision decimals, integers of different bit-widths, *etc.*).
  - ▶ In this course, we will only cover a subset of this rich type hierarchy.
- The hierarchy of atomic types is rooted in the special type `anyAtomicType`.

# Hierarchy of atomic types

## Atomic Type Hierachy (excerpt)



### Numeric literals

12345        (: integer :)

12.345      (: decimal :)

12.345E0    (: double :)

### Boolean literals

true()

false()

## Computing with untyped values

Atomic values of type `untypedAtomic`, which appear whenever text content is extracted from non-validated XML documents, are **implicitly converted** if they occur in expressions.

Implicit extraction<sup>25</sup> of element content and conversion of values of type `untypedAtomic`

```
"42" + 1    ⇒ ⚡ type error (compile time)
<x>42</x> + 1 ⇒ 43.0E0 (: double :)
<x>fortytwo</x> + 1 ⇒ ⚡ conversion error (runtime)
```

- This behavior saves a lot of explicit casting in queries over non-validated XML documents.

---

<sup>25</sup>Known as *atomization*, discussed later.

# Items: nodes

Just like XML, XQuery differentiates between several **kinds** of nodes:

## Six XML node kinds

```
<element attribute="foo">
  text <!--comment-->
  <?processing instruction?>
</element>
```

+ The (“invisible”) root node of any complete XML document is the so-called **document node**.

- In XQuery, a query may **extract** and **construct** nodes of all these kinds.

# Nodes: observable properties

Each node kind has specific properties but a few important properties are **shared by all kinds**:

## Node identity and document order

Each node has a **unique node identity** which is *never* modified. XQuery allows for node identity comparison using the operator `is`.

*All* nodes are ordered relative to each other, determined by the so-called **document order** (XQuery operator `<<`). This orders nodes of the same tree according to a *pre-order traversal*.

Nodes in different trees are ordered consistently.

# Node identity



## Node identity

`<x>foo</x> is <x>foo</x> ⇒ false()`

- Note: To compare items based on their **value**, XQuery offers the operators `=` and `eq`.

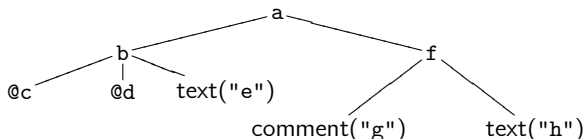
## Value comparison

`<x>foo</x> = <x>foo</x> ⇒ true()`



# Document order

```
<a>  
  <b c="..." d="...">e</b>  
  <f><!--g-->h</f>  
</a>
```



- Parent nodes precede their children and attributes (e.g.,  $a \ll b$  and  $b \ll @d$ ).  $\ll$  is transitive.
- Siblings are ordered with attributes coming first (e.g.,  $b \ll f$ ,  $@d \ll \text{text}("e")$ ), but the relative order of attributes ( $@c, @d$ ) is implementation-dependent.

# Notes on document order

- XML documents always carry this implicit order of their contents.
- Typical XML processing follows this order when accessing components of an XML document (see, *e.g.*, SAX parsing).
- Often, operations on XML documents are supposed to deliver their results also in this particular order. Document order is part of the (formal) semantics of many XML related languages.
- Contrast this with relational database query languages, where **set-orientation** always gives the freedom to the query processor to access and deliver tuples in arbitrary order!
- We will (later) see that document order has far-reaching consequences XML query processing.

## Part VIII

# XPath—Navigating XML Documents

# Outline of this part

## 23 XPath—Navigational access to XML documents

- Context
- Location steps
- Navigation axes
- Examples

## 24 XPath Semantics

- Document order & duplicates
- Predicates
- Atomization
- Positional access

# XPath—Navigational access to XML documents

- In a sense, the **traversal** or **navigation** of trees of XML nodes lies at the core of every XML query language.
- To this end, XQuery *embeds* **XPath** as its tree navigation sub-language:
  - ▶ Every XPath expression also is a correct XQuery expression.
  - ▶ XPath 2.0: [W3C](http://www.w3.org/TR/xpath20/) <http://www.w3.org/TR/xpath20/> .
- Since navigation expressions extract (potentially huge volumes of) nodes from input XML documents, the efficient implementation of the sub-language XPath is a prime concern when it comes to the construction of XQuery processors.

# Context node

- In XPath, a path traversal starts off from a **sequence of context nodes**.
  - ▶ XPath navigation syntax is simple:

## An XPath step

$cs_0/step$

- ★  $cs_0$  denotes the context node sequence, from which a navigation in direction  $step$  is taken.

- ▶ It is a common error in XQuery expressions to try and start an XPath traversal *without* the context node sequence being actually defined.

# Multiple steps

- An XPath navigation may consist of **multiple steps**  $step_i, i \geq 1$  taken in succession.
- Step  $step_1$  starts off from the context node sequence  $cs_0$  and arrives at a sequence of new nodes  $cs_1$ .
- $cs_1$  is then used as the **new context node sequence** for  $step_2$ , and so on.

## Multi-step XPath path

$$\begin{aligned} & cs_0/step_1/step_2/\dots \\ & \equiv \\ & ( \underbrace{(cs_0/step_1)}_{cs_1} / step_2 ) / \dots \end{aligned}$$

# XPath location steps

## XPath step

### Step syntax:

$$ax::nt[p_1]\cdots[p_n]$$

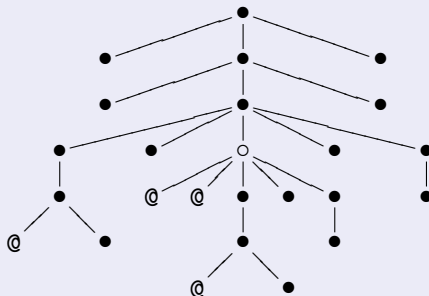
- A **step** (or **location step**)  $step_i$  specifies
  - 1 the **axis**  $ax$ , *i.e.*, the direction of navigation taken from the context nodes,
  - 2 a **node test**  $nt$ , which can be used to navigate to nodes of certain kind (*e.g.*, only attribute nodes) or name,
  - 3 optional **predicates**  $p_i$  which further filter the sequence of nodes we navigated to.



## XPath axes

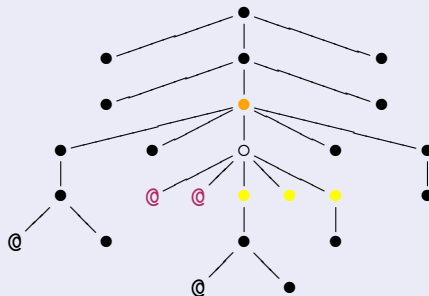
XPath defines a family of **12 axes** allowing for flexible navigation within the node hierarchy of an XML tree.

XPath axes semantics,  $\circ$  marks the context node



- @ marks attribute nodes, • represents any other node kind (inner • nodes are element nodes).

XPath axes: **child**, **parent**, **attribute**

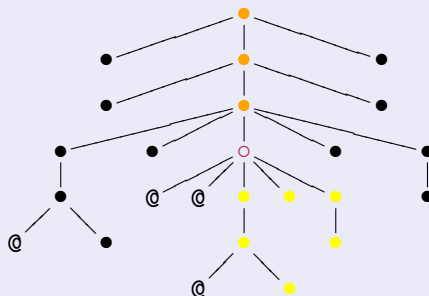


- **Note:** the `child` axis does *not* navigate to the attribute nodes below `o`. The only way to access attributes is to use the `attribute` axis explicitly.



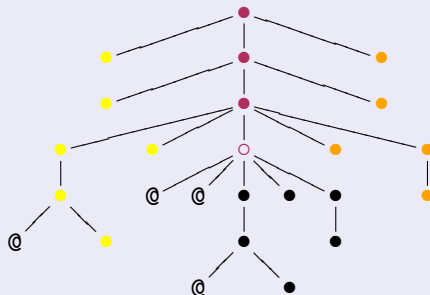
XPath axes: descendant, ancestor, self

XPath axes: descendant, ancestor, self



- In a sense, descendant and ancestor represent the transitive closures of child and parent, respectively.

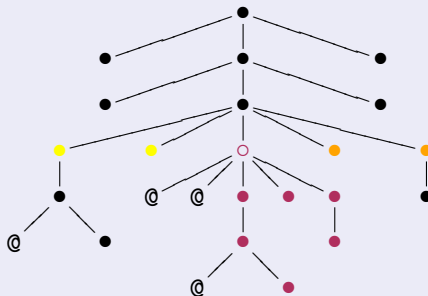
XPath axes: preceding, following, ancestor-or-self



- **Note:** In the serialized XML document, nodes in the preceding (following) axis appear completely before (after) the context node.

XPath axes: preceding-sibling,  
following-sibling, descendant-or-self

XPath axes: preceding-sibling, following-sibling,  
descendant-or-self



# XPath axes: Examples (1)

In these first examples, there is a single initial context node, *i.e.*, a context node sequence of length 1: the root element `a`.

- Here, we set the node test *nt* to simply `node()` which means to *not* filter any nodes selected by the axis.

## XPath example

```
(<a b="0">  
  <c d="1"><e>f</e></c>  $\Rightarrow$  (<c d="1"><e>f</e></c>,  
  <g><h/></g>  
  </a>)/child::node()
```

# XPath axes: Examples (2)

## XPath example

```
(<a b="0">
  <c d="1"><e>f</e></c>
  <g><h/></g>
</a>)/attribute::node() ⇒ attribute b { "0" }
```

## XPath example

```
(<a b="0">
  <c d="1"><e>f</e></c>
  <g><h/></g>
</a>)/descendant::node() ⇒ (<c d="1"><e>f</e></c>,
  <e>f</e>,
  text { "f" },
  <g><h/></g>,
  <h/>
)
```

## XPath axes: Examples (3)

### XPath example

```
(<a b="0">  
  <c d="1"><e>f</e></c>  
  <g><h/></g>  
</a>)/child::node()/child::node()      =>      (<e>f</e>,  
                                              <h/>  
                                              )
```

### Notes:

- If an extracted node has no suitable XML representation by itself, XQuery serializes the result using the XQuery node constructor syntax, e.g.,

*attribute b { "0" }    or    text { "f" } .*

- Nodes are serialized showing their content. This does *not* imply that all of the content nodes have been selected by the XPath expression!





# XPath results: Order & duplicates

## XPath Semantics

The result node sequence of any XPath navigation is returned in **document order** with **no duplicate nodes** (remember: node identity).

### Examples:

#### Duplicate nodes are removed in XPath results ...

```

(<a b="0">
  <c d="1"><e>f</e></c>
  <g><h/></g>
</a>)/child::node()/parent::node()
    ⇒
    <a>
    ...
    </a>
  
```

```

(<a><b/><c/><d/>
</a>)/child::node()/following-sibling::node()
    ⇒
    (<c/>,
    <d/>
    )
  
```

# XPath: Results in document order

XPath: context node sequence of length  $> 1$

$(\langle a \rangle \langle b \rangle \langle c \rangle \langle /a \rangle, \langle d \rangle \langle e \rangle \langle f \rangle \langle /d \rangle) / \text{child}::\text{node}() \Rightarrow (\langle b \rangle, \langle c \rangle, \langle e \rangle, \langle f \rangle)$

## Note:

- The XPath document order semantics require  $\langle b \rangle$  to occur before  $\langle c \rangle$  and  $\langle e \rangle$  to occur before  $\langle f \rangle$ .
  - ▶ The result  $(\langle e \rangle, \langle f \rangle, \langle b \rangle, \langle c \rangle)$  would have been OK as well.
  - ▶ In contrast, the result  $(\langle b \rangle, \langle e \rangle, \langle c \rangle, \langle f \rangle)$  is *inconsistent* with respect to the order of nodes from *separate* trees!



## XPath: Node test

Once an XPath step arrives at a sequence of nodes, we may apply a **node test** to filter nodes based on **kind** and **name**.

### XPath node test

Kind Test	Semantics
<code>node()</code>	let any node pass
<code>text()</code>	preserve text nodes only
<code>comment()</code>	preserve comment nodes only
<code>processing-instruction()</code>	preserve processing instructions
<code>processing-instruction(<i>p</i>)</code>	preserve processing instructions of the form <code>&lt;?<i>p</i> ... ?&gt;</code>
<code>document-node()</code>	preserve the (invisible) document root node

## XPath: Name test

A node test may also be a **name test**, preserving only those element or attribute nodes with matching names.

### XPath name test

Name Test	Semantics
<i>name</i>	preserve <u>element</u> nodes with tag <i>name</i> only (for attribute axis: preserve attributes)
*	preserve <u>element</u> nodes with arbitrary tag names (for attribute axis: preserve attributes)

**Note:** In general we will have  $cs/ax::* \subseteq cs/ax::node()$ .



# XPath: Node test example

✎ Retrieve all attributes named `id` from this XML tree:

```
<a id="0">  
  <b><c id="1"><d id="2"/></c>  
    <c id="3"/>  
  </b>  
  <e di="X" id="4">f</e>  
</a>
```

A solution

## XPath: Node test example

Collect and concatenate all text nodes of a tree

```
string-join(<a><b>A<c>B</c></b>  
            <d>C</d>  
            </a>/descendant-or-self::node()/child::text()  
            , "")
```

- The XQuery **builtin function** `string-join` has signature  
`string-join(string*, string) as string` .

Equivalent: compute the *string value* of node a

```
string(<a><b>A<c>B</c></b>  
      <d>C</d>  
      </a>) ⇒ "ABC"
```

## XPath: Ensuring order is not for free

The strict XPath requirement to construct a result in document order may imply **sorting effort** depending on the actual XPath implementation strategy used by the processor.

```
(<x>
  <x><y id="0"/></x>
  <y id="1"/>
</x>)/descendant-or-self::x/child::y      ⇒      (<y id="0"/>,
  <y id="1"/>)
```

- In many implementations, the `descendant-or-self::x` step will yield the context node sequence `(<x>...</x>, <x>...</x>)` for the `child::y` step.
- Such implementations thus will typically extract `<y id="1"/>` before `<y id="0"/>` from the input document.

# XPath: Predicates

The optional third component of a step formulates a list of **predicates**  $[p_1] \cdots [p_n]$  against the nodes selected by an axis.

## XPath predicate evaluation

Predicates have higher precedence than the XPath step operator  $/$ ,  
i.e.:



$$cs/step[p_1][p_2] \equiv cs/((step[p_1])[p_2])$$

The  $p_i$  are evaluated left-to-right for each node in turn. In  $p_i$ , the **current context node**<sup>26</sup> is available as `'.'`.

---

<sup>26</sup> *Context item*, actually: predicates may be applied to sequences of arbitrary items.



## XPath: Predicates

An XPath predicate  $p_i$  may be *any* XQuery expression evaluating to some value  $v$ . To finally evaluate the predicate, XQuery computes the **effective Boolean value**  $ebv(v)$ .

### Effective Boolean value

Value $v^{27}$	$ebv(v)$
()	false()
0, NaN	false()
""	false()
false()	false()
$x$	true()
$(x_1, x_2, \dots, x_n)$	true()

<sup>27</sup>Item  $x \notin \{0, "", \text{NaN}, \text{false}()\}$ , items  $x_i$  arbitrary. Builtin function `boolean(item*)` as `boolean` also computes the effective Boolean value.

# XPath: Predicate example

## Select all elements with an id attribute

<pre>(&lt;a id="0"&gt;   &lt;b&gt;&lt;c id="1"/&gt;&lt;/b&gt;   &lt;b&gt;&lt;c&gt;&lt;b/&gt;&lt;/c&gt;&lt;/b&gt;   &lt;d id="2"&gt;e&lt;/d&gt; &lt;/a&gt;)/descendant-or-self::*[./attribute::id]</pre>	$\Rightarrow$	<pre>(&lt;a id="0"&gt;   ...   &lt;/a&gt;,   &lt;c id="1"/&gt;,   &lt;d id="2"&gt;e&lt;/d&gt; )</pre>
---	---------------	---

## Select all elements with a b grandchild element

<pre>(&lt;a id="0"&gt;   &lt;b&gt;&lt;c id="1"/&gt;&lt;/b&gt;   &lt;b&gt;&lt;c&gt;&lt;b/&gt;&lt;/c&gt;&lt;/b&gt;   &lt;d id="2"&gt;e&lt;/d&gt; &lt;/a&gt;)/descendant-or-self::*[./child:*/child::b]</pre>	$\Rightarrow$	<pre>&lt;b&gt;   &lt;c&gt;&lt;b/&gt;&lt;/c&gt; &lt;/b&gt;</pre>
--	---------------	---

- **Note: Existential semantics** of path predicates.

## XPath: Predicate example

### ✎ How to select all non-leaf elements of a tree?

You may use the builtin function `not (item*)` as `boolean` which computes the inverted effective Boolean value, *i.e.*,

$$\text{not}(v) \equiv \neg \text{boolean}(v).$$

### A solution

## XPath: Predicates and atomization

In XQuery, if any item  $x$ —atomic value or node—is used in a context where a value is required, **atomization** is applied to convert  $x$  into an atomic value.

- Nodes in value contexts commonly appear in XPath predicates.  
Consider:

### Value comparison in a predicate

<pre>(&lt;a&gt;   &lt;b&gt;42&lt;/b&gt;   &lt;c&gt;&lt;d&gt;42&lt;/d&gt;&lt;/c&gt;   &lt;e&gt;43&lt;/e&gt; &lt;/a&gt;)/descendant-or-self::*[. eq 42]</pre>	$\Rightarrow$	<pre>(&lt;b&gt;42&lt;/b&gt;,  &lt;c&gt;&lt;d&gt;42&lt;/d&gt;&lt;/c&gt;,  &lt;d&gt;42&lt;/d&gt; )</pre>
---	---------------	--

# Atomization

## Atomization

**Atomization** turns a sequence  $(x_1, \dots, x_n)$  of items into a sequence of atomic values  $(v_1, \dots, v_n)$ :

- ① If  $x_i$  is an atomic value,  $v_i \equiv x_i$ ,
  - ② if  $x_i$  is a node,  $v_i$  is the *typed value*<sup>28</sup> of  $x_i$ .
- The XQuery builtin function `data(item*) as anyAtomicType*` may be used to perform atomization explicitly (rarely necessary).

---

<sup>28</sup>Remember: the *typed value* is equal to the *string value* if  $x_i$  has not been validated. In this case,  $v_i$  has type `untypedAtomic`.

# XPath: Predicates and atomization

## Atomization (and casting) made explicit

```
<a>
  <b>42</b>
  <c><d>42</d></c>
  <e>43</e>
</a>/descendant-or-self::*[data(.) cast as double
                             eq
                             42 cast as double]
```

- **Note:** the value comparison operator **eq** is witness to the value context in which '.' is used in this query.
- For the context item `<c><d>42</d></c>` (a non-validated node), `data(.)` returns "42" of type `untypedAtomic`.

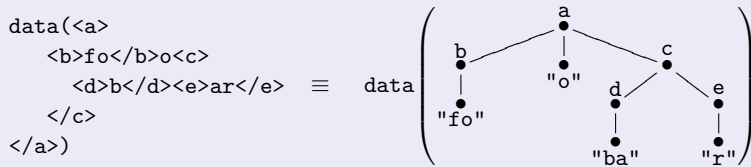
## Atomization and subtree traversals

Since atomization of nodes is pervasive in XQuery expression evaluation, *e.g.*, during evaluation of

- arithmetic and comparison expressions,
- function call and return,
- explicit sorting (`order by`),

efficient **subtree traversals** are of prime importance for any implementation of the language:

Applying `data()` to a node and its subtree:



## XPath: Positional access

Inside a predicate  $[p]$  the current context item is `'.'`.

- An expression may also access the **position** of `'.'` in the context sequence via `position()`. The first item is located at position 1.
- Furthermore, the position of the **last** context item is available via `last()`.

### Positional access

$$\begin{aligned}(x_1, x_2, \dots, x_n)[\text{position()} \text{ eq } i] &\Rightarrow x_i \\ (x_1, x_2, \dots, x_n)[\text{position()} \text{ eq last}()] &\Rightarrow x_n\end{aligned}$$

A predicate of the form  $[\text{position()} \text{ eq } i]$  with  $i$  being any XQuery expression of numeric type, may be abbreviated by  $[i]$ .



## XPath: Positional access example

✎ Predicates `[.]` bind stronger than `/`



Given the XML tree below as context *cs*, what is the result of evaluating

```
(cs/descendant-or-self::node()/child::x)[2]
```

vs.

```
cs/descendant-or-self::node()/child::x[2] ?
```

```
<a>
  <b><x id="1"/></b>
  <d><x id="2"/><x id="3"/></d>
</a>
```

Solution

## XPath: Positional access example

Predicates are evaluated after step and node test

Given the XML tree below as context *cs*, what is the result of evaluating

`cs/descendant::*[2]`

vs.

`cs/descendant::x[2] ?`

```
<a>
  <b><x id="1"/></b>
  <d><x id="2"/><x id="3"/></d>
</a>
```

Solution

# XPath: Predicate evaluation order

Remember: predicates are evaluated left to right

$$\begin{aligned}(1,2,3,4)[. \text{ gt } 2][2] &\equiv ((1,2,3,4)[. \text{ gt } 2])[2] \\ &\neq (1,2,3,4)[2][. \text{ gt } 2]\end{aligned}$$

## XPath: The context item '.'

As a useful generalization, XPath makes the **current context item** '.' available in each **step** (not only in predicates).

### XPath steps (/) and the context item

In the expression

$$cs/e$$

expression  $e$  will be evaluated with '.' set to each item in the context sequence  $cs$  (in order). The resulting sequence is returned.<sup>2930</sup>

---

<sup>29</sup>Remember: if  $e$  returns nodes ( $e$  has type `node*`), the resulting sequence is sorted in document order with duplicates removed.

<sup>30</sup>Compare this with the expression  $\text{map } (\lambda . \rightarrow e) cs$  in functional programming languages.

# XPath: Using the context item

## Accessing '.'

$(\langle a \rangle 1 \langle /a \rangle, \langle b \rangle 2 \langle /b \rangle, \langle c \rangle 3 \langle /c \rangle) / (. + 42) \Rightarrow (43.0, 44.0, 45.0)$

$(\langle a \rangle 1 \langle /a \rangle, \langle b \rangle 2 \langle /b \rangle, \langle c \rangle 3 \langle /c \rangle) / \text{name}(\cdot) \Rightarrow ("a", "b", "c")$

$(\langle a \rangle 1 \langle /a \rangle, \langle b \rangle 2 \langle /b \rangle, \langle c \rangle 3 \langle /c \rangle) / \text{position}() \Rightarrow (1, 2, 3)$

$(\langle a \rangle \langle b \rangle \langle /a \rangle) / (./\text{child::b}, \cdot) \Rightarrow (\langle a \rangle \langle b \rangle \langle /a \rangle, \langle b \rangle)$

## Evaluate the following

❶ `cs/descendant-or-self::node()/count(./descendant::node())`

❷ `cs/descendant-or-self::node()/count(./ancestor::*)`

with  $cs \equiv \left( \begin{array}{c} \langle a \rangle \\ \langle b \ c="0"/ \rangle \\ \langle d \rangle \langle e \rangle f \langle /e \rangle \langle /d \rangle \\ \langle /a \rangle \end{array} \right) \cdot$

# Combining node sequences

## Node sequence combinations

Sequences of nodes (e.g., the results of XPath location step) may be combined via

```
| union31  
intersect  
except .
```

These operators **remove duplicate nodes** based on identity and return their result in **document order**.

- **Note:** Introduced in the XPath context because a number of useful navigation idioms are based on these operators.

---

<sup>31</sup> | and `union` are synonyms

# Navigation idioms (1)


Selecting all x children and attributes of context node

```
cs/(./child::x | ./attribute::x)
```

Select all siblings of context node

```
cs/(./preceding-sibling::node() | ./following-sibling::node())  
or  
cs/(./parent::node()/child::node() except .)
```

Select context node + all its siblings

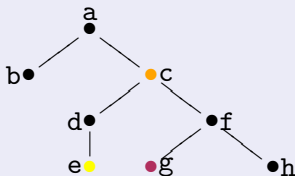
```
cs/(./parent::node()/child::node() |  . )  
*
```

Why is (\*) required?

## Navigation idioms (2)

### First common ancestor (fca)

Compute the first common ancestor (fca) of two contexts,  $cs_0$  and  $cs_1$ , in the same tree:



$(cs_0/ancestor::^* \text{ intersect } cs_1/ancestor::^*)[last()]$

✍ What is going on here?

And: Will this work for non-singleton  $cs_{0,1}$ ?



## XPath: Simulate `intersect` and `except`

In earlier versions of XPath (1.0), the following expressions could simulate `intersect` and `except` of two node sequences  $cs_{0,1}$ :<sup>32</sup>

### Simulate `intersect` and `except`

$$cs_0 \text{ intersect } cs_1 \equiv cs_0[\text{count}(. \text{ } cs_1) \text{ eq count}(cs_1)]$$
$$cs_0 \text{ except } cs_1 \equiv cs_0[\text{count}(. \text{ } cs_1) \text{ ne count}(cs_1)]$$

---

<sup>32</sup>XQuery builtin operators `eq` and `ne` compare two single items for equality and inequality, respectively.

## XPath: Abbreviations

Since XPath expressions are pervasive in XQuery, query authors commonly use the succinct **abbreviated XPath syntax** to specify location steps.

### Abbreviated XPath syntax

Abbreviation	Expansion
<i>nt</i>	<code>child::<i>nt</i></code>
@	<code>attribute::</code>
..	<code>parent::node()</code>
//	<code>/descendant-or-self::node()/</code>
/ <sup>33</sup>	<code>root(.)</code>
<i>step</i> <sup>33</sup>	<code>./<i>step</i></code>

<sup>33</sup>At the beginning of a path expression.

# XPath: Abbreviations

## XPath abbreviation examples

Abbreviation	Expansion
<code>a/b/c</code>	<code>./child::a/child::b/child::c</code>
<code>a//@id</code>	<code>./child::a/descendant-or-self::node()/attribute::id</code>
<code>//a</code>	<code>root()/descendant-or-self::node()/child::a</code>
<code>a/text()</code>	<code>./child::a/child::text()</code>

## XPath abbreviation quiz

What is the expansion (and semantics) of

`a/( * @* )`    and    `a[ * ]`    ?

# XPath: Abbreviations

**NB:** Use of these abbreviations may lead to confusion and surprises!



## Abbreviations + predicates = confusion

$$cs//c[1] \not\equiv cs/descendant-or-self::c[1]$$

Evaluate both path expressions against

$$cs = \left( \begin{array}{l} \langle a \rangle \\ \quad \langle b \rangle \langle c \text{ id}="0"/ \rangle \langle c \text{ id}="1"/ \rangle \langle /b \rangle \\ \quad \langle d \rangle \langle c \text{ id}="2"/ \rangle \langle /d \rangle \\ \quad \langle c \text{ id}="3"/ \rangle \\ \langle /a \rangle \end{array} \right)$$

## More XPath weirdness

`cs/(/)/(/)`

`parent::text()`

`attribute::comment()`

## Part IX


# XSLT—Presentation of XML Documents

# Outline of this part

## 25 XSLT—An XML Presentation Processor

- Separating content from style
- XSL Stylesheets
- XSLT Templates
- Examples
- Conflict Resolution and Modes in XSLT
- More on XSLT

# XSLT—An XML Presentation Processor

- XML in itself is quite weak when it comes to **data presentation**.
  - An XML processor can derive nothing but the tree structure of the XML data.
    - ▶ XML by itself has **no semantic meaning**.
    - ▶ XML markup (usually) does **not include formatting information**.
    - ▶ The “vanilla” XML tree structure might not be the appropriate form of presentation for all types of data.
  - **XSLT (Extensible Style Sheet Language/Transformations)**  
 <http://www.w3.org/TR/xslt> introduces a separate **presentation processor** that maps XML trees into
    - ① other XML trees (e.g., XHTML),
    - ② instructions for various output formatters (PDF writers, ...)
- N.B.** ① makes XSLT a general XML → XML transformer.

# Separating content from style

Contrary to when style information is hard-coded into the content, **separation of style from content** allows for the same data to be presented in many ways:

- ➊ **Reuse** fragments of data  
(same contents looks different depending on context),
- ➋ **multiple output formats**  
(media [online, paper], sizes, devices [workstation, handheld]),
- ➌ styles tailored to **reader's preference**  
(accessibility issues, audio rendering),
- ➍ **standardized** styles  
(corporate identity, web site identity),
- ➎ **freedom from style**  
(do not bother tech writers with layout issues).



- An XSL stylesheet describes XML presentation using two basic categories of techniques:
  - ① Optional **transformation of XML document tree into another structure**,
  - ② specification of **presentation properties to associate to each of the various parts** of the transformed tree.

## XSL vs. CSS

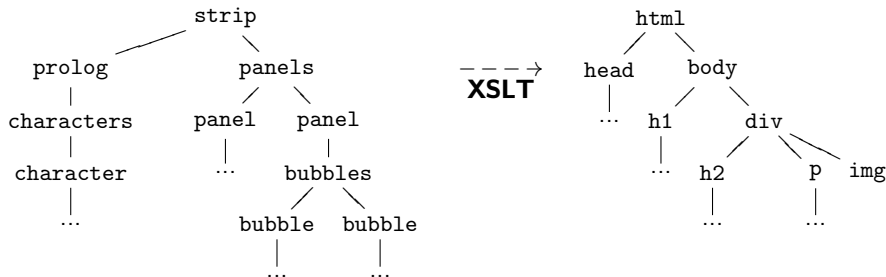
How does CSS (Cascading Style Sheet Language) compare to XSL as described until now?

- **Transformation?**
  - ▶ Generation of **(new) constant content**,
  - ▶ **suppress** content,
  - ▶ **moving** subtrees (e.g., *swap* day/month in a date),
  - ▶ **copying** subtrees (e.g., *copy* section titles into tables of contents),
  - ▶ **sorting**,
  - ▶ general transformations that **compute new from given content**.

## ● Presentation properties?

- ▶ General page (or screen) **layout**,
- ▶ assign content to **“containers”** (e.g., lists, paragraphs),
- ▶ **formatting properties** (e.g, spacing, margins, alignment, fonts) for each such container.

**Example:** XML → XHTML transformation via XSLT:



# XSL Stylesheets

- An **XSL stylesheet** defines a set of **templates** (“tree patterns and actions”).

Each template ...

- ❶ **matches** specific elements in the XML doc tree, and then
  - ❷ **constructs** the contribution that the elements make to the transformed tree.
- XSL is **an application of XML** itself:
    - ▶ Each XSL stylesheet is an XML document,
    - ▶ elements with a **name prefix**<sup>34</sup> `xs1:` are part of the XSLT language,
    - ▶ non-“`xs1:`” elements are used to construct the transformed tree.

---

<sup>34</sup>More correctly: elements in the **namespace**

<http://www.w3.org/1999/XSL/Transform>. For details on namespaces, see <http://www.w3.org/TR/REC-xml-names>.

**Example:** Transform text markup into HTML style paragraph and emphasis tags:

style.xsl

```
1 <?xml version="1.0"?>
2 <xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform" version="1.0">
3
4 <xsl:template match="para">
5   <p><xsl:apply-templates/></p>
6 </xsl:template>
7
8 <xsl:template match="emphasis">
9   <i><xsl:apply-templates/></i>
10 </xsl:template>
11
12 </xsl:stylesheet>
```

input.xml

```
1 <?xml version="1.0"?>
2 <para>This is a <emphasis>test</emphasis>.</para>
```

output.xml

```
1 <?xml version="1.0" encoding="UTF-8"?>
2 <p>This is a <i>test</i>.</p>
```

**N.B.** Note how XSLT acts like a **tree transformer** in this simple example.

# XSLT templates

```
<xsl:template match="e">
  cons
</xsl:template>
```

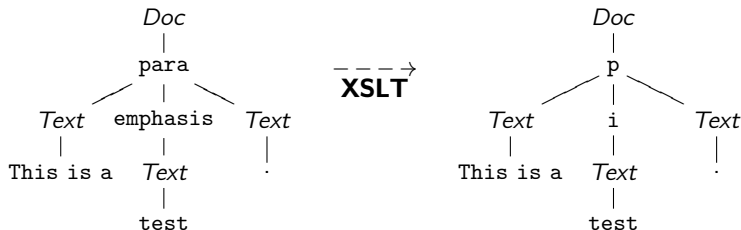
- *e* is an **XPath expression**, selecting the nodes in the document tree XSLT will apply the template to,
- *cons* is the **result constructor**, describing the transformation result that the XSLT processor will produce for the nodes selected by *e*.

**N.B.** “xsl:” elements in *cons* will be interpreted by the XSLT processor.

`<xsl:apply-templates/>` applies the template matching process **recursively to all child nodes** of the matched node.

## Applying the template ...

The actual tree transformation in our previous example goes like this:



Something else must be going on here:

- 1 The *Text* nodes have *automatically* been copied into the result tree.
- 2 How could the *para* and *emphasis* elements match anyway?  
(The XPath patterns for both templates used *relative* paths expressions.)

## Default templates

Each XSLT stylesheet contains two **default templates** which

- 1 **copy Text and Attr (attribute) nodes** into the result tree:

```
<xsl:template match="text()|@*">
    <xsl:value-of select="self::node()"/>
</xsl:template>
```

- 2 **recursively drive the matching process**, starting from the document root:

```
<xsl:template match="/*">
    <xsl:apply-templates/>
</xsl:template>
```

`<xsl:value-of select="e"/>` copies those nodes into the result tree that are reachable by the XPath expression `e` (context node is the matched node).

The default templates may be **overridden**.

## Overriding default XSLT templates

What would be the effect of applying the following XSLT stylesheet?

style.xsl

```
1 <?xml version="1.0"?>
2 <xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform" version="1.0">
3
4 <xsl:template match="text()">foo</xsl:template>
5
6 <xsl:template match="para">
7   <p><xsl:apply-templates/></p>
8 </xsl:template>
9
10 <xsl:template match="emphasis">
11   <i><xsl:apply-templates/></i>
12 </xsl:template>
13
14 </xsl:stylesheet>
```

## More XSLT defaults

XSLT contains the following additional default template. Explain its effect.

```
<xsl:template match="processing-instruction()|comment()"/>
```



## Intermediate summary

XSLT Instruction	Effect
<pre>&lt;xsl:template match="e"&gt;   cons &lt;/xsl:template&gt;</pre>	Replace nodes matching path expression <i>e</i> by <i>cons</i> .
<pre>&lt;xsl:apply-templates select="e"/&gt;</pre>	Initiate template matching for those nodes returned by path expression <i>e</i> (default: path <i>e</i> = <code>child::node()</code> ).
<pre>&lt;xsl:value-of select="e"/&gt;</pre>	Returns the ( <i>string value</i> <sup>35</sup> of the) result of XPath expression <i>e</i> .

<sup>35</sup>Read: The *string value* of an XML element node is the concatenation of the contents of all *Text* nodes in the subtree below that element (in document order).

# More XSLT features

## ✎ Recursion in XSLT

Explain the effect of the following XSLT stylesheet

```
<?xml version="1.0"?>
<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
                version="1.0">
  <xsl:template match="foo">
    <xsl:apply-templates select="/" />
  </xsl:template>
</xsl:stylesheet>
```

## ✎ What would be the effect of applying an empty stylesheet?

```
<?xml version="1.0"?>
<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
                version="1.0"/>
```

## Example: Dilbert comic strips ...

Transform a DilbertML document into an HTML representation that reflects the comic strip's story:

- From the `prolog`, generate the HTML header, title, heading, copyright information.
- From `characters`, generate an unordered HTML list (`ul`) of all featured comic characters.
- For all `panels`, reproduce the `scene` as well as all spoken `bubbles`, indicating who is speaking to whom (if available).

### Note:

`<xsl:if test="p"/> cons </xsl:if>` reproduces `cons` in the result tree, if the XPath predicate `p` evaluates to true.<sup>36</sup>

---

<sup>36</sup>Remember from XPath: an empty node sequence is interpreted as false, a non-empty sequence as true.

## dilbert.xml

```
1 <?xml version="1.0"?>
2 <xsl:stylesheet version="1.0"
3   xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
4
5 <!-- Generate document head and body, insert prolog information -->
6 <xsl:template match="/">
7   <html>
8     <head>
9       <title>
10        <xsl:value-of select="/strip/prolog/series"/>
11      </title>
12    </head>
13    <body>
14      <h1> <xsl:value-of select="/strip/prolog/series"/>
15    </h1>
16    <p>A comic series by
17      <xsl:value-of select="/strip/prolog/author"/>,
18      copyright (C)
19      <xsl:value-of select="/strip/@year"/> by
20      <xsl:value-of select="/strip/@copyright"/>
21    </p>
22    <xsl:apply-templates/>
23  </body>
```

```
24     </html>
25 </xsl:template>
26
27 <!-- The next 2 templates generate the
28      "Featured Characters" bullet list -->
29 <xsl:template match="characters">
30     <h2> Featured Characters </h2>
31     <ul>
32         <xsl:apply-templates/>
33     </ul>
34 </xsl:template>
35
36 <xsl:template match="character">
37     <li> <xsl:value-of select="."/> </li>
38 </xsl:template>
39
40 <!-- Reproduce the panel and the scene it displays -->
41 <xsl:template match="panel">
42     <h3> Panel <xsl:value-of select="@no"/> </h3>
43     <p> <xsl:value-of select="scene"/> </p>
44     <xsl:apply-templates select="bubbles"/>
45 </xsl:template>
46
```

```
47 <!-- Reproduce spoken text, indicating tone and
48      who is speaking to whom -->
49 <xsl:template match="bubble">
50   <p> <xsl:value-of select="id(@speaker)"/> speaking
51     <xsl:if test="@to">
52       to <xsl:value-of select="id(@to)"/>
53     </xsl:if>
54     <xsl:if test="@tone">
55       (<xsl:value-of select="@tone"/>)
56     </xsl:if>
57     :<br/>
58     <em>
59       <xsl:value-of select="."/>
60     </em>
61   </p>
62 </xsl:template>
63
64 <!-- Suppress all other text/attributes -->
65 <xsl:template match="text()|@*" />
66
67 </xsl:stylesheet>
```

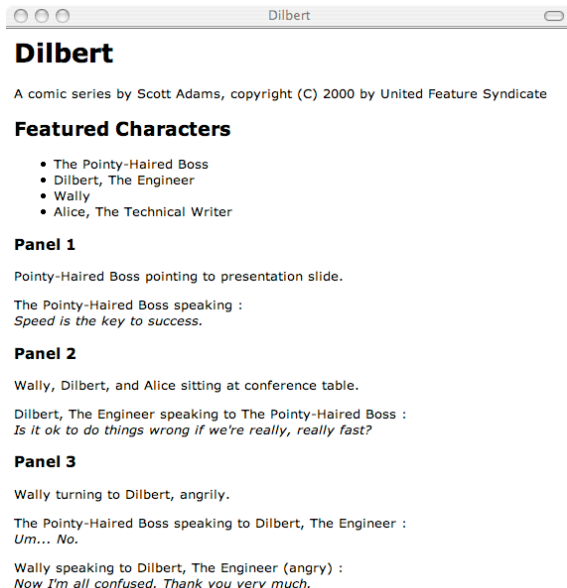
## dilbert.html

```
1 <html>
2 <head> <title>Dilbert</title> </head>
3 <body>
4   <h1>Dilbert</h1>
5   <p>A comic series by Scott Adams, copyright
6     (C) 2000 by United Feature Syndicate </p>
7   <h2> Featured Characters </h2>
8   <ul>
9     <li>The Pointy-Haired Boss</li>
10    <li>Dilbert, The Engineer</li>
11    <li>Wally</li>
12    <li>Alice, The Technical Writer</li>
13  </ul>
14  <h3> Panel 1</h3>
15  <p>Pointy-Haired Boss pointing to presentation slide.
16  </p>
17  <p>The Pointy-Haired Boss speaking : <br>
18    <em>Speed is the key to success.</em>
19  </p>
20  <h3> Panel 2</h3>
21  <p>Wally, Dilbert, and Alice sitting at conference table.
22  </p>
23  <p>Dilbert, The Engineer speaking
```

```
24      to The Pointy-Haired Boss : <br>
25      <em>Is it ok to do things wrong if
26          we're really, really fast?</em>
27  </p>
28  <h3> Panel 3</h3>
29  <p>Wally turning to Dilbert, angrily.
30  </p>
31  <p>The Pointy-Haired Boss speaking
32      to Dilbert, The Engineer : <br>
33      <em>Um... No.</em>
34  </p>
35  <p>Wally speaking
36      to Dilbert, The Engineer (angry) : <br>
37      <em>Now I'm all confused. Thank you very much.</em>
38  </p>
39 </body>
40 </html>
```



# Screenshot of Mozilla rendering file `dilbert.html`:



# Conflict Resolution and Modes in XSLT

- Note that for each node visited by the XSLT processor (cf. default template ②), **more than one template might yield a match**.
- XSLT assigns a **priority** to each template. The more specific the template pattern, the higher the priority:

```
<xsl:template match="e"> cons </xsl:template>
```

Pattern <i>e</i>	Priority
*	−0.5
<i>ns</i> :*	−0.25
element/attribute name	0
any other XPath expression	0.5

- Example:**

Priority of `author` is 0, priority of `/strip/prolog/author` is 0.5.

- Alternatively, make priority explicit:

```
<xsl:template priority="p" ...>
```

## Context

Quite often, an XSLT stylesheet wants to be **context-aware**.

- Since the XSLT priority mechanism is *not* dynamic, this can cause problems.

**Example:** Transform the following XML document (sectioned text with cross references) into XHTML:

### self-ref.xml

```
1 <section id="intro">
2   <title>Introduction</title>
3   <para> This section is self-referential: <xref to="intro">. </para>
4 </section>
```

We want to generate XHTML code that looks somewhat like this:

### self-ref.html

```
1 <h1>Introduction</h1>
2 <p> This section is self-referential: <em>Introduction</em>. </p>
```



The section title needs to be processed twice, once to produce the heading and once to produce the cross reference.

The “obvious” XSLT stylesheet produces erroneous output:

buggy-self-ref.xsl

```
1 <?xml version="1.0"?>
2 <xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
3   version="1.0">
4
5 <xsl:template match="title">
6   <h1><xsl:apply-templates/></h1>
7 </xsl:template>
8
9 <xsl:template match="para">
10  <p><xsl:apply-templates/></p>
11 </xsl:template>
12
13 <xsl:template match="xref">
14   <xsl:apply-templates select="id(@to)/title"/>
15 </xsl:template>
16
17 </xsl:stylesheet>
```

buggy-output.html

```
1 <h1>Introduction</h1>
2 <p> This section is self-referential: <h1>Introduction</h1>. </p>
```

# XSLT modes

- We need to make the processing of the `title` element aware of the context (or **mode**) it is used in: inside an `xref` or not.
- This is a job for **XSLT modes**.
  - ▶ In `<xsl:apply-templates>` switch to a certain mode *m* depending on the context:

```
<xsl:apply-templates mode="m" .../>
```

- ▶ After mode switching, only `<xsl:template>` instructions with a `mode` attribute of value *m* will match:

```
<xsl:template mode="m" .../>
```

- ▶ As soon as `<xsl:apply-templates mode="m" .../>` has finished matching nodes, the previous mode (if any) is restored.

## self-ref.xsl

```
1 <xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
2     version="1.0">
3
4 <xsl:template match="title">
5     <h1><xsl:apply-templates/></h1>
6 </xsl:template>
7
8 <xsl:template match="title" mode="ref">
9     <em><xsl:apply-templates/></em>
10 </xsl:template>
11
12 .
13 .
14
15 <xsl:template match="xref">
16     <xsl:apply-templates select="id(@to)/title" mode="ref"/>
17 </xsl:template>
18
19 </xsl:stylesheet>
```

## output.html

```
1 <h1>Introduction</h1>
2 <p> This section is self-referential: <em>Introduction</em>. </p>
```

# More on XSLT

XSLT Instruction	Effect
<code>xsl:choose</code> , <code>xsl:when</code>	switch statement (ala C)
<code>xsl:call-template</code>	explicitly invoke a (named) template
<code>xsl:for-each</code>	replicate result construction for a sequence of nodes
<code>xsl:import</code>	import instructions from another stylesheet
<code>xsl:output</code>	influence XSLT processor's output behaviour
<code>xsl:variable</code>	set/read variables

- For a complete XSLT reference, refer to [W3C http://www.w3.org/TR/xslt](http://www.w3.org/TR/xslt)
- Apache's Cocoon is an XSLT-enabled web server (see <http://xml.apache.org/cocoon/>).

# Part X

## XQuery—Querying XML Documents



# Outline of this part

## 26 XQuery—Declarative querying over XML documents

- Introduction
- Preliminaries

## 27 Iteration (FLWORs)

- For loop
- Examples
- Variable bindings
- `where` clause
- FLWOR Semantics
- Variable bindings
- Constructing XML Fragments
- User-Defined Functions

# XQuery—Introduction

- XQuery is a truly **declarative** language specifically designed for the purpose of querying XML data.
- As such, XML assumes the role that SQL occupies in the context of relational databases.
- XQuery exhibits properties known from database (DB) languages as well as from (functional) programming (PL) languages.
- The language is designed and formally specified by the W3C XQuery Working Group (W3C <http://www.w3.org/XML/XQuery/>).
  - ▶ The first working draft documents date back to February 2001. The XQuery specification is expected to become a W3C Recommendation during the summer of 2006.
  - ▶ Members of the working group include Dana Florescu<sup>DB</sup>, Ioana Manolescu<sup>DB</sup>, Phil Wadler<sup>PL</sup>, Mary Fernández<sup>DB+PL</sup>, Don Chamberlin<sup>DB,37</sup>, Jérôme Siméon<sup>DB</sup>, Michael Rys<sup>DB</sup>, and many others.

---

<sup>37</sup>Don is the “father” of SQL.

# 1/2 Programming Language, 1/2 Query Language

XQuery is a hybrid exhibiting features commonly found in **programming** as well as **database query** languages:

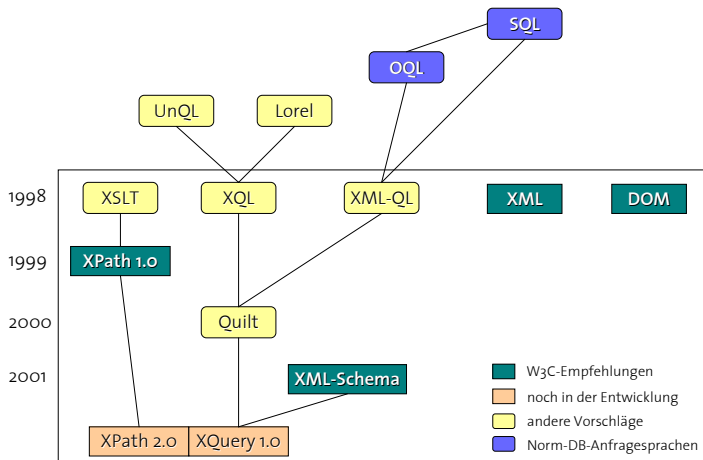
- **Programming language** features:

- ▶ explicit iteration and variable bindings (`for...in`, `let...in`)
- ▶ recursive, user-defined functions
- ▶ regular expressions, strong [static] typing
- ▶ ordered sequences (much like lists or arrays)

- **Database query language** features:

- ▶ filtering
- ▶ grouping, joins } expressed via nested `for` loops

# History of XQuery



[illustration © C. Türker]

# XQuery—Preliminaries

- **Remember:** XPath is part of XQuery (as a sublanguage).
- Some constructs that have not previously been discussed, yet are not within the core of our focus on XQuery include:
  - ▶ **Comparisons:** any XQuery expression evaluates to a **sequence** of items. Consequently, many XQuery concepts are prepared to accept sequences (as opposed to single items).

## General Comparisons

The **general comparison**  $e_1 \theta e_2$  with

$$\theta \in \{=, !=, <, <=, >=, >\}$$

yields `true()` if *any* of the items in the sequences  $e_{1,2}$  compare true (*existential semantics*).

# Comparisons

## General comparison examples

```

(1,2,3) > (2,4,5)  ⇒ true()
  (1,2,3) = 1      ⇒ true()
    () = 0         ⇒ false()
      2 <= 1       ⇒ false()
    (1,2,3) != 3   ⇒ true()
  (1,2) != (1,2)   ⇒ true()
not((1,2) = (1,2)) ⇒ false()

```



## Value comparisons

The six **value comparison operators** eq, ne, lt, le, ge, gt compare *single items by value* (atomization!):

```

      2 gt 1.0      ⇒ true()
<x>42</x> eq <y>42</y> ⇒ true()
    (0,1) eq 0      ⇒ ⚡ (type error)

```

## More on comparisons ...

**Note:** The existential semantics of the general comparison operators may lead to unexpected behavior:



### Surprises

$$(1,2,3) = (1,3) \Rightarrow \text{true()}^a$$
$$("2",1) = 1 \Rightarrow \text{true()} \text{ or } \frac{1}{2} \text{ (impl. dependent)}$$

---

<sup>a</sup>For an *item-by-item* comparison use `deep-equal()`.

# Node comparisons

## Node comparison

... based on *identity* and *document order*:

$e_1$ is $e_2$	nodes $e_{1,2}$ identical?
$e_1 << e_2$	node $e_1$ before $e_2$ ?
$e_1 >> e_2$	node $e_1$ after $e_2$ ?

## Node comparison examples

`<x>42</x> eq <x>42</x>`  $\Rightarrow$  `true()`

`<x>42</x> is <x>42</x>`  $\Rightarrow$  `false()`

`root( $e_1$ ) is root( $e_2$ )`  $\Rightarrow$  nodes  $e_{1,2}$  in same tree?

`let $a := <x><y/></x>`  $\Rightarrow$  `true()`

`return $a << $a/y`




# Working with sequences

XQuery comes with an extensive **library of builtin functions** to perform common computations over sequences:

## Common sequence operations

Function	Example
count	<code>count((0,4,2))</code> $\Rightarrow$ 3
max	<code>max((0,4,2))</code> $\Rightarrow$ 4
subsequence	<code>subsequence((1,3,5,7),2,3)</code> $\Rightarrow$ (3,5,7)
empty	<code>empty((0,4,2))</code> $\Rightarrow$ false()
exists	<code>exists((0,4,2))</code> $\Rightarrow$ true()
distinct-values	<code>distinct-values((4,4,2,4))</code> $\Rightarrow$ (4,2)
to	<code>(1 to 10)[. mod 2 eq 1]</code> $\Rightarrow$ (1,3,5,7,9)

See  <http://www.w3.org/TR/xpath-functions/>.

# Arithmetics

Only a few words on arithmetics—XQuery meets the common expectation here. Points to note:

- ❶ Infix operators: +, −, \*, div, idiv (integer division),
- ❷ operators first **atomize** their operands, then perform **promotion** to a common numeric type,
- ❸ if at least one operand is (), the result is ().

## Examples and pitfalls

`<x>1</x> + 41`  $\Rightarrow$  `42.0`

`() * 42`  $\Rightarrow$  `()`

`(1,2) - (2,3)`  $\Rightarrow$   $\downarrow$  (type error)

`x-42`  $\Rightarrow$  `./child::x-42` (use `x-42`)

`x/y`  $\Rightarrow$  `./child::x/child::y` (use `x div y`)

# XQuery Iteration: FLWORS



- Remember that XPath steps perform **implicit iteration**: in  $cs/e$ , evaluation of  $e$  is iterated with  $'.'$  bound to each item in  $cs$  in turn.
- XPath subexpressions aside, **iteration in XQuery is explicit** via the **FLWOR** (*"flower"*) construct.
  - ▶ The versatile FLWOR is used to express
    - ★ nested iteration,
    - ★ joins between sequences (of nodes),
    - ★ groupings,
    - ★ orderings beyond document order, *etc.*
  - ▶ In a sense, FLWOR assumes the role of the SELECT-FROM-WHERE block in SQL.

# FLWOR: Iteration via `for...in`

## Explicit iteration

Explicit iteration is expressed using the `for...in` construct:<sup>a</sup>

```
for $v [at $p] in e1  
  return e2
```

If  $e_1$  evaluates to the sequence  $(x_1, \dots, x_n)$ , the loop body  $e_2$  is evaluated  $n$  times with variable  $\$v$  bound to each  $x_i$  [and  $\$p$  bound to  $i$ ] in order. The results of these evaluations are concatenated to form a single sequence.

---

<sup>a</sup>The construct '`at $p`' is optional.

# Iteration

## Iteration examples

```
for $x in (3,2,1)
return ($x,"*")
```

 $\Rightarrow$ 

```
(3,"*",2,"*",1,"*")
```

```
for $x in (3,2,1)
return $x,"*"
```

 $\Rightarrow$ 

```
(3,2,1,"*")
```



```
for $x in (3,2,1)
return for $y in ("a","b")
return ($x,$y)
```

 $\Rightarrow$ 

```
(3,"a",3,"b",
2,"a",2,"b",
1,"a",1,"b")
```

## FLWOR: Abbreviations

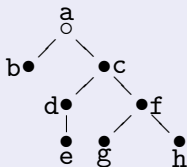
<pre>for \$v<sub>1</sub> in e<sub>1</sub> return   for \$v<sub>2</sub> in e<sub>2</sub> return e<sub>3</sub></pre>	$\equiv$	<pre>for \$v<sub>1</sub> in e<sub>1</sub> for \$v<sub>2</sub> in e<sub>2</sub> return e<sub>3</sub></pre>	$\equiv$	<pre>for \$v<sub>1</sub> in e<sub>1</sub>,   \$v<sub>2</sub> in e<sub>2</sub> return e<sub>3</sub></pre>
--	----------	---	----------	--

# FLWOR: Iteration via `for...in`

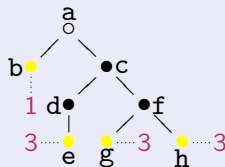
## ✎ Purpose of this query *Q*?

```
max( for $i in cs/descendant-or-self::*[not(*)]
      return count($i/ancestor::*) )
```

### A sample *cs*



### "Annotated" sample *cs*



## Answer

# FLWOR: Iteration via `for...in`

## Return every other item in sequence

These queries both return the items at odd positions in the input sequence `e`:

```
for $i in (1 to count(e)) [. mod 2 eq 1]
return e[$i]
```

```
for $i at $p in e
return if ($p mod 2)
      then e[$p]
      else ()
```

- Remember: `ebv(0) = false()`.

## FLWOR: Variable Binding via `let... :=`

Note that in the examples on the last slide, expression  $e$  is re-evaluated  $\text{count}(e)/2$  times although  $e$  is constant in the loop.

### Variable bindings

The result of evaluating an expression  $e$  may be bound to a variable  $\$v$  via `let`:

```
let $v := e1  
return e2
```

evaluates  $e_2$  with free occurrences of  $\$v$  replaced by  $e$ .

- `for` and `let` clauses may be freely intermixed.



# FLWOR: Variable Binding via `let... :=`

## Iteration vs. variable binding

```
for $x in (3,2,1)
return ($x,"*")    ⇒    (3,"*",2,"*",1,"*")
```

```
let $x := (3,2,1)
return ($x,"*")    ⇒    (3,2,1,"*")
```

## “Every other item” revisited (flip back two slides)

The following hoists the constant `e` out of the loop body:

```
let $seq := e
return for $i at $p in $seq
      return if ($p mod 2)
              then $seq[$p]
              else ()
```

## Adding a where clause

Inside loop bodies, the idiom `if (p) then e else ()` is so common that FLWOR comes with a SQL-like `where` clause to address this.

### A where clause

If `ebv(p)` evaluates to `false()` under the current variable bindings, the current iteration does not contribute to the result:

<code>for \$v in e<sub>1</sub></code>		<code>for \$v in e<sub>1</sub></code>
<code>where p</code>	<code>≡</code>	<code>return if (p)</code>
<code>return e<sub>2</sub></code>		<code>then e<sub>2</sub></code>
		<code>else ()</code>

# Explicit vs. implicit iteration

## XPath: implicit iteration

```
a[@b = "foo"]/c[2]/d[@e = 42]
```

## Equivalent nested FLWOR blocks

```
for $a in a
where $a/@b = "foo"
return for $c at $p in $a/c
      where $p = 2
      return for $d in $c/d
            where $d/@e = 42
            return $d
```

**NB.** Unlike the XPath step operator /, for does not change the context item '.

## FLWOR: Reorder iteration result via order by

In a FLWOR block for  $\$v$  in  $e_1$  return  $e_2$ , the order of  $e_1$  determines the order of the resulting sequence.

### Reordering via order by

In the FLWOR block

```
for  $\$v$  in  $e_1$   
order by  $e_3$  [ascending|descending] [empty greatest|least]  
return  $e_2$ 
```

the value (atomization!) of  $e_3$  determines the order in which the bindings of  $\$v$  are used to evaluate  $e_2$ .

# FLWOR: Reordering examples

## An order by “no-op”: reordering by sequence order

```
for $x at $p in (5,3,1,4,2)
order by $p           ⇒ (5,3,1,4,2)
return $x
```

## All bound variables in scope in order by

```
for $x at $p in (5,3,1,4,2)
order by $p + $x      ⇒ (1,3,5,2,4)
return $x
```

## Reordering as in SQL's ORDER BY

```
for $x in (5,3,1,4,2)
order by $x           ⇒ (1,2,3,4,5)
return $x
```

# FLWOR: Reordering examples

## ✎ Value-based reordering of an XPath step result

This query reorders the result of the XPath location step `descendant::b` **based on (string) value**. Which result is to be expected?

```
let $a := <a>
  <b id="0">42</b>
  <b id="1">5</b>
  <b id="2"/>
  <b id="3">3</b>
  <b id="4">1</b>
</a>
for $b in $a/descendant::b
order by $b/text() empty greatest
return $b/@id
```

## Answer

# FLWOR semantics: tuple space

- In the **W3C** XQuery specification, the interaction of the five clauses of a FLWOR (for-let-where-order by-return) block is formally explained by means of a **tuple space**:
  - ▶ **Size** of tuple space  $\equiv$  **number of iterations** performed by FLWOR block.
  - ▶ The fields of the tuples represent, for each iteration,
    - 1 for/let variable bindings,
    - 2 the outcome of the where clause,
    - 3 the value of the reordering criterion, and
    - 4 the value returned by the return clause.
- Let us exemplify this here because our own **relational compilation scheme** for FLWOR blocks resembles the tuple space idea.

# FLWOR semantics: tuple space (1)

## Sample FLWOR block

```
for $x at $p in reverse(1 to 10)
let $y := $x * $x
where $y <= 42
order by 5 - $p
return ($p,$x)
```

### 1 Complete tuple space

\$x	\$p	\$y	where	order by	return
10	1	100	false	4	(1,10)
9	2	81	false	3	(2,9)
8	3	64	false	2	(3,8)
7	4	49	false	1	(4,7)
6	5	36	true	0	(5,6)
5	6	25	true	-1	(6,5)
4	7	16	true	-2	(7,4)
3	8	9	true	-3	(8,3)
2	9	4	true	-4	(9,2)
1	10	1	true	-5	(10,1)



# FLWOR semantics: tuple space (2)

② Filtering: where clause ( $\$y \leq 42$ )

$\$x$	$\$p$	$\$y$	where	order by	return
10	1	100	false	4	(1,10)
9	2	81	false	3	(2,9)
8	3	64	false	2	(3,8)
7	4	49	false	1	(4,7)
6	5	36	true	0	(5,6)
5	6	25	true	-1	(6,5)
4	7	16	true	-2	(7,4)
3	8	9	true	-3	(8,3)
2	9	4	true	-4	(9,2)
1	10	1	true	-5	(10,1)

# FLWOR semantics: tuple space (3)

- ③ Reordering: order by clause

\$x	\$p	\$y	where	order by	return
1	10	1	true	-5	(10,1)
2	9	4	true	-4	(9,2)
3	8	9	true	-3	(8,3)
4	7	16	true	-2	(7,4)
5	6	25	true	-1	(6,5)
6	5	36	true	0	(5,6)

- ④ To emit the final result, scan the tuple space in the order specified by the `order by` column, and concatenate the `return` column entries:

(10,1,9,2,8,3,7,4,6,5,5,6) .

**Observation:** some values have been computed, but *never* used ...

# FLWOR: populate tuple space lazily (1)

## Sample FLWOR block

```
for $x at $p in reverse(1 to 10)
let $y := $x * $x
where $y <= 42
order by 5 - $p
return ($p,$x)
```

### 1 Populate variable bindings only

\$x	\$p	\$y			
10	1	100			
9	2	81			
8	3	64			
7	4	49			
6	5	36			
5	6	25			
4	7	16			
3	8	9			
2	9	4			
1	10	1			

# FLWOR: populate tuple space lazily (2)

2 Evaluate: where clause ( $\$y \leq 42$ )

$\$x$	$\$p$	$\$y$	where		
10	1	100	false		
9	2	81	false		
8	3	64	false		
7	4	49	false		
6	5	36	true		
5	6	25	true		
4	7	16	true		
3	8	9	true		
2	9	4	true		
1	10	1	true		

3 Prune tuples

$\$x$	$\$p$	$\$y$	where		
6	5	36	true		
5	6	25	true		
4	7	16	true		
3	8	9	true		
2	9	4	true		
1	10	1	true		

# FLWOR: populate tuple space lazily (3)

- 4 Evaluate: order by clause

\$x	\$p	\$y		order by	
6	5	36		0	
5	6	25		-1	
4	7	16		-2	
3	8	9		-3	
2	9	4		-4	
1	10	1		-5	

- 5 Normalize order by column, evaluate return clause

\$x	\$p	\$y		position()	return
6	5	36		6	(5,6)
5	6	25		5	(6,5)
4	7	16		4	(7,4)
3	8	9		3	(8,3)
2	9	4		2	(9,2)
1	10	1		1	(10,1)

# Variable bindings: Variables are not variable!

## “Imperative” XQuery

Evaluate the expression

```
let $x :=  
  <x><y>12</y>  
    <y>10</y>  
    <y>7</y>  
    <y>13</y>  
</x>  
let $sum := 0  
for $y in $x//y  
let $sum := $sum + $y  
return $sum
```

## Equivalent query

```
let $x :=  
  <x><y>12</y>  
    <y>10</y>  
    <y>7</y>  
    <y>13</y>  
</x>  
  
for $y in $x//y  
  
return 0 + $y
```

- let-bound variables are named values and thus **immutable**.
- Obtain equivalent query via textual **replacement** (lhs  $\rightarrow$  rhs).<sup>38</sup>

<sup>38</sup>Not valid if rhs value depends on a node constructor!

# Constructing XML fragments

- XQuery expressions may **construct nodes with new identity** of all 7 node kinds known in XML:
  - ▶ document nodes, elements, attributes, text nodes, comments, processing instructions (and namespace nodes).
- Since item sequences are flat, the nested application of **node constructors** is the only way to hierarchically structure values in XQuery:
  - ▶ Nested elements may be used to **group** or **compose** data, and, ultimately,
  - ▶ XQuery may be used as an XSLT replacement, *i.e.*, as an XML **transformation** language.

# Direct node constructors

XQuery node constructors come in two flavors:

- 1 **direct** constructors and
- 2 **computed** constructors.

## Direct constructors

The syntax of **direct constructors** exactly matches the **XML syntax**: any well-formed XML fragment  $f$  also is a correct XQuery expression (which, when evaluated, yields  $f$ ).

**Note:** Text content and CDATA sections are both mapped into text nodes by the XQuery data model (“*CDATA isn’t remembered.*”)



# Direct element constructors

## “CDATA isn't remembered”

`<x><![CDATA[foo & bar]]></x>`  $\equiv$  `<x>foo & bar</x>`  
XQuery

- The **tag name** of a direct constructor is constant, its **content**, however, may be computed by any XQuery expression enclosed in curly braces `{...}`.

## Computed element content

`<x>4{ max((1,2,0)) }</x>`  $\Rightarrow$  `<x>42</x>`

- ▶ Double curly braces (`{{` or `}}`) may be used to create content containing literal curly braces.

# Computed element constructors

## Definition

In a **computed element constructor**

$$\text{element } \{e_1\} \{e_2\}$$

expression  $e_1$  (of type `string` or `QName`) determines the element name,  $e_2$  determines the sequence of nodes in the element's content.

## Example: computed element name and content

```
element { string-join(("foo","bar"),"-") } { 40+2 }  
⇒      <foo-bar>42</foo-bar>
```

# Computed element constructors

## An application of computed element constructors: i18n

Consider a dictionary in XML format (bound to variable `$dict`) with entries like

```
<entry word="address">
  <variant lang="de">Adresse</variant>
  <variant lang="it">indirizzo</variant>
</entry>
```

We can use this dictionary to “translate” the tag name of an XML element `$e` into Italian as follows, preserving its contents:

```
element
{ $dict/entry[@word=name($e)]/variant[lang="it"] }
{ $e/@*, $e/node() }
```

# Direct and computed attribute constructors

- In **direct attribute constructors**, computed content may be embedded using curly braces.

## Computed attribute content

```

<x a="{(4,2)}"/>  ⇒  <x a="4 2"/>
<x a="{{' b=''}}"/>  ⇒  <x b="" a=""/>
<x a="'' b='''"/>  ⇒  <x a="'' b="&quot;"/>
  
```

- A **computed attribute constructor** attribute  $\{e_1\} \{e_2\}$  allows to construct *parent-less* attributes (impossible in XML) with computed names and content.

## A computed and re-parented attribute

```

let $a := attribute {"a"} { sum((40,2)) }
return <x>{ $a }</x>
  
```

# Text node constructors

## Text node construction

**Text nodes** may be constructed in one of three ways:

- 1 Characters in element content,
- 2 via `<![CDATA[...]]>`, or
- 3 using the **computed text constructor** `text {e}`.

Content sequence `e` is atomized to yield a sequence of type `anyAtomicType*`. The atomic values are converted to type `string` and then concatenated with an intervening `"_"`.

If `e` is `()`, no text node is constructed—the constructor yields `()`.

# Examples: computed text node constructor

## Explicit semantics of text node construction `text {e}`

```

if (empty(e))
then ()
else text { string-join(for $i in data(e)
                        return string($i),
                        "_") }

```

## Text node construction examples

```

text { (1,2,3) }  ≡  text { "1 2 3" }

let $n := <x>
           <y/><z/>
           </x>//name(.)
return <t>{ text {$n} }</t>

```

⇒ <t>x y z</t>

# XML documents vs. fragments

- Unlike XML **fragments**, an XML **document** is rooted in its **document node**. The difference is observable via XPath:



Remember the (invisible) document root node!

**xy.xml**

```

1  <x>
2    <y/>
3  </x>
  
```

`doc("xy.xml")/*`  $\Rightarrow$  `<x><y/></x>`

`<x><y/></x>/*`  $\Rightarrow$  `<y/>`

The context node for the first expression above is the document node for document `xy.xml`.

- A document node may be constructed via `document {e}`.

## Creating a document node

`(document { <x><y/></x> })*`  $\Rightarrow$  `<x><y/></x>`

# Processing element content

- The XQuery element constructor is quite flexible: the **content sequence** is not restricted and may have type `item*`.
- Yet, the content of an element needs to be of type `node*`:
  - 0 **Consecutive literal characters** yield a single text node containing these characters.
  - 1 Expression enclosed in `{...}` are evaluated.
  - 2 **Adjacent atomic values** are cast to type `string` and collected in a single text node with intervening `"_"`.
  - 3 A **node** is **copied** into the content **together with its content**. *All copied nodes receive a new identity.*
  - 4 Then, **adjacent text nodes** are merged by concatenating their content. Text nodes with content `" "` are dropped.



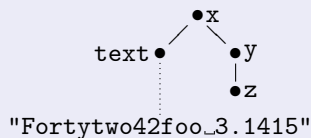
# Example: processing element content

✎ Evaluate the expression below

```
count(
  <x>Fortytwo{40 + 2}{ "foo",3.1415,<y><z></y>,
    ("","!") [1] }</x>/node())
```

## Solution:

The constructed node is



# Well-formed element content

XML fragments constructed by XQuery expressions are subject to the XML rules of **well-formedness**, *e.g.*,

- no two attributes of the same element may share a name,
- attribute nodes precede any other element content.<sup>39</sup>

## Violating the well-formedness rules

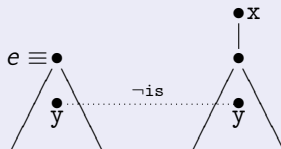
```
let $id := "id"
return
  element x {
    attribute {$id} {0},
    attribute {"id"} {1} } ⇒ ⚡ (dynamic error)
```

```
<x>foo{ attribute id {0} }</x> ⇒ ⚡ (type error)
```

<sup>39</sup>The content type needs to be a subtype of  
`attribute(*)*, (element(*)|text()|...)*.`

# Construction generates new node identities

element  $x \{e\}$ : Deep subtree copy



```
let $e := <a><b/><c><y>foo</y></c></a>
let $x := element x { $e }
return exactly-one($e//y) is exactly-one($x//y) ⇒ false()
```

- Node constructors have **side effects**.



Referential transparency is lost!

```
let $x := <x/>
return $x is $x ⇒ false() | let $d := doc(uri)
return $d is $d ⇒ true()
```

# Construction establishes document order

## ✎ Result of the following query?

```
let $x := <x/>
let $y := <y/>
let $unrelated := ($x, $y)
let $related   := <z>{ $unrelated }</z>/*
return ($unrelated[1] << $unrelated[2],
        $related[1]   << $related[2]   )
```

## Solution

# Construction: pair join partners

## A join query

```
let $a := <a><b><c>0</c></b>
          <b><c>0</c><c>1</c><c>2</c></b>
        </a>
let $x := <x><z id="2">two</z><z id="0">zero</z>
          <y><z id="0">zero'</z><z id="3">three</z></y>
        </x>
for $c in $a/b/c
for $z in $x//z[@id eq $c]           (: join predicate :)
return <pair>{ $c,$z/text() }</pair>
```

## Result

# Grouping (attempt #1)

## A grouping query

```
let $a := <a><b><c>0</c></b>
          <b><c>0</c><c>1</c><c>2</c></b>
        </a>
let $x := <x><z id="2">two</z><z id="0">zero</z>
          <y><z id="0">zero'</z><z id="3">three</z></y>
        </x>
for $c in $a/b/c
return <group>{
  $c, <mem>{ for $z in $x//z[@id eq $c]
             return $z/text() }</mem>
}</group>
```

- **Aggregate functions** (sum, count, ...) may be applied to group members, *i.e.*, element `mem` inside each group.

# Grouping (attempt #1)

Result (NB: group of `<c>0</c>` appears twice)

```
<group><c>0</c><mem>zerozero'</mem></group>
```

```
<group><c>0</c><mem>zerozero'</mem></group>
```

```
<group><c>1</c><mem/></group>
```

← empty group!

```
<group><c>2</c><mem>two</mem></group>
```

## Remarks:

- The preservation of the empty group for `<c>1</c>` resembles the effect of a relational **left outer join**.
- The duplicate elimination implicit in `$a/b/c` is based on node identity but we **group by value** (`@id eq $c`).  
⇒ Such groupings call for value-based duplicate elimination.



## Grouping (attempt #2)

### Improved grouping query

```
let $a := ... unchanged ...  
let $x := ... unchanged ...  
for $c in distinct-values($a/b/c)  
return <group>{  
    <c>{ $c }</c>,  
    <mem>{ $x//z[@id eq $c]/text() }</mem>  
}</group>
```

### Note:

- Need to “rebuild” element c (\$c bound to values).
- Inner for loop replaced by equivalent XPath expression.



## XQuery: user-defined functions

It is typical for non-toy XQuery expressions to contain **user-defined functions** which encapsulate query details.

- User-defined functions may be collected into **modules** and then 'import'ed by a query.
- Function declarations may be directly embedded into the **query prolog** (prepended to query, separated by ';' ).

### Declaration of $n$ -ary function $f$ with body $e$

```
declare function  $f$ ($ $p_1$  as  $t_1$ , ..., $ $p_n$  as  $t_n$ ) as  $t_0$  {  $e$  }
```

- If  $t_i$  is omitted, it defaults to `item()*`.
- The pair  $(f, n)$  is required to be unique (overloading).
- Atomization is applied to the  $i$ -th parameter, if  $t_i$  is atomic.

# User-defined function examples

## Form textual root-to-node paths

```
declare default function namespace
    "http://www-db.in.tum.de/XQuery/functions";

declare function path($n as node()) as xs:string
{ fn:string-join(for $a in $n/ancestor-or-self::*
                  return fn:name($a), "/")
};

let $a := <a><b><c><d/></c><d/></b></a>
return $a//d/path(.)

⇒ ("/a/b/c/d", "/a/b/d")
```

Avoid to place user-def'd functions in the XQuery builtin function namespace (predefined prefix `fn`).

⇒ Use explicit prefix for user-def'd or builtin functions.

# User-defined function examples

## Reverse a sequence

Reversing a sequence does not inspect the sequence's items in any way:

```
declare function reverse($seq)
{ for $i at $p in $seq
  order by $p descending
  return $i
};

reverse((42,"a",<b/>,doc("foo.xml")))
```

### Note:

- The calls  $f()$  and  $f(() )$  invoke different functions.

## Uder-defined functions: recursion

Trees are the prototypical **recursive** data structure in Computer Science and it is natural to describe computations over trees in a recursive fashion.<sup>40</sup>

### Simulate XPath ancestor via parent axis

```
declare function ancestors($n as node()?) as node()*  
{ if (fn:empty($n)) then ()  
  else (ancestors($n/..), $n/..)  
}
```

### Questions

- 1 Will the result be in document order and duplicate free?
- 2 What if we declare the parameter type as `node()*?`

---

<sup>40</sup>This is a general and powerful principle in programming: *derive a function's implementation from the shape of the data it operates over.*

# Answers

# User-defined functions: recursion examples

## Purpose of function `hmm` and output of this query?

```
declare function local:hmm($e as node()) as xs:integer
{ if (fn:empty($e/*)) then 1
  else fn:max(for $c in $e/*
              return local:hmm($c)) + 1
};

local:hmm(<a><b/>
        <b><c><d>foo</d><e/></c></b>
        </a>)
```

### Good style:

- Use predefined namespace `local` for user-def'd functions.
- `hmm` has a more efficient equivalent (*cf.* a previous slide 262), exploiting the recursion “built into” axes `descendant` and `ancestor`.

# User-defined functions: “rename” attribute

## Rename attribute \$from to \$to

```
declare function local:xlate($n      as node(),
                             $from as xs:string,
                             $to   as xs:string)

{ typeswitch ($n)
  case $e as element() return
    let $a := ($e/@*)[name(.) eq $from]
    return
      element
        { node-name($e) }
        { $e/(@* except $a),
          if ($a) then attribute {$to} {data($a)}
          else (),
          for $c in $e/node()
            return local:xlate($c, $from, $to) }
    default return $n
};
```

## User-defined functions: “rename” attribute

### Invoke `xlate`

```
local:xlate(<x id="0" foo="!">
  foo
  <y zoo="1">bar</y>
</x>,
"foo",
"bar")
```



```
<x id="0" bar="!">
  foo
  <y zoo="1">bar</y>
</x>
```

- **NB:** This constructs an entirely new tree.
- In XQuery 1.0, there is currently no way to modify the properties or content of a node.
- XQuery Update will fill in this gap (work in progress at [W3C](#)).

**N.B.: XSLT** (see above) has been designed to support **XML transformations** like the one exemplified here.



## “Rename” attribute in XSLT

### XSLT: rename attributes foo to bar

```
<xsl:template match="@foo">
  <xsl:attribute name="bar">
    <xsl:value-of select="."/>
  </xsl:attribute>
</xsl:template>

<xsl:template match="node()|@*">
  <xsl:copy>
    <xsl:apply-templates select="node()|@*" />
  </xsl:copy>
</xsl:template>
```

### Remember:

- The XSLT processor implicitly matches the given pattern rules against the input tree (recursive traversal “built into” XSLT).
- > 1 pattern matches: more specific rules override generic rules.

# XQuery: the missing pieces

- This chapter did *not* cover XQuery exhaustively. As we go on, we might fill in missing pieces (e.g., `typeswitch`, `validate`).
- This course will not cover the following XQuery aspects:
  - ▶ **(namespaces)**,
  - ▶ **modules** (declaration and import),
  - ▶ **collations** (string equality and comparison).

Reminder:  XQuery specification

<http://www.w3.org/TR/xquery/>

(Has entered *Candidate Recommendation* phase as we speak.)

# Part XI

## Mapping Relational Databases to XML

# Outline of this part

## 28 Mapping Relational Databases to XML

- Introduction
- Wrapping Tables into XML
- Beyond Flat Relations
- Generating XML from within SQL

## 29 Some XML Benchmarking Data Sets

# 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

<i>Employees</i>			
<u><i>eno</i></u>	<u><i>name</i></u>	<u><i>salary</i></u>	<u><i>phone</i></u>
⋮	⋮	⋮	⋮
007	James	1,000,000	123 456
⋮	⋮	⋮	⋮

⇒

```

<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:

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

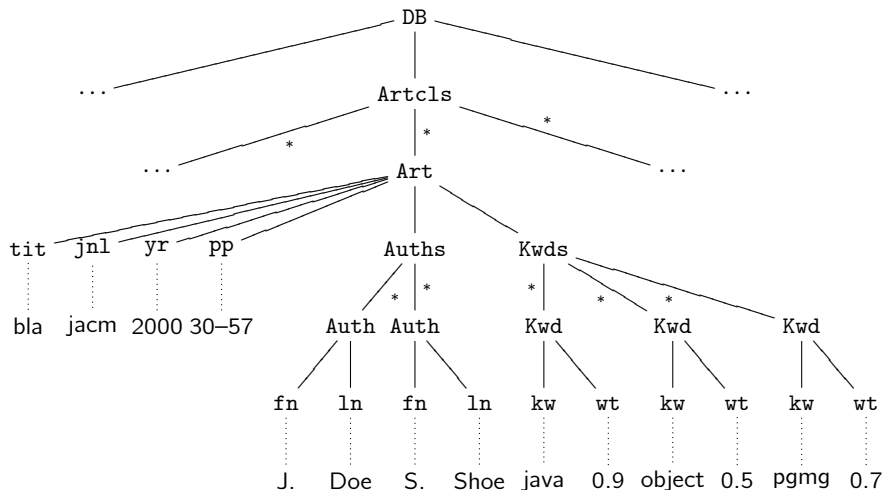
## SQL-3 tables

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
```

⇓

---

*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>

⋮

---

## Some XML benchmarking data sets ... (1)

Among the benchmarks that are commonly used for comparing the performance of various aspects of XML database technologies, there are quite a few that are more or less XML-wrapped relational data, others have converted special-purpose legacy data formats into XML.

- **Xmark** is a very popular XML benchmark. It models an Internet auctioning application. The data used is not XML wrapped relations (the benchmark has been developed for XML), but quite a few bits and pieces might as well have been transferred from (extended) relational.
- **SwissProt** is a large Bioinformatics protein “database”. Today, it is offered in XML form, while it used to be in a special-purpose, line-oriented “keyword-data” format.
  - ▶ Swissprot XML databases typically exhibit **multi-gigabyte** file sizes.
  - ▶ The hierarchical XML tag structure allows for rich annotation and far-reaching queries on content.



## Some XML benchmarking data sets ... (2)

### Swissprot Database Entry (original; non-XML)

```
ID 104K_THEPA      STANDARD;      PRT;    924 AA.
AC P15711;
DT 01-APR-1990 (Rel. 14, Created)
DT 10-MAY-2005 (Rel. 47, Last annotation update)
DE 104 kDa microneme-rhoptry antigen.
OS Theileria parva.
OC Eukaryota; Alveolata; Apicomplexa; Piroplasmida; Theileriidae;
...
```

# Some XML benchmarking data sets ... (3)

## Swissprot XML Database Entry

```
<entry dataset="Swiss-Prot" created="1990-04-01" modified="2005-05-10">
  <accession>P15711</accession>
  <name>104K_THEPA</name>
  <protein>
    <name>104 kDa microneme-rhoptry antigen</name>
  </protein>
  <organism key="1">
    <name type="scientific">Theileria parva</name>
    <dbReference type="NCBI Taxonomy" id="5875" key="2"/>
    <lineage>
      <taxon>Eukaryota</taxon>
      <taxon>Alveolata</taxon>
      ...
    </lineage>
  </organism>
  ...
  <sequence length="924" mass="103626"
    checksum="289B4B554A61870E" modified="1990-04-01">
    MKFLILLFNILCLFPVLAADNHGVPQGASGVDPITFDINSNQTGPAFLT
    AVEMAGVKYLQVQHGSNVNIHRLVEGNVVIWENASTPLYTGAIVTNNDGP
    ...
  </sequence>
</entry>
```

## Some XML benchmarking data sets ... (4)

- **MedLine** is a commercial bibliographic database in the biochemical/medical topic area.  
Some XML database performance studies have been carried out using an XML'ified version akin to the journal bibliography discussed above.
- **Astronomy** data has also been used for benchmarking *really* large data sets (satellites) beam down *enormous* amounts of—typically simply structured—sensor data for (astro-) physical or geo-observation experiments.  
Here, (flat) relational representations would be possible, too.
- **DBLP**<sup>41</sup> is an on-line bibliographic service running on a special purpose internal data representation. The data can be offloaded from the server in a variety of formats, one of those is XML.

---

<sup>41</sup>[dblp.uni-trier.de](http://dblp.uni-trier.de)

## Part XII

# Mapping XML to Databases

# Outline of this part

## 30 Mapping XML to Databases

- Introduction

## 31 Relational Tree Encoding

- Dead Ends
- Node-Based Encoding
- Working With Node-Based Encodings

## 32 XPath Accelerator Encoding

- Tree Partitions and XPath Axes
- Pre-Order and Post-Order Traversal Ranks
- Relational Evaluation of XPath Location Steps

## 33 Path-Based Encodings

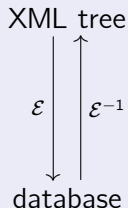
- Motivation
- Data Guides
- Skeleton Extraction and Compression
- Data Vectors
- Skeleton Compression and Semi-Structured Data
- Improving Skeleton Compression

# 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)



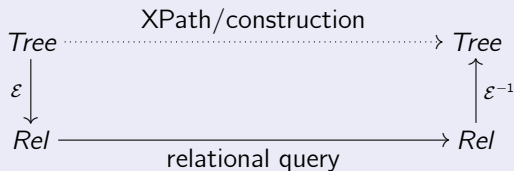
We will call the mapping  $\mathcal{E}$  an *encoding* in the sequel.

## 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:

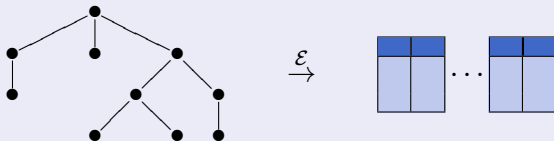
- Re-use existing relational database infrastructure—table storage layer and indexes (*e.g.*, B-trees), SQL or algebraic query engine and optimizer—and invade the database kernel in a very limited fashion (or, ideally, not at all).

## 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"	<a id="0"><b>fo</b>o...</a>	...

- ▶ 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

<u>id</u>	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

<u>id</u>	@no	b
0	0	$\alpha$
3	1	$\beta$
5	NULL <sup>a</sup>	$\gamma$
6	3	NULL <sup>b</sup>

<u>id</u>	b	c
1	$\alpha$	X
2	$\alpha$	NULL <sup>c</sup>
4	$\beta$	Y

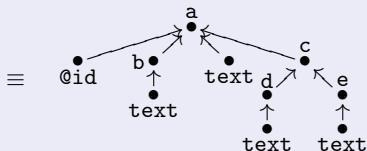
## Issues:

- Number of encoding tables depends on nesting depth.
- Empty element c encoded by NULL<sup>c</sup>, empty element b encoded by absence of  $\gamma$  (will need *outer join* on column b).
- NULL<sup>a</sup> encodes absence of attribute, NULL<sup>b</sup> 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

<u>id</u>	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 with *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 choosing 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

<u>id</u>	parent
⋮	⋮

could be considered a *node-based* (1 tuple per node ID) as well as an *edge-based* (1 tuple per edge) representation.<sup>42</sup>

<sup>42</sup>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:

$$\langle \text{rootID} \rangle . \langle \text{rootchildID} \rangle . \dots . \langle \text{parentID} \rangle . \langle \text{nodeID} \rangle$$

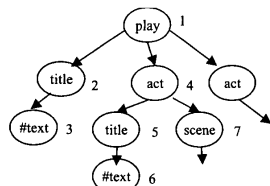
### Observations:

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

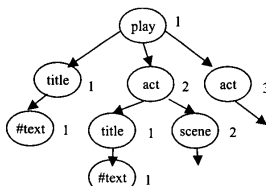
# The need for renumbering ...

Depending on the choice of node IDs, updates to the document (structure) may require the reassignment of IDs to (parts of) the document's nodes.

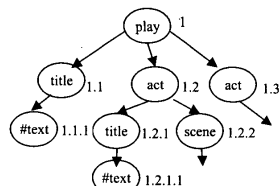
- Insertions and deletions of nodes/subtrees may require renumbering of all following nodes within the document (global numbering) or within the same parent (local).
- In some cases, renumbering can be avoided, *e.g.*, by leaving gaps (sparse numbering).



Global Order



Local Order



Dewey Order



## 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

<u>nodeID</u>	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

<u>nodeID</u>	parentID	pathID	value
⋮	⋮	⋮	⋮

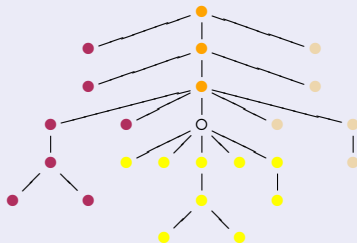
pathtable

<u>pathID</u>	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

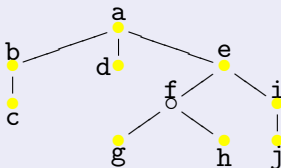
Axes: descendant, ancestor, preceding, following



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

# Tree partitions and XPath axes

Context node (here: *f*) is arbitrary



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

**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.

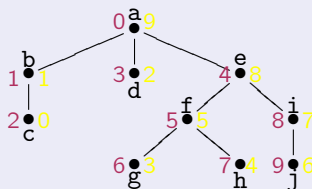
- If we can exploit the partitioning property, the encoding will represent each tree node exactly once.
- In a sense, the semantics of the XPath axes **descendant**, **ancestor**, **preceding**, and **following** will be “built into” the encoding  $\Rightarrow$  **“XPath awareness”**.
- XPath accelerator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

# Pre-order and post-order traversal ranks

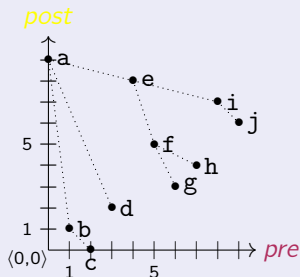
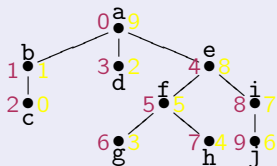
## Pre-order/post-order traversal

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

## Pre-order/post-order traversal rank assignment



# Pre-order/post-order: Tree isomorphism

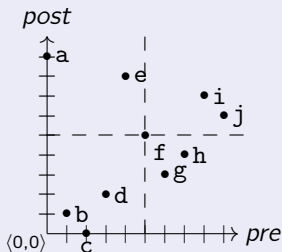
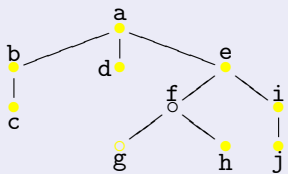


$pre(v)$  encodes document order and node identity

$$v_1 \ll v_2 \iff pre(v_1) < pre(v_2) \quad | \quad v_1 \text{ is } v_2 \iff pre(v_1) = pre(v_2)$$

# XPath axes in the pre/post plane

Plane partitions  $\equiv$  XPath axes,  $\circ$  is arbitrary!



Pre/post plane regions  $\equiv$  major XPath axes

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



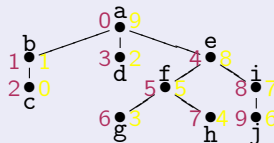
# XPath Accelerator encoding

## 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

<i>pre</i>	<i>post</i>	<i>par</i>	<i>kind</i>	<i>tag</i>	<i>text</i>
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  $\alpha$ )  $\Rightarrow$  SQL window query

```
SELECT  DISTINCT  $v'.*$ 
      FROM    context  $v$ , accel  $v'$ 
      WHERE    $v'$  INSIDE  $window(\alpha, v)$ 
      ORDER BY  $v'.pre$ 
```

# 10 XPath axes<sup>43</sup> and *pre/post* plane windows

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

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

<sup>43</sup>Missing axes in this definition: *self* and *attribute*.

# Pre/post plane window $\Rightarrow$ SQL predicate

descendant::foo, context node  $v$

$$\begin{aligned} v' \text{ INSIDE } \langle (v.pre, *), (*, v.post), *, elem, foo \rangle \\ \equiv \\ v'.pre > v.pre \text{ AND } v'.post < v.post \text{ AND } \\ v'.kind = elem \text{ AND } v'.tag = foo \end{aligned}$$

ancestor-or-self::\*, context node  $v$

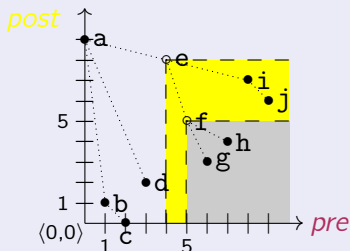
$$\begin{aligned} v' \text{ INSIDE } \langle (*, v.pre], [v.post, *), *, elem, * \rangle \\ \equiv \\ v'.pre \leq v.pre \text{ AND } v'.post \geq v.post \text{ AND } \\ v'.kind = elem \end{aligned}$$

$(e,f)/\text{descendant}::\text{node}()$

## Context & frag. encodings

context		
<i>pre</i>	<i>post</i>	...
5	5	
4	8	

accel		
<i>pre</i>	<i>post</i>	...
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)

$$\text{path}(\text{fn:root}()) = \begin{array}{l} \text{SELECT } v'.* \\ \text{FROM } \textit{accel } v' \\ \text{WHERE } v'.pre = 0 \end{array}$$
$$\text{path}(c/\alpha) = \begin{array}{l} \text{SELECT } \text{DISTINCT } v'.* \\ \text{FROM } \textit{path}(c) \textit{ } v, \textit{accel } v' \\ \text{WHERE } v' \text{ INSIDE } \textit{window}(\alpha, v) \\ \text{ORDER BY } v'.pre \end{array}$$
$$\text{path}(c[\alpha]) = \begin{array}{l} \text{SELECT } \text{DISTINCT } v.* \\ \text{FROM } \textit{path}(c) \textit{ } v, \textit{accel } v' \\ \text{WHERE } v' \text{ INSIDE } \textit{window}(\alpha, v) \\ \text{ORDER BY } v.pre \end{array}$$

# An example: Compiling XPath into SQL

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

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

=

```
SELECT DISTINCT v1.*
  FROM path(fn:root/descendant::a) v, accel v1
 WHERE v1 INSIDE window(child::text(), v)
ORDER BY v1.pre
```

=

```
SELECT DISTINCT v1.*
  FROM (
    SELECT DISTINCT v2.*
      FROM path(fn:root) v, accel v2
     WHERE v2 INSIDE window(descendant::a, v)
    ORDER BY v2.pre
  ) v,
  accel v1
 WHERE v1 INSIDE window(child::text(), v)
ORDER BY v1.pre
```

# 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*( $\cdot$ ) 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!**

# Path-based encodings

## Some observations:

- In many cases, the volume of *large* XML documents mainly comes from their *text contents* (PCDATA); their markup/structure is of moderate size.
- In contrast, most *queries* tend to focus on structural aspects (XPath navigation, tag name tests, ...), with only *occasional access* to character contents.
- Many document collections—even though of only *semi-structured* objects—share large fractions of structure across individual documents/fragments.

## Possible conclusions: try to ...

- represent structure *separate* from contents,
- keep structural representation in (*main*) *memory*,
- identify *common* structure (and possibly contents as well), and store only once

# Data guides/skeletons

## Separate structure from contents . . .

- Chose representations for XML structure (non-leaf nodes) and text contents *independently*.
- Store the two representations separate from each other, such that **structural info** (“skeleton” or “data guide”)
  - ▶ can be kept small (and thus, in main memory),
  - ▶ supports major XQuery functionality (*esp.*, XPath navigation) efficiently,

and **text contents** data

- ▶ can be accessed only on demand,
- ▶ directed by structure (hence the term “data guide”).

Often, main memory-oriented data structures are used for the skeleton, while external memory data structures hold text contents.

# Skeleton extraction

- Conceptually, a *skeleton* of an XML document can be obtained by replacing all text content (leaf) nodes of an XML tree with a special “marker” (e.g., a hash mark “#”), indicating that some textual content has been removed.
- The resulting XML tree is a faithful representation of the *structure* of the original document, while all actual content has to be stored elsewhere.
- Since the skeleton is small (compared to the whole document), it may even be feasible to represent it as a DOM tree in main memory.
  - ▶ If we assign (global) *node IDs* to text contents nodes (as usual), those IDs can be used to access text contents from the skeleton.
  - ▶ If text contents is stored separately *in document order*, we may not even need the IDs, since a joint traversal of the skeleton and the list of text contents nodes can bring them together.

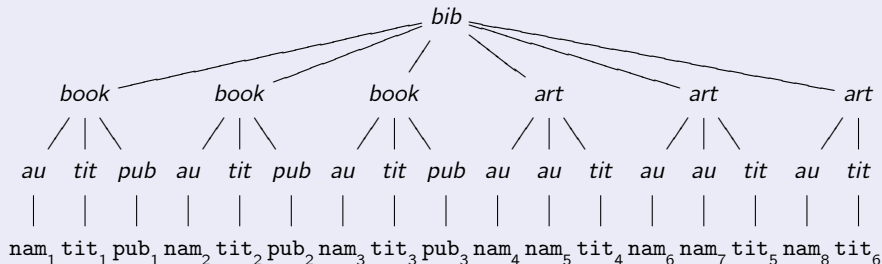
# Skeleton compression

## Notice the following:

- 1 the more regular the structure of the XML document (collection), the more *identical* subtrees the skeleton will have,
- 2 it conserves (memory) space, if we *fold* identical, adjacent subtrees in the skeleton,
- 3 an even more compact representation can be obtained, if we *share common subtrees*, resulting in a skeleton DAG.

# Skeleton compression (example)

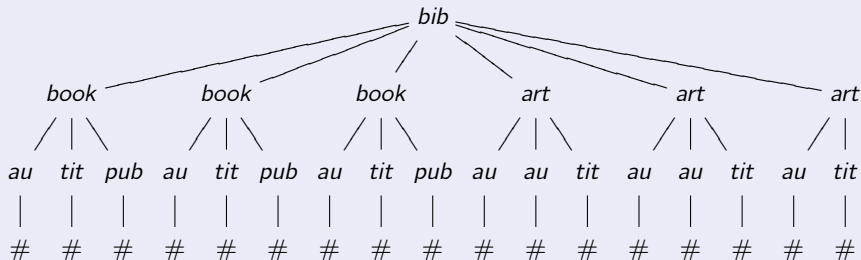
Given this XML document ...



- ❶ Replace text contents by special marker “#” to obtain skeleton.
- ❷ Fold identical, adjacent subtrees to obtain first version of a *compressed* skeleton.
- ❸ Share common subtrees obtaining *compressed skeleton DAG*.

# Skeleton compression (example)

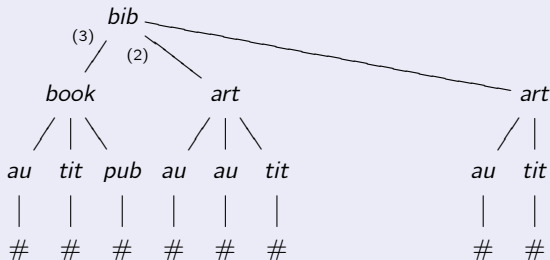
Extract the skeleton ...



- ➊ Replace text contents by special marker “#” to obtain skeleton.
- ➋ Fold identical subtrees to obtain first version of a *compressed* skeleton.
- ➌ Share common subtrees obtaining *compressed skeleton DAG*.

# Skeleton compression (example)

## Compress the skeleton (1) ...

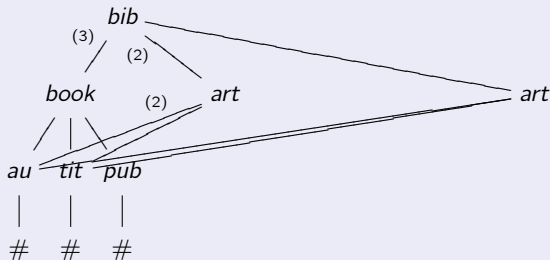


- ① Replace text contents by special marker “#” to obtain skeleton.
- ② Fold identical subtrees to obtain first version of a *compressed* skeleton.
- ③ Share common subtrees obtaining *compressed skeleton DAG*.



# Skeleton compression (example)

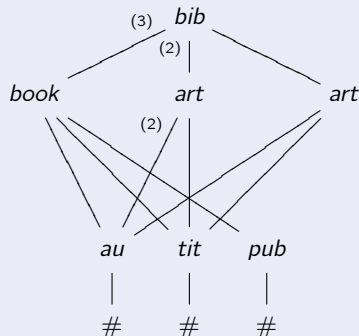
## Compress the skeleton (2) ...



- 1 Replace text contents by special marker “#” to obtain skeleton.
- 2 Fold identical subtrees to obtain first version of a *compressed* skeleton.
- 3 Share common subtrees obtaining *compressed skeleton DAG*.

# Skeleton compression (example)

Resulting compressed skeleton, redrawn ...



**N.B.:** text contents could be stored in several, different formats.  
In the literature, skeleton compression has been proposed in combination with *data vectorization* ... (see below)

## Data vectors

For each distinct (element name) path from the document root to a text node, create a relational table named after that path. Tuples contain node IDs and text contents.

### Example (continued from above)

- 1 Distinct paths from the root node to text contents nodes in the `bib` document are: `/bib/book/au`, `/bib/book/tit`, `/bib/book/pub`, `/bib/art/au`, `/bib/art/tit`.
- 2 Vectorization thus generates 5 tables:

/bib/book/au	
ID	text
...	nam <sub>1</sub>
...	nam <sub>2</sub>
...	nam <sub>3</sub>

/bib/book/tit	
ID	text
...	tit <sub>1</sub>
...	tit <sub>2</sub>
...	tit <sub>3</sub>

/bib/book/pub	
ID	text
...	pub <sub>1</sub>
...	pub <sub>2</sub>
...	pub <sub>3</sub>

/bib/art/au	
ID	text
...	nam <sub>4</sub>
...	nam <sub>5</sub>
...	nam <sub>6</sub>
...	nam <sub>7</sub>
...	nam <sub>8</sub>

/bib/art/tit	
ID	text
...	tit <sub>4</sub>
...	tit <sub>5</sub>
...	tit <sub>6</sub>

**Question now:** What are suitable “IDs” for text contents nodes in this representation?

## Array implementation of data vectors

If we assign IDs *locally*, within each of the vectorized tables, and *in document order*, i.e., we sequentially number tuples in those tables, sorted by document order,

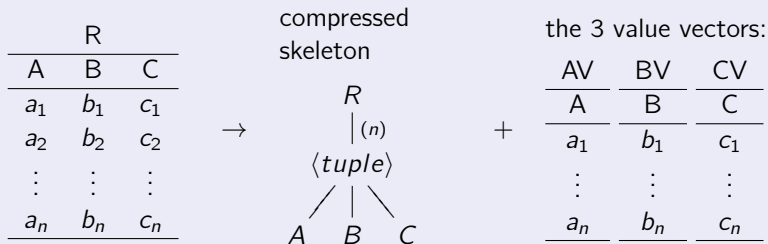
- we can completely dispense with the ID columns, and use offset addressing (like in an array or *vector*—hence the name!),
- a parallel, sequential traversal of the skeleton and the data vectors will allow faithful reproduction of the original document,
- structure-oriented queries will only need to access those large tables/vectors, whose text contents is really needed for query processing (predicate testing or result construction).

The vectorization approach to data storage corresponds to a *full vertical partitioning* scheme for relational tables.

# Skeleton compression and semi-structured data

Skeleton compression works most effectively, if the XML data exhibits a highly regular structure.<sup>44 45</sup>

In the extreme (XML-wrapped flat table data)



... an RDBMS would/could do (roughly) the same: schema info separate from values, cardinality (the “(n)”) in the catalogs.

<sup>44</sup>In the sequel, we discuss skeleton compression together with data vectorization.

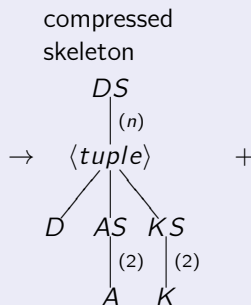
<sup>45</sup>Also, we do not explicitly show the “#” leaf nodes.

# Another example

## XML-wrapped *nested* table data

(think of: D...document, A...author, K...keyword, xS...x-set)

DS		
D	AS	KS
	A	K
$d_1$	$a_1$	$k_1$
	$a_2$	$k_2$
$d_2$	$a_3$	$k_3$
	$a_4$	$k_4$
$\vdots$	$\vdots$	$\vdots$
$d_n$	$a_{2n-1}$	$k_{2n-1}$
	$a_{2n}$	$k_{2n}$



the 3 value vectors:

DV	AV	KV
D	A	K
$d_1$	$a_1$	$k_1$
$\vdots$	$\vdots$	$\vdots$
$d_n$	$a_n$	$k_n$
	$a_{n+1}$	$k_{n+1}$
	$\vdots$	$\vdots$
	$a_{2n}$	$k_{2n}$

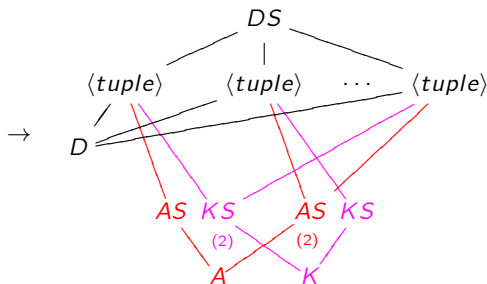
**N.B.** Notice how this works, if *and only if* each document has exactly the same number of authors and keywords!

## Less regularity

Assume no. of authors and/or keywords varies with documents ...  
 Skeleton compression suffers from lack of uniformity and adjacency!

*not so very much compressed* skeleton

	DS	
D	AS	KS
	A	K
$d_1$	$a_1$	$k_1$
		$k_2$
$d_2$	$a_3$	$k_3$
	$a_4$	
$\vdots$	$\vdots$	$\vdots$
$d_n$	$a_{2n-1}$	$k_{2n-1}$
	$a_{2n}$	$k_{2n}$



+ (... the value vectors as above)

## Sharing of common subtrees (1)

In the example on the previous slide,

- if **adjacent** document tuples share the exact same number of authors and keywords,
  - ▶ no new “ $\langle tuple \rangle$ ”-node will be generated in the compressed skeleton, but rather
  - ▶ the multiplicity counter “ $(i)$ ” of the corresponding “ $DS \rightarrow \langle tuple \rangle$ ”-edge will be incremented;
- if, however, **non-adjacent** tuples share the exact same number of authors and keywords,
  - ▶ new “ $DS \rightarrow \langle tuple \rangle$ ”-edges will be created between the “ $DS$ ” node and the corresponding “ $\langle tuple \rangle$ ”-node.

Notice the “spaghetti” structure of the compressed skeleton!



## Sharing of common subtrees (2)

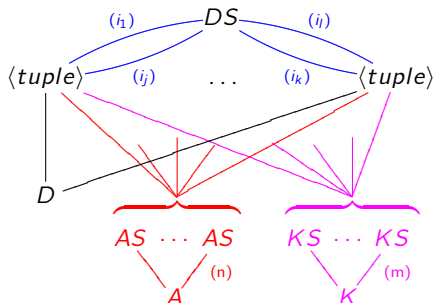
As a result, a compressed skeleton will have

- as many “ $\langle tuple \rangle$ ” nodes as there are distinct ( $\#$  authors,  $\#$  keywords)-pairs
- each of these nodes will have as many edges connecting to the “ $DS$ ” parent node as there are groups of adjacent documents sharing this number of authors and keywords,
- each of these edges will have a multiplicity counter “ $(i)$ ” attached to it, giving the cardinality of the corresponding group of adjacent documents.

The “ $DS$ ” node and its “ $\langle tuple \rangle$ ” children are linked by an *ordered sequence* of multiple edges.

## Example of the general case

For  $N$  documents with  $1 \dots n$  authors and  $1 \dots m$  keywords, we get:



- Each “ $\langle tuple \rangle$ ” node connects to exactly one “AS” and one “KS” node.
- Edges with  $i_j > 1$  represent sequences of  $i_j$  adjacent documents with same  $\#(\text{authors})$  and  $\#(\text{keywords})$ .
- The sum of all  $i_j$ 's is equal to  $N$ .
- Not all of the  $n$  “AS” and  $m$  “KS” nodes are necessarily present.

# Discussion (1)

## Pros:

- Skeleton extraction/compression follows the (database) idea of *separating* type and instance information.
- (Compressed) skeletons are typically small enough to fit into main memory, while only the (mass) instance data needs to be paged in from secondary storage.
- Experiments reported in the literature prove large performance gains compared to both
  - ▶ completely disk-based storage schemes (because of skeleton being kept in main memory), and
  - ▶ completely memory-based schemes (because of capability to handle much larger document collections).

## Discussion (2)

### Cons:

- Skeletons do not compress too well in some cases (*semi*-structured data).
- Compressed skeletons exhibit very clumsy structure (typically implemented in some kind of spaghetti, main memory-only data structure).
- Consequently, if skeleton does not fit into memory, usefulness is unclear.

### Possible ways out . . .

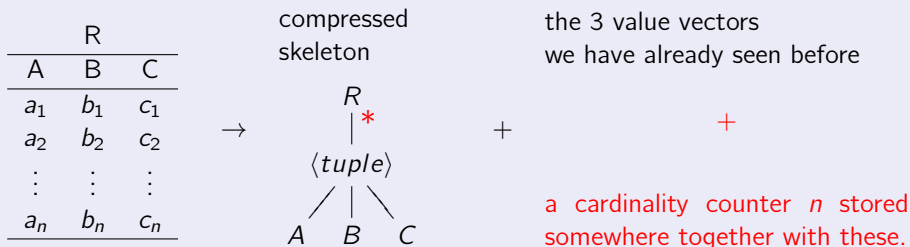
- Improve compression scheme.
- Chose skeleton representation also suitable for secondary storage.
- Combine basic ideas with other representation schemes.

# Improving skeleton compression

## Basic idea:

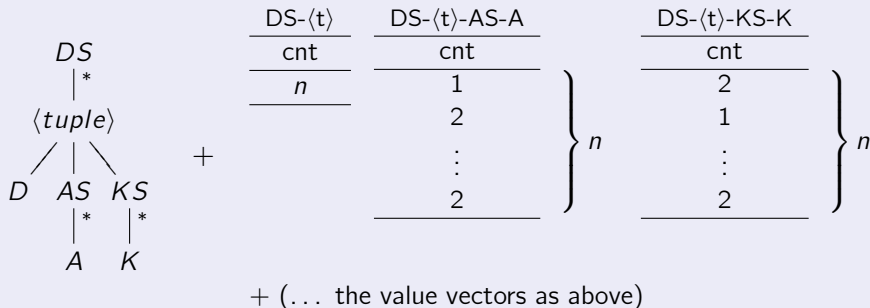
- Even more separation between structural (type) and contents (instance) information. For instance:
  - ▶ *number of repetitions* of set/list-valued substructures is not part of structural (skeleton), but of contents representation,
  - ▶ while the fact that *there is* a repeating substructure is clearly part of the type info.

## In the fully regular, flat table example:



## Nested table example

In general, we need multiple cardinality counters, one for each parent node:



⇒ **Keep one count-vector per “\*” path.**

(assuming we want to store counts in vectors again, to avoid new kinds of data structures)

## Space & time comparison

### Space needed.

Compared to the original skeleton compression scheme, this structure does *not* introduce any space overhead, on the contrary:

- repeating identical structural information is avoided, while
- the counters have been present in the compressed skeleton before.

### Algorithms.

Like the original scheme, this structure lends itself towards sequential, top-down processing (e.g., document serialization, SAX parsing).

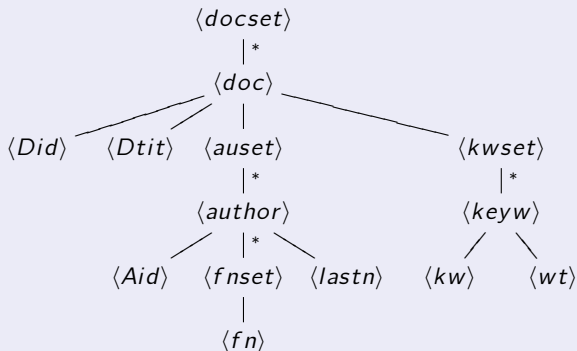
- In the original scheme, traversal needs to follow a more “blown-up” tree structure, while
- in the modified scheme, traversal needs to tally counters.

The (in-memory) cost should be comparable.

## More complex (semi-) structures

In reality, single documents have more “attributes”, authors have more attributes, among them more repeating items, such as firstnames . . .

The compressed skeleton could then look like this:



. . . nothing more than bare *structural* information (aka. a “schema”).



## Future work

This initial idea needs further elaboration.

- Similar idea can be applied for *optional* substructures (0 or 1 repetition).
- Still no good solution for non-consecutive shared substructures.
- Skeleton extraction and compression can be viewed as one approach to *schema inference* for XML documents.
- In the general case, though, it has already been shown that
  - ▶ Generating a DTD from an XML document is an  $\mathcal{NP}$ -complete problem!

So, there are performance limitations . . .

# Part XIII

## Index Support

# Outline of this part

## 34 Index Support

- Overview
- Hierarchical Node IDs and  $B^+$  Trees
- *Pre/Post* Encoding and  $B^+$  Trees
- *Pre/Post* Encoding and R Trees
- More on Physical Design Issues

# Index support

All known database indexing techniques (such as  $B^+$  trees, hashing, ...) can be employed to—depending on the chosen representation—support some or all of the following:

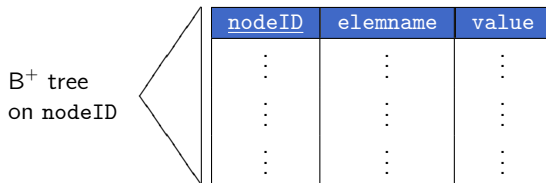
- uniqueness of node IDs,
- direct access to a node, given its node ID,
- ordered sequential access to document parts (serialization),
- name tests,
- value predicates,
- structural traversal along some or all of the XPath axes,
- ...

We will only look into a few interesting special cases here.

## Hierarchical node IDs and B<sup>+</sup> trees

Using a hierarchical numbering scheme for node IDs captures the complete XML structure in the IDs. Hence, no separate representation of structure is needed.

A simplified edge table could be stored as a B<sup>+</sup> tree over the nodeID field:

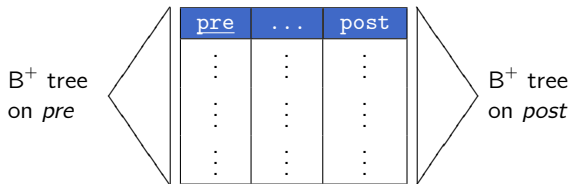


Since nodeIDs are of the hierarchical form  $\langle \text{root\#} \rangle . \langle \text{rootchild\#} \rangle . \dots . \langle \text{parent\#} \rangle . \langle \text{node\#} \rangle$ , with local numbers assigned within each parent, a left-to-right traversal of all leaf node entries of the B<sup>+</sup> tree reads all element nodes in document order.

## Pre/post encoding and B<sup>+</sup> trees

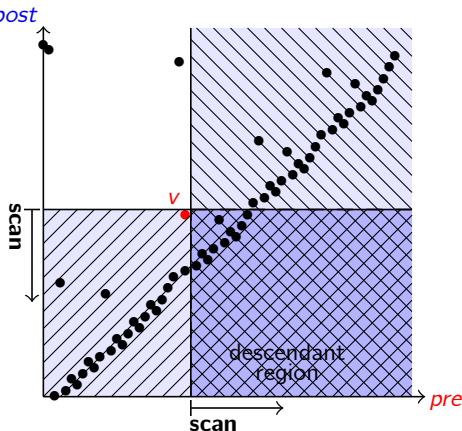
As we have already seen before, the XPath Accelerator encoding leads to *conjunctions* of a lot of *range selection predicates* on the *pre* and *post* attributes in the resulting SQL queries.

Two B<sup>+</sup> tree indexes on the *accel* table, defined over *pre* and *post* attributes:



## Query evaluation (example)

Evaluating, e.g., a descendant step can be supported by either one of the B<sup>+</sup> trees:



Two options:

- ① Use index on *pre*.
  - Start at *v* and **scan** along *pre*.
  - Many **false hits**!
- ② Use index on *post*.
  - Start at *v* and **scan** along *post*.
  - Many **false hits**!

• **Many false hits either way!**

## Query evaluation using index intersection

Standard B<sup>+</sup> trees on those columns will support *really* efficient query evaluation, if the DBMS optimizer generates *index intersection* evaluation plans.

Query evaluation plans for predicates of the form  
“ $pre \in [\dots] \wedge post \in [\dots]$ ” should will then

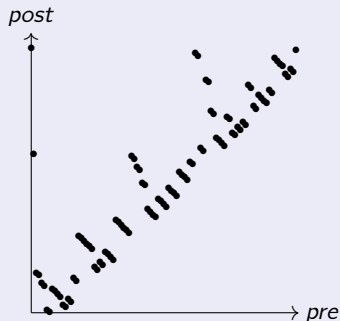
- 1 evaluate both indexes separately to obtain pointer lists,
- 2 merge (*i.e.*, intersect) the pointer lists,
- 3 only *afterwards* access accel tuples satisfying *both* predicates.



## Pre/post encoding and R trees

In the geometric/spatial database application area, quite a few *multi-dimensional* index structures have been developed. Such indexes support range predicates along arbitrary combinations of dimensions.

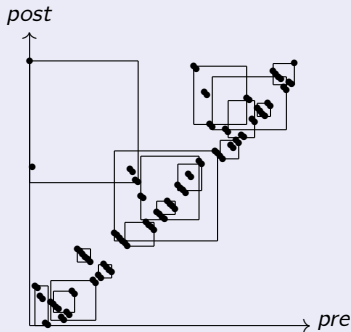
### Pre/post encoding of a 100-node XML fragment



- Diagonal of *pre/post* plane densely populated.
- R-Trees partition plane incompletely, adapts well to node distribution.
- Node encodings are points in 5-dimensional space.
- 5-dimensional R-Tree evaluates XPath **axis** and **node tests** in **parallel**.

# Preorder packed R tree

R tree loaded in ascending preorder, leaf capacity 6 nodes



- Insert node encodings into R tree in ascending order of *pre* ranks.
- Storage utilization in R tree leaf pages maximized.
- Coverage and overlapping of leaves minimized.
- Typical: preorder packing **preserves document order** on retrieval.

## More on physical design issues

As always, choosing a clever physical database layout can greatly improve query (and update) performance.

- Note that all information necessary to evaluate XPath **axes** is encoded in columns *pre* and *post* (and *par*) of table *accel*.
- Also, **kind tests** rely on column *kind*, **name tests** on column *tag* only.

Which columns are required to evaluate the steps below?

Location step	Columns needed
<code>descendant::text()</code>	
<code>ancestor::x</code>	
<code>child::comment()</code>	
<code>/descendant::y</code>	

# Splitting the encoding table

These observations suggest to split *accel* into **binary tables**:

## Full split of *accel* table

prepost		prepar		prekind		pretag		pretext	
<i>pre</i>	<i>post</i>	<i>pre</i>	<i>par</i>	<i>pre</i>	<i>kind</i>	<i>pre</i>	<i>tag</i>	<i>pre</i>	<i>text</i>
0	9	0	NULL	0	elem	0	a	2	c
1	1	1	0	1	elem	1	b	3	d
2	0	2	1	2	text	4	e	7	h
3	2	3	0	3	com	5	f	9	j
4	8	4	0	4	elem	6	g		
5	5	5	4	5	elem	8	i		
6	3	6	5	6	elem				
7	4	7	5	7	pi				
8	7	8	4	8	elem				
9	6	9	8	9	text				

- **NB.** Tuples are **narrow** (typically  $\leq 8$  bytes wide)
  - ⇒ reduce amount of (secondary) memory fetched
  - ⇒ lots of tuples fit in the buffer pool/CPU data cache

## “Vectorization”

- In an **ordered** storage (clustered index!), the *pre* column of table *prepost* is plain redundant.
- Tuples even narrower. Tree shape now encoded by ordered integer sequence (*cf.* “data vectors” idea).

### Dense *pre* column

prepost	
	<i>post</i>
	9
	1
	0
	2
	8
	5
	3
	4
	7
	6

- Use **positional access** to access such tables ( $\rightarrow$  MonetDB).
  - ▶ Retrieving a tuple  $t$  in row  $\#n$  implies  $t.pre = n$ .

## Indexes on encoding tables?

- Analyse compiled XPath query to obtain advise on which **indexes** to create on the encoding tables.<sup>46</sup>

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

```
SELECT DISTINCT v1.pre
  FROM accel v2, accel v1
 WHERE v2.kind = elem and v2.tag = a           ::a
       AND v1.pre > v2.pre                     } descendant
       AND v1.post < v2.post
       AND v1.kind = text                      ::text()
 ORDER BY v1.pre
```

<sup>46</sup>Supported by tools like the IBM DB2 *index advisor* db2advis.

# Indexes on encoding tables

Query analysis suggests:

## SQL index creation commands

```
① CREATE          INDEX itag  ON accel (tag)
② CREATE          INDEX ikind ON accel (kind)
③ CREATE          INDEX ipar  ON accel (par)
④ CREATE UNIQUE INDEX ipost  ON accel (post ASC)
⑤ CREATE UNIQUE INDEX ipre   ON accel (pre ASC) CLUSTER
```

- ①–③: **Hash/B-tree indexes**      ④–⑤: **B-tree indexes**

# Resulting storage layer layout

## Table and index contents (ordered!)

accel			
RID	<i>pre</i>	<i>post</i>	...
$\rho_0$	0	9	
$\rho_1$	1	1	
$\rho_2$	2	0	
$\rho_3$	3	2	
$\rho_4$	4	8	
$\rho_5$	5	5	
$\rho_6$	6	3	
$\rho_7$	7	4	
$\rho_8$	8	7	
$\rho_9$	9	6	

ipost	
RID	<i>post</i>
$\rho_2$	0
$\rho_1$	1
$\rho_3$	2
$\rho_6$	3
$\rho_7$	4
$\rho_5$	5
$\rho_9$	6
$\rho_8$	7
$\rho_4$	8
$\rho_0$	9

ikind	
RID	<i>kind</i>
$\rho_0$	elem
$\rho_1$	elem
$\rho_4$	elem
$\rho_5$	elem
$\rho_6$	elem
$\rho_8$	elem
$\rho_2$	text
$\rho_9$	text
$\rho_3$	com
$\rho_7$	pi

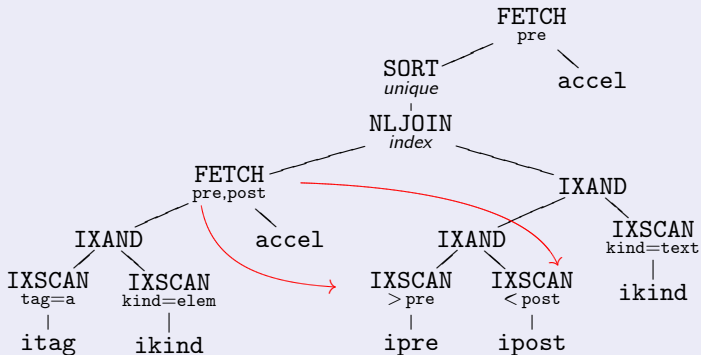
## Notes:

- $\rho_i$  in RID column: database internal *row identifiers*.
- Rows of table `accel` ordered in preorder (CLUSTER).



# Evaluation plan (DB2)

## Plan for the query given above



# A note on the IBM DB2 plan operators

## Query plan operators used by IBM DB2 (excerpt)

Operator	Effect
IXSCAN	<b>Index scan</b> controlled by predicate on indexed column(s); yields row ID set
IXAND	<b>Intersection</b> of two row ID sets; yields row ID set
FETCH	Given a row ID set, <b>fetch specified columns</b> from table; yields tuple set
SORT	<b>Sort</b> given row ID/tuple set, optionally removing duplicates
NLJOIN	<b>Nested loops join</b> , optionally using index lookup for inner input
TBSCAN	<b>Scan entire table</b> , with an optional predicate filter

## Part XIV

# Some Optimizations of the XPath Accelerator Representation

# Outline of this part

## 35 Scan Ranges

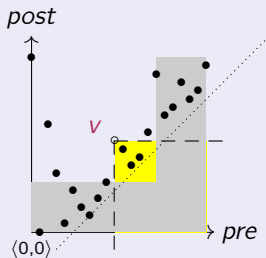
- descendant Axis

## 36 Stretched *Pre/Post* Plane

## 37 XPath Symmetries

## Scan ranges: descendant axis

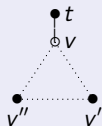
Consider a descendant step originating in context node  $v$ :



A significant fraction of the  $i_{pre}$  and  $i_{post}$  B-tree index scan is guaranteed to deliver **false hits** only.

# Shrink-wrapping the descendant window

## Subtree below $v$



- $v''$  has min. postorder rank below  $v$
- $v'$  has max. preorder rank below  $v$
- $pre(v') = pre(v) + size(v)$
- $post(v'') = post(v) - size(v)$
- Sufficient to scan B-tree in the  $(pre(v), pre(v'))$  range

- $size(v) = |v/descendant::node()|$



If we can derive (a reasonable estimate for)  $size(v)$  from  $pre(v)$  and  $post(v)$ , we can **shrink** the descendant window.

# Shrink-wrapping the descendant window

An alternative characterization of preorder/postorder ranks

$$\begin{aligned}
 pre(v) &= |v/\text{preceding}::\text{node}()| + \overbrace{|v/\text{ancestor}::\text{node}()|}^{= \text{level}(v)} + 1 \\
 post(v) &= |v/\text{preceding}::\text{node}()| + \underbrace{|v/\text{descendant}::\text{node}()|}_{= \text{size}(v)} + 1
 \end{aligned}$$

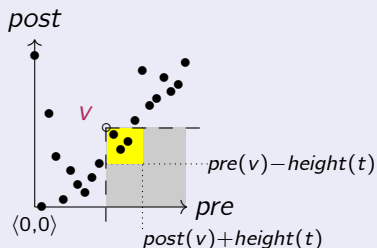
⇓

$$\begin{aligned}
 post(v) - pre(v) &= \text{size}(v) - \underbrace{\text{level}(v)}_{\leq \text{height}(t)} \\
 &\leq \text{height}(t)
 \end{aligned}$$

Estimate the location of  $v'$  and  $v''$  in the *pre/post* plane

$$pre(v') \leq post(v) + \text{height}(t) \mid post(v'') \geq pre(v) - \text{height}(t)$$

# Shrink-wrapping the descendant window



- Size of B-tree scan region now **dependent on actual subtree size** below  $v$  (and independent of fragment  $t$ 's size!).
- Scan region size estimate maximally off by  $height(t)$ .

## Overestimation of descendant window size

How significant would you judge this estimation error? How to avoid the error at all?



## Stretched *pre/post* plane

- While **index intersection** (IXAND) and **window shrinking** go a long way in making location step evaluation efficient in the *pre/post* plane, windows are still evaluated in a two-step process, leading to false hits.
  - ▶ A different way to approach this problem is to employ **concatenated  $\langle pre, post \rangle$  B-trees**.
- Here, instead we will exploit the observation that predicate *window*( $\cdot$ ) solely depends on comparisons ( $<$ ,  $>$ ) on *pre* and *post*. The **absolute *pre/post* values are immaterial**.

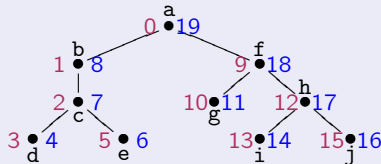
# Stretched *pre/post* plane

## “Stretched” (or coupled) *preorder/postorder* ranks

Perform a depth-first, left-to-right traversal of the skeleton tree.  
Maintain counter *rank* (initially 0).

- 1 Whenever a node  $v$  is visited first, assign  $pre(v) \leftarrow rank$ ; increment  $rank$ .
- 2 When  $v$  is visited last, assign  $post(v) \leftarrow rank$ ; increment  $rank$ .

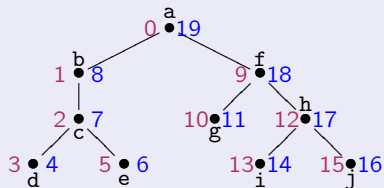
## Example



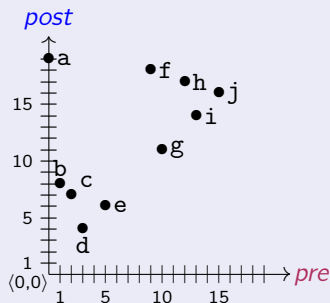
This encoding is also known as “start–end” numbering.

# Stretched *pre/post* plane

## start-end numbering



## Stretched *pre/post* plane



Node identifiers of bit width  $n$  encode  $2^{n-1}$  nodes.

## XPath axes in the stretched *pre/post* plane

Node distribution in the stretched *pre/post* plane has interesting properties:

- The axes *window*( $\cdot$ ) predicates continue to work as before.

Further:

### Characterization of descendant axis

Node  $v$  is selected by *c/descendant::node()*, iff

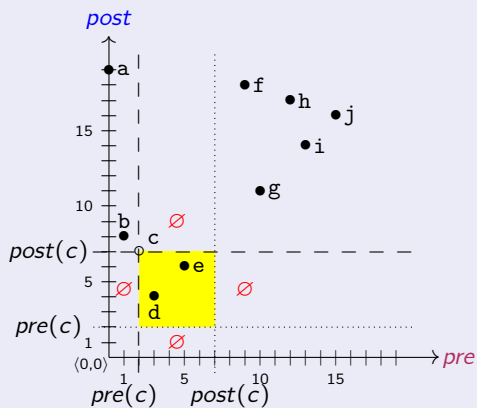
$$pre(v) \in (pre(c), post(c)) \quad \mathbf{or} \quad post(v) \in (pre(c), post(c))$$

### Subtree size (exact, no estimation)

For any node  $v$ :

$$size(v) = 1/2 \cdot (post(v) - pre(v) - 1)$$

## c/descendant::node()



## XPath axes in the stretched *pre/post* plane

In terms of query windows, on the stretched *pre/post* plane we may modify *window*( $\cdot$ ) as follows:

### Axis descendant in the stretched plane

$$window(\text{descendant} :: t, v) = \begin{cases} \langle (pre(v), post(v)), *, *, elem, t \rangle \\ \text{or} \\ \langle *, (pre(v), post(v)), *, elem, t \rangle \end{cases}$$

A **single index scan** suffices (no IXAND, no false hits).

- Axes descendant-or-self and child benefit, too.

## Leaf node access

For a certain class of XPath steps, we can statically<sup>47</sup> infer that **all result nodes will be leaves** (let  $c$  denote an arbitrary XPath expression):

- $c/\text{text}()$ ,  $c/\text{comment}()$ ,  $c/\text{processing-instruction}()$
- $c[\text{not}(\text{child}::\text{node}())]$

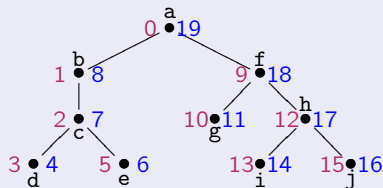
### Characterization of any leaf node $\ell$

A diagonal in the stretched *pre/post* plane:

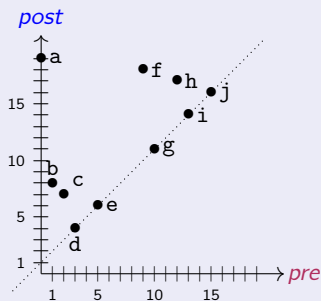
$$\text{post}(\ell) = \text{pre}(\ell) + 1$$

<sup>47</sup>At query compile time.

# Leaves diagonal



$$post(\ell) = pre(\ell) + 1$$





## “Backwards” step processing

Presence of the leaves diagonal enables the RDBMS to evaluate certain XPath expressions in a “backwards” fashion.

### Exploit symmetries in XPath

Consider the query

```
descendant::t/child::text() .
```

We can instead process the **equivalent symmetric** query

```
descendant::text() [parent::t]
```

found on leaves diagonal

**NB.** The latter query does *not* require window evaluation at all.

## Exploiting schema/DTD information

The presence of a **DTD** (or XML Schema description) for a pre-/postorder encoded document may be used to generalize the leaves diagonal discussion.

- From a DTD we can derive **maximal/minimal subtree sizes** for any XML element node  $v$  with tag  $t$ .
- Together with

$$\begin{aligned}
 size(v) &= 1/2 \cdot (post(v) - pre(v) - 1) \\
 &\quad \updownarrow \\
 post(v) &= 2 \cdot size(v) + pre(v) + 1
 \end{aligned}$$

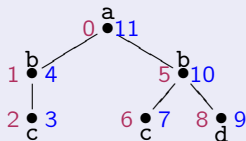
we can establish a **stripe** in the stretched *pre/post* plane which is guaranteed to contain all elements with tag  $t$ .

# Exploiting schema/DTD information

## Sample DTD and encoding of a **valid** fragment

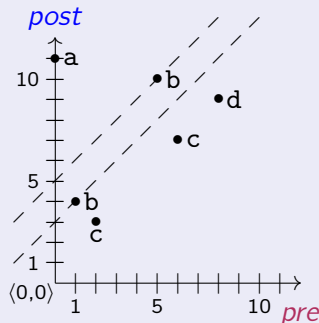
```
<!ELEMENT a (b+)>
<!ELEMENT b (c,d?)>
<!ELEMENT c EMPTY>
<!ELEMENT d EMPTY>
```

⇒ Minimum (maximum) subtree size of b elements in a valid fragment is 1 (2).



⇒ All b elements in stripe

$$3 \leq post(v) - pre(v) \leq 5$$



# XPath symmetries

- Clearly, *pre/post* plane **window size** is the dominating cost factor for the XPath Accelerator.
  - ▶ The window size determines the stride of B-tree range scans and thus the amount of secondary memory touched (affects # I/O operations necessary).

(We could even try to derive a **cost model** from window size.)

- How can we benefit from this observation?

# XPath symmetries

Plan choices: `/descendant::t/ancestor::s`

❶ **Forward mode.**

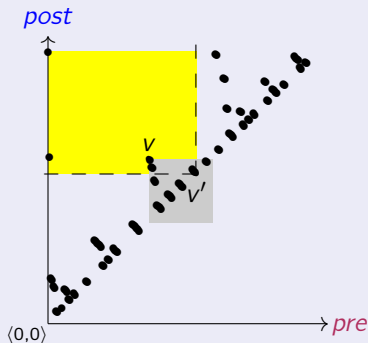
Find intermediary context node sequence of elements with tag  $t$ .  
Then, for each node  $v'$  in this sequence, evaluate  $window(ancestor::s, v')$ .

❷ **Backward mode.**

Find intermediary context node sequence of elements with tag  $s$ .  
Then, for each node  $v$  in this sequence, check whether  $window(descendant::t)$  yields at least one node  $v'$ . If no such  $v'$  is found, drop  $v$ .

**NB.** Based on the `descendant`  $\leftrightarrow$  `ancestor` symmetry.

# XPath symmetries and window size



**Note:** plan ① evaluates the    , plan ② the     window(s).

# XPath symmetries

- Note that plan ② corresponds to the **symmetrical equivalent** of the original location path:

## XPath Symmetry

①	/descendant::t/ancestor::s
	↕
②	/descendant-or-self::s[descendant::t]

Can you suggest a proof for the symmetry?

Why is axis descendant-or-self used in ②?

- The **query rewrite** ① → ② could also be initiated on the XQuery (XPath) source level.

# More XPath symmetries

## XPath Symmetries (due to Dan Olteanu, *et.al.*)

---

```

      ⋮
descendant::t/parent::s ↔ descendant-or-self::s[child::t]
  child::t/parent::s ↔ self::s[child::t]
    c/child::t/ancestor::s ↔ c[child::t]/ancestor-or-self::s
/descendant::t/preceding::s ↔ /descendant::s[following::t]
      ⋮

```

---



## Part XV

# Updating XML Documents

# Outline of this part

- 38 Updating XML Trees
  - Update Specification
  - XUpdate
- 39 Impact on XPath Accelerator Encoding
- 40 Impacts on Other Encoding Schemes

# Updating XML trees

Throughout the course, up to now, we have **not** been looking into **updates** to XML documents at all.

- If we want to discuss *efficiency/performance* issues w.r.t. mappings of XML documents to databases, though, we need to take **modifications** into account as well as pure retrieval operations.
- As always during *physical database design*, there is a trade-off between accelerated retrieval and update performance.
- While there is a whole host of languages for *querying* (*i.e.*, read access to) XML documents, there is not yet an *update language* (for write access) that has been agreed upon.
- We will briefly sketch the XUpdate language, currently under consideration in the XML and XQuery communities.<sup>48</sup>

---

<sup>48</sup><http://xmldb-org.sourceforge.net/xupdate/>

# Updates and tree structures

During our discussion of XQuery, we have seen that *tree construction* has been a major concern. Updates, however, cannot be expressed with XQuery.

- Yet, we need to be able to specify *modifications* of existing XML documents/fragments as well.
- The basic necessary update functionality is largely agreed upon, syntax and semantic details, however, are subject to discussion.
- We certainly need to be able to express:
  - ▶ modification of all aspects (name, attributes, attribute values, text contents) of XML nodes, and
  - ▶ modifications of the tree structure (add/delete/move nodes or subtrees).
- Like in the SQL case, target node(s) of such modifications should be identifiable by means of expressions in an/any XML query language.

# XUpdate: Identify, then modify

## XUpdate element update statement

```
<xupdate:modifications>
  ...
  <xupdate:update select="p">
    c
  </xupdate:update>
  ...
</xupdate:modifications>
```

- 1 Given a context node, evaluate XPath expression *p* to **identify** an XML element node *v*.
- 2 The content of element *v* will be **modified** to be *c*. Otherwise, the updated tree does not change.

(Compare with the XSLT approach!)

## XUpdate: Text node updates

Obviously, the kind of **c** determines the overall impact on the updated tree and its encoding.

### XUpdate: replacing text by text

```
<a>
  <b id="0">foo</b>
  <b id="1">bar</b>
</a>
```



```
<a>
  <b id="0">foo</b>
  <b id="1">foo</b>
</a>
```

```
<xupdate:update select="//b[@id = 1]">
  foo
</xupdate:update>
```

- New content **c**: a **text node**.

## XUpdate: Text node updates

Translated into, e.g., the XPath Accelerator representation, we see that

- Replacing text nodes by text nodes has **local impact** only on the *pre/post* encoding of the updated tree.

### XUpdate statement leads to local relational update

<i>pre</i>	<i>post</i>	...	<i>text</i>
0	4		NULL
1	1		NULL
2	0		foo
3	3		NULL
4	2		bar

⇒

<i>pre</i>	<i>post</i>	...	<i>text</i>
0	4		NULL
1	1		NULL
2	0		foo
3	3		NULL
4	2		foo

- Similar observations can be made for updates on comment and processing instruction nodes.

# XUpdate: Structural updates

## XUpdate: inserting a new subtree

```

<a>
  <b><c><d/><e/></c></b>
  <f><g/>
    <h><i/><j/></h>
  </f>
</a>

```

↓

```

<a>
  <b><c><d/><e/></c></b>
  <f><g><k><l/><m/></k></g>
    <h><i/><j/></h>
  </f>
</a>

```

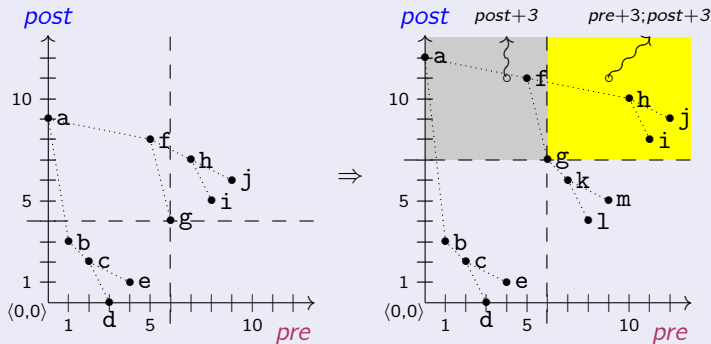
<xupdate:update select="/a/f/g">  
 <k><l/><m/></k>  
 </xupdate:update>

**Question:** What are the effects w.r.t. our structure encoding...?



# XUpdate: Global impact on encoding

## Global shifts in the *pre/post* Plane



# XUpdate: Global impact on *pre/post* plane

Insert a subtree of  $n$  nodes below parent element  $v$

- ①  $post(v) \leftarrow post(v) + n$
- ②  $\forall v' \in v/following::node():$   
 $pre(v') \leftarrow pre(v') + n; post(v') \leftarrow post(v') + n$
- ③  $\forall v' \in v/ancestor::node():$   
 $post(v') \leftarrow post(v') + n$

Cost (tree of  $N$  nodes)

$$\underbrace{O(N)}_{\textcircled{2}} + \underbrace{O(\log N)}_{\textcircled{3}}$$

Update cost

③ is not so much a problem of cost but of **locking**. Why?

# Updates and fixed-width encodings

## Theoretical result [Milo *et.al.*, PODS 2002]

There is a sequence of updates (subtree insertions) for any persistent<sup>49</sup> tree encoding scheme  $\mathcal{E}$ , such that  $\mathcal{E}$  **needs labels of length**  $\Omega(N)$  to encode the resulting tree of  $N$  nodes.

- **Fixed-width** tree encodings (like XPath Accelerator) are inherently **static**.

⇒ Non-solutions:

- ▶ **Gaps** in the encoding,
- ▶ encodings based on **decimal fractions**.

---

<sup>49</sup>A node keeps its initial encoding label even if its tree is updated.

## A variable-width tree encoding: ORDPATH

Here we look at a particular variant of a hierarchical numbering scheme, optimized for updates.

- The **ORDPATH** encoding (used in MS SQL Server<sup>TM</sup>) assigns node labels of **variable length**.

### ORDPATH labels for an XML fragment

- 1 The fragment root receives label 1.
  - 2 The  $n$ th ( $n = 1, 2, \dots$ ) child of a parent node labelled  $p$  receives label  $p \cdot (2 \cdot n - 1)$ .
- Internally, ORDPATH labels are not stored as  $\cdot$ -separated ordinals but using a prefix-encoding (similarities with Unicode).

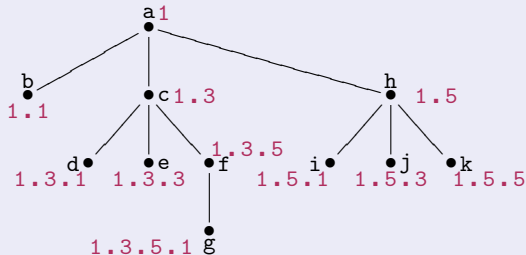
# ORDPATH encoding: Example

## ORDPATH encoding of a sample XML fragment

```

<a>
  <b/>
  <c>
    <d/><e/>
    <f><g/></f>
  </c>
  <h>
    <i/><j/><k/>
  </h>
</a>

```



### Note:

- **Lexicographic** order of ORDPATH labels  $\equiv$  document order
- ⇒ **Clustered index on ORDPATH labels** will be helpful.

## ORDPATH: Insertion between siblings

In ORDPATH, the **insertion of new nodes** between two existing sibling nodes is referred to as “*caretting in*” (caret  $\hat{=}$  insertion mark,  $\wedge$ ).

### ORDPATH: node insertion

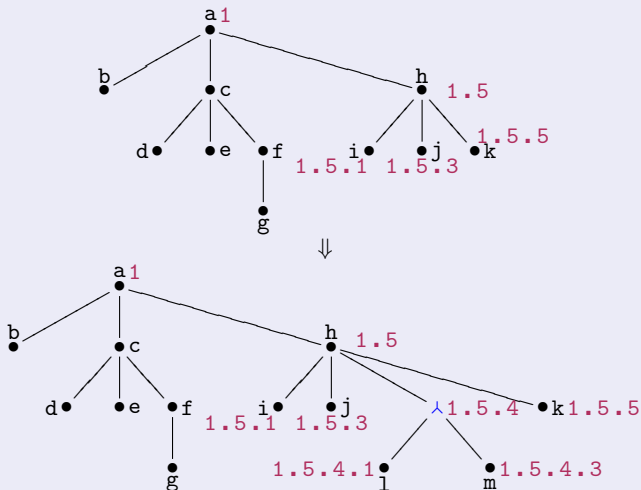
Let  $(v_1, \dots, v_n)$  denote a sequence of nodes to be inserted between two existing sibling nodes with labels  $p.s$  and  $p.(s+2)$ ,  $s$  odd. After insertion, the new label of  $v_i$  is

$$p.(s+1).(2 \cdot i - 1) .$$

Label  $p.(s+1)$  is referred to as a **caret**.

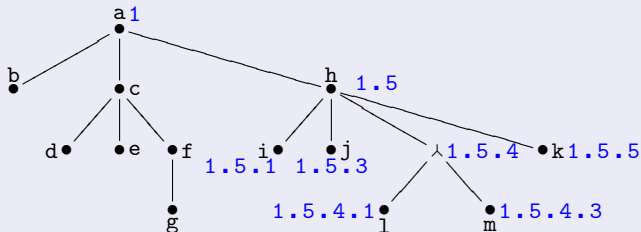
# ORDPATH: Insertion between siblings (Example)

Insertion of (<l/>, <m/>) between <j/> and <k/>



# ORDPATH: Insertion between siblings

## ORDPATH: Insertions at arbitrary locations?



Determine ORDPATH label of new node  $v$  inserted

- ① to the right of  $\langle k/\rangle$ ,
- ② to the left of  $\langle i/\rangle$ ,
- ③ between  $\langle j/\rangle$  and  $\langle l/\rangle$ ,
- ④ between  $\langle l/\rangle$  and  $\langle m/\rangle$ ,



# Processing XQuery and ORDPATH

Is ORDPATH a suitable encoding  $\mathcal{E}$ ?

Mapping core operations of the XQuery processing model to operations on ORDPATH labels:

## $v/\text{parent}::\text{node}()$

- ① Let  $p.m.n$  denote  $v$ 's label ( $n$  is odd).
- ② If the rightmost ordinal ( $m$ ) is even, remove it. Goto ②.

In other words: the carets ( $\wedge$ ) do not count for ancestry.

## $v/\text{descendant}::\text{node}()$

- ① Let  $p.n$  denote  $v$ 's label ( $n$  is odd).
- ② Perform a lexicographic index range scan from  $p.n$  to  $p.(n+1)$ —the *virtual following sibling* of  $v$ .

## ORDPATH: Variable-length node encoding

- Using (4 byte) integers for all numbers in the hierarchical numbering scheme is an obvious waste of space!
- Fewer (and variable number of) bits are typically sufficient;
- they may bear the risk of running out of new numbers, though. In that case, even ORDPATH cannot avoid *renumbering*.
  - ▶ In principle, though, *no bounded* representation can absolutely avoid the need for renumbering.
- Several approaches have been proposed so as to alleviate the problem, for instance:
  - ▶ use a variable number of bits/bytes, akin to Unicode,
  - ▶ apply some (order-preserving) hashing schemes to shorten the numbers,
  - ▶ ...

## ORDPATH: Variable-length node encoding

- For a 10 MB XML sample document, the authors of ORDPATH observed label lengths between 6 and 12 bytes (using Unicode-like compact representations).
- Since ORDPATH labels encode **root-to-node** paths, node labels share **common prefixes**.

ORDPATH labels of <1/> and <m/>

1 . 5 . 4 . 1

1 . 5 . 4 . 3

⇒ Label comparisons often need to inspect encoding bits at the far right.

- MS SQL Server<sup>TM</sup> employs further path encodings organized in **reverse** (node-to-root) order.
- **Note:** Fixed-length node IDs (such as, e.g., preorder ranks) typically fit into CPU registers.

## Part XVI

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

# Outline of this part

## 41 Serialization

- Problem
- Serialization & *Pre/Post* Encoding

## 42 Shredding ( $\mathcal{E}$ )


## 43 Completing the *Pre/Post* Encoding Table Layout

## 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** ( $\mathcal{E}^{-1}$ ).

 <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 `acce1` 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  $\text{not}(S.\text{empty}()) \wedge \text{post}(S.\text{top}()) < \text{post}(v)$  do  
        | print('</',  $\text{name}(S.\text{top}())$ , '>');  
        |  $S.\text{pop}()$ ;  
    if  $\text{kind}(v) = \text{elem}$  then  
        | print('<',  $\text{name}(v)$ , '>');  
        |  $S.\text{push}(v)$ ;  
    else  
        | { process other node kinds here }  
  
while  $\text{not}(S.\text{empty}())$  do  
    | print('</',  $\text{name}(S.\text{top}())$ , '>');  
    |  $S.\text{pop}()$ ;
```



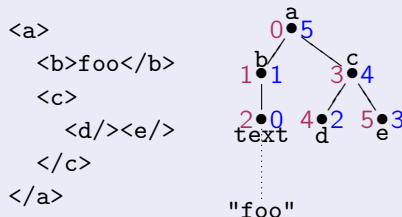
## Serialization & *pre/post* encoding

- 1 To serialize an encoded XML document in its entirety, invoke *serialize*(*acce1*).
- 2 To serialize the XML fragment with root element *v*, invoke *serialize*(*·*) on the result of query *Q*, where

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

# Serialization: Example (1)

## Sample XML fragment and *pre/post* encoding



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

## Stack S

S				
<i>pre</i>	<i>post</i>	<i>kind</i>	<i>tag</i>	<i>text</i>
0	5	<i>elem</i>	a	NULL

## Output (console)

<a>

# Serialization: Example (3)

## Scan of *pre/post* encoding

accel				
<i>pre</i>	<i>post</i>	<i>kind</i>	<i>tag</i>	<i>text</i>
0	5	<i>elem</i>	a	NULL
1	1	<i>elem</i>	b	NULL
2	0	<i>text</i>	NULL	foo
3	4	<i>elem</i>	c	NULL
4	2	<i>elem</i>	d	NULL
5	3	<i>elem</i>	e	NULL

## Stack S

S				
<i>pre</i>	<i>post</i>	<i>kind</i>	<i>tag</i>	<i>text</i>
1	1	<i>elem</i>	b	NULL
0	5	<i>elem</i>	a	NULL

## Output (console)

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

# Serialization: Example (4)

## Scan of *pre/post* encoding

→

accel				
<i>pre</i>	<i>post</i>	<i>kind</i>	<i>tag</i>	<i>text</i>
0	5	<i>elem</i>	a	NULL
1	1	<i>elem</i>	b	NULL
2	0	<i>text</i>	NULL	foo
3	4	<i>elem</i>	c	NULL
4	2	<i>elem</i>	d	NULL
5	3	<i>elem</i>	e	NULL

## Stack S

S				
<i>pre</i>	<i>post</i>	<i>kind</i>	<i>tag</i>	<i>text</i>
1	1	<i>elem</i>	b	NULL
0	5	<i>elem</i>	a	NULL

## Output (console)

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

# Serialization: Example (5)

## Scan of *pre/post* encoding

accel				
<i>pre</i>	<i>post</i>	<i>kind</i>	<i>tag</i>	<i>text</i>
0	5	<i>elem</i>	a	NULL
1	1	<i>elem</i>	b	NULL
2	0	<i>text</i>	NULL	foo
→ 3	4	<i>elem</i>	c	NULL
4	2	<i>elem</i>	d	NULL
5	3	<i>elem</i>	e	NULL

## Stack S

S				
<i>pre</i>	<i>post</i>	<i>kind</i>	<i>tag</i>	<i>text</i>
3	4	<i>elem</i>	c	NULL
0	5	<i>elem</i>	a	NULL

## Output (console)

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

# Serialization: Example (5)

## Scan of *pre/post* encoding

accel				
<i>pre</i>	<i>post</i>	<i>kind</i>	<i>tag</i>	<i>text</i>
0	5	<i>elem</i>	a	NULL
1	1	<i>elem</i>	b	NULL
2	0	<i>text</i>	NULL	foo
3	4	<i>elem</i>	c	NULL
4	2	<i>elem</i>	d	NULL
5	3	<i>elem</i>	e	NULL

→

## Stack S

S				
<i>pre</i>	<i>post</i>	<i>kind</i>	<i>tag</i>	<i>text</i>
4	2	<i>elem</i>	d	NULL
3	4	<i>elem</i>	c	NULL
0	5	<i>elem</i>	a	NULL

## Output (console)

```

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

```

# Serialization: Example (6)

## Scan of *pre/post* encoding

accel				
<i>pre</i>	<i>post</i>	<i>kind</i>	<i>tag</i>	<i>text</i>
0	5	<i>elem</i>	a	NULL
1	1	<i>elem</i>	b	NULL
2	0	<i>text</i>	NULL	foo
3	4	<i>elem</i>	c	NULL
4	2	<i>elem</i>	d	NULL
→ 5	3	<i>elem</i>	e	NULL

## Stack S

S				
<i>pre</i>	<i>post</i>	<i>kind</i>	<i>tag</i>	<i>text</i>
5	3	<i>elem</i>	e	NULL
3	4	<i>elem</i>	c	NULL
0	5	<i>elem</i>	a	NULL

## Output (console)

```

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

```



# Serialization: Example (7)

## Scan of *pre/post* encoding

accel				
<i>pre</i>	<i>post</i>	<i>kind</i>	<i>tag</i>	<i>text</i>
0	5	<i>elem</i>	a	NULL
1	1	<i>elem</i>	b	NULL
2	0	<i>text</i>	NULL	foo
3	4	<i>elem</i>	c	NULL
4	2	<i>elem</i>	d	NULL
5	3	<i>elem</i>	e	NULL

✗

## Stack S

S				
<i>pre</i>	<i>post</i>	<i>kind</i>	<i>tag</i>	<i>text</i>
3	4	<i>elem</i>	c	NULL
0	5	<i>elem</i>	a	NULL

## Output (console)

```

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

```

# Serialization: Example (8)

## Scan of *pre/post* encoding

accel				
<i>pre</i>	<i>post</i>	<i>kind</i>	<i>tag</i>	<i>text</i>
0	5	<i>elem</i>	a	NULL
1	1	<i>elem</i>	b	NULL
2	0	<i>text</i>	NULL	foo
3	4	<i>elem</i>	c	NULL
4	2	<i>elem</i>	d	NULL
5	3	<i>elem</i>	e	NULL

✗

## Stack S

S				
<i>pre</i>	<i>post</i>	<i>kind</i>	<i>tag</i>	<i>text</i>
0	5	<i>elem</i>	a	NULL

## Output (console)

```

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

```

# Serialization: Example (9)

## Scan of *pre/post* encoding

accel				
<i>pre</i>	<i>post</i>	<i>kind</i>	<i>tag</i>	<i>text</i>
0	5	<i>elem</i>	a	NULL
1	1	<i>elem</i>	b	NULL
2	0	<i>text</i>	NULL	foo
3	4	<i>elem</i>	c	NULL
4	2	<i>elem</i>	d	NULL
5	3	<i>elem</i>	e	NULL

✗

## Stack S

S				
<i>pre</i>	<i>post</i>	<i>kind</i>	<i>tag</i>	<i>text</i>

## Output (console)

```

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

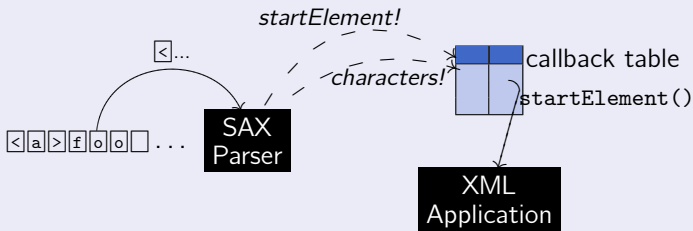
```

# 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,  $\sqcup$ , NULL, doc, NULL, NULL $\rangle$ );  
pre  $\leftarrow$  pre + 1;
```

### *startElement*(*t*, (*a*<sub>1</sub>, *v*<sub>1</sub>), . . . , (*a*<sub>*n*</sub>, *v*<sub>*n*</sub>))

```
v  $\leftarrow$   $\langle$ pre,  $\sqcup$ , S.top().pre, elem, t, NULL $\rangle$ ;  
S.push(v);  
pre  $\leftarrow$  pre + 1;  
{ process attributes ai 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, 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”)

- ❶ What is the maximum depth of stack  $S$ ?
- ❷ How can the shredder detect that the input is not well-formed (improper tag nesting)?
- ❸ 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*

$\langle 0, \sqcup, \text{NULL}, \text{doc}, \text{NULL}, \text{NULL} \rangle$

## Table *accel*

<i>pre</i>	<i>post</i>	<i>par</i>	<i>kind</i>	<i>tag</i>	<i>text</i>

# 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*

```
<1, ␣, 0, elem, a, NULL>
<0, ␣, NULL, doc, NULL, NULL>
```

## Table *accel*

<i>pre</i>	<i>post</i>	<i>par</i>	<i>kind</i>	<i>tag</i>	<i>text</i>

## 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*

<i>pre</i>	<i>post</i>	<i>par</i>	<i>kind</i>	<i>tag</i>	<i>text</i>

# 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*

<i>pre</i>	<i>post</i>	<i>par</i>	<i>kind</i>	<i>tag</i>	<i>text</i>
3	0	2	<i>text</i>	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*

<i>pre</i>	<i>post</i>	<i>par</i>	<i>kind</i>	<i>tag</i>	<i>text</i>
3	0	2	<i>text</i>	NULL	foo
2	1	1	<i>elem</i>	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*

<i>pre</i>	<i>post</i>	<i>par</i>	<i>kind</i>	<i>tag</i>	<i>text</i>
3	0	2	<i>text</i>	NULL	foo
2	1	1	<i>elem</i>	b	NULL

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

- As discussed up to now, table `acce1` lacks some critical details to really support XQuery evaluation. We need to
  - ➊ add support for **attribute** nodes,
  - ➋ reflect the fact that **multiple tree fragments** may be constructed by an XQuery compression (with more than one fragment “alive” at a time),
  - ➌ 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

<i>pre</i>	<i>post</i>	<i>...</i>	<i>tag</i>	<i>...</i>
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.

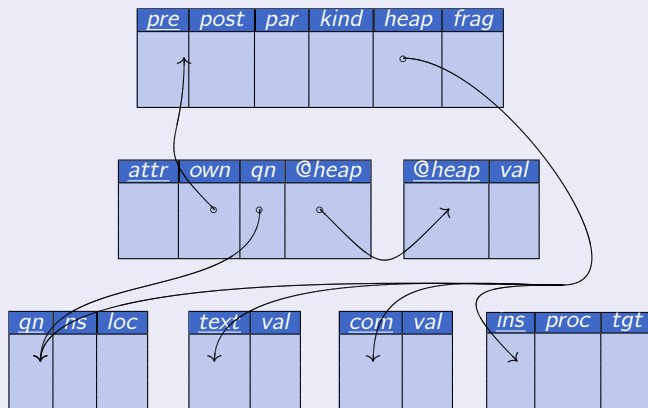
# Relational encoding in MonetDB/XQuery

In MonetDB/XQuery,

- 1 the central table `acce1` is extended by a column *frag* which identifies the **fragment** a node belongs to,
- 2 **attribute nodes** live in a separate table, using column *pre* as a foreign key to identify the owner element,
- 3 the **qualified names of tags and attributes** (*ns:loc*) are held in separate tables (sharing!),
- 4 any **textual content** (text, comments, processing instructions) resides in separate tables,
- 5 finally, a table of referenced **documents** (referred to via `doc(·)`) is maintained.

# Table layout in MonetDB/XQuery

Table schemas ( $\circ \rightarrow$  denotes foreign key relationship)



## MonetDB/XQuery: Encoded fragment

```

<my:a lv="0">
  <b lv="1">
    <!--two foos-->
    <c>foo</c><d>foo</d>
  </b>
  <b lv="1"/>
</my:a>

```

<i>pre</i>	<i>post</i>	<i>par</i>	<i>kind</i>	<i>heap</i>	<i>frag</i>
0	7	NULL	<i>elem</i>	0	0
1	5	0	<i>elem</i>	2	0
2	3	1	<i>com</i>	0	0
3	2	1	<i>elem</i>	3	0
4	1	3	<i>text</i>	0	0
5	4	1	<i>elem</i>	4	0
6	3	5	<i>text</i>	0	0
7	6	0	<i>elem</i>	2	0

<i>attr</i>	<i>own</i>	<i>qn</i>	<i>@heap</i>
0*	0	1	0
1*	1	1	1
2*	7	1	1

<i>@heap</i>	<i>val</i>
0	"0"
1	"1"

<i>qn</i>	<i>ns</i>	<i>loc</i>
0	<i>ns<sup>my</sup></i>	"a"
1		"lv"
2		"b"
3		"c"
4		"d"

<i>text</i>	<i>val</i>
0	"foo"

<i>com</i>	<i>val</i>
0	"two foos"

<i>ins</i>	<i>proc</i>	<i>tgt</i>

# MonetDB/XQuery: Encoded fragment

- Column *frag* indicates the fragment a node belongs to. Windows for axes following, preceding modified to guarantee that **axis evaluation does not escape fragment.**
- Note:** Size of *QName* table typically **independent of fragment size** (usually  $\leq 20$  rows).
  - Value  $ns^{my}$  encodes namespace with prefix *my* (prefixes immaterial for *QName* comparison).
- Identifiers of attributes ( $0^*, \dots$ ) distinguishable from node ids.<sup>50</sup>  
**Document order of attributes derived from document order of owner element** (column *own*).
- Generally ignored here: **white space only text nodes.**




<sup>50</sup>Most significant bit set, for example

## Part XVII

# Staircase Join—Tree-Aware Relational (X)Query Processing

# Outline of this part

- 44 XPath Accelerator—Tree aware relational XML representation
  - Enhancing Tree Awareness
- 45 Staircase Join
  - Tree Awareness
  - Context Sequence Pruning
  - Staircases
- 46 Injecting  into PostgreSQL
- 47 Outlook: More on Performance Tuning in MonetDB/XQuery

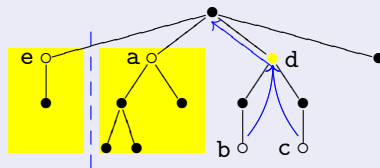


# Enhancing tree awareness

- We now know that the XPath Accelerator is a true **isomorphism** with respect to the XML skeleton **tree structure**.
  - ▶ Witnessed by our discussion of **shredder** ( $\mathcal{E}$ ) and **serializer** ( $\mathcal{E}^{-1}$ ).
- We will now see how the database kernel can benefit from a more elaborate **tree awareness** (beyond document order and semantics of the four major XPath axes).
- This will lead to the design of **staircase join**  $\Join$ , the core of MonetDB/XQuery's XPath engine.
  - ▶ We will also discuss issues of how to tune  $\Join$  to get the most out of modern CPUs and memory architectures.

# Tree awareness?

Document order and XPath semantics aside, what are further **tree properties** of value to a relational XML processor?



- 1 The **size of the subtree** rooted in node a is 5
- 2 The leaf-to-root **paths** of nodes b, c **meet** in node d
- 3 The **subtrees** rooted in e and a are necessarily **disjoint**

## Tree awareness ①: Subtree size

We have seen that tree property **subtree size** (① on previous slide) is implicitly present in a *pre/post*-based tree encoding:

$$post(v) - pre(v) = size(v) - level(v)$$

- To exploit property **subtree size**, we were able to find a means on the **SQL language level**, *i.e.*, **outside the database kernel**.
- ⇒ This led to *window shrink-wrapping* for the XPath descendant axis.

# Tree awareness on the SQL level

## Shrink-wrapping for the descendant axis

$$Q \equiv (c)/\text{following}::\text{node}()/\text{descendant}::\text{node}()$$

*path(Q)*

```
SELECT  DISTINCT  $v_2.pre$ 
FROM    accel  $v_1$ , accel  $v_2$ 
WHERE    $v_1.pre > c.pre$ 
        AND  $v_1.pre < v_2.pre$ 
        AND  $v_1.post > c.post$ 
        AND  $v_1.post > v_2.post$ 
        AND  $v_2.pre \leq v_1.post + h$  AND  $v_2.post \geq v_1.pre + h$ 
ORDER BY  $v_2.pre$ 
```

## Tree awareness ②: Meeting ancestor paths

- Evaluation of axis `ancestor` can clearly benefit from knowledge about the exact element node where several given **node-to-root paths meet**.
  - ▶ For example:  
For context nodes  $c_1, \dots, c_n$ , determine their **lowest common ancestor**  $v = lca(c_1, \dots, c_n)$ .  
 $\Rightarrow$  Above  $v$ , produce result nodes once only.  
(This still produces duplicate nodes below  $v$ .)
- This knowledge *is* present in the encoding but is **not as easily expressed on the level of commonly available relational query languages** (such as, SQL or relational algebra).

## Flashback: XPath: Ensuring order is not for free

The strict XPath requirement to construct a result in document order may imply **sorting effort** depending on the actual XPath implementation strategy used by the processor.

```
(<x>
  <x><y id="0"/></x>
  <y id="1"/>
</x>)/descendant-or-self::x/child::y    ⇒    (<y id="0"/>,
  <y id="1"/>)
```

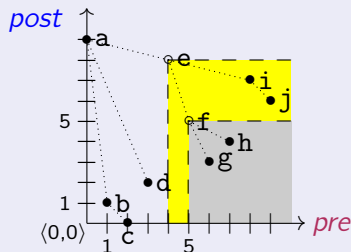
- In many implementations, the `descendant-or-self::x` step will yield the context node sequence (**<x>...</x>**, `<x>...</x>`) for the `child::y` step.
- Such implementations thus will typically extract `<y id="1"/>` before `<y id="0"/>` from the input document.

Flashback:  $(e, f) / \text{descendant}::\text{node}()$ 

## Context &amp; frag. encodings

context		
<i>pre</i>	<i>post</i>	...
5	5	
4	8	

accel		
<i>pre</i>	<i>post</i>	...
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

```

## Tree awareness ③: Disjoint subtrees

- An XPath location step  $cs/\alpha$  is evaluated for a context node **sequence**  $cs$ .
  - ▶ This “*set-at-a-time*” processing mode is key to the efficient evaluation of queries against bulk data. We want to map this into **set-oriented operations** on the RDBMS.  
(Remember: location step is translated into **join** between context node sequence and document encoding table *accel*.)
- But: If two context nodes  $c_{i,j} \in cs$  are in  $\alpha$ -relationship, **duplicates** and **out-of-order** results may occur.
  - ▶ Need efficient way to identify the  $c_i \in cs$  which are *not* in  $\alpha$ -relationship with any other  $c_j$   
(for  $\alpha = \text{descendant}$ : “ $c_{i,j}$  in disjoint subtrees?”).



# Staircase Join: An injection of tree awareness

Since we fail to explain tree properties ② and ③ at the relational language level interface, we opt to **invade the database kernel** in a controlled fashion.<sup>51</sup>

- Inject a new relational operator, **staircase join**  $\Join$ , into the relational query engine.
- Query translation and optimization in the presence of  $\Join$  continues to work like before (e.g., selection pushdown).
- The  $\Join$  algorithm encapsulates the necessary tree knowledge.  $\Join$  is a **local change** to the database kernel.

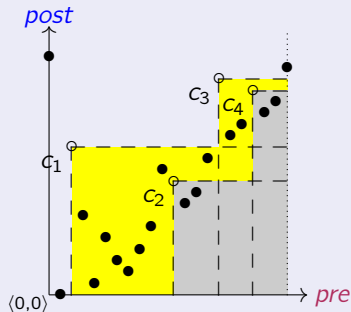
---

<sup>51</sup>Remember: All of this is optional. XPath Accelerator is a purely relational XML document encoding, working on top of *any* RDBMS.

# Tree awareness: Window overlap, coverage

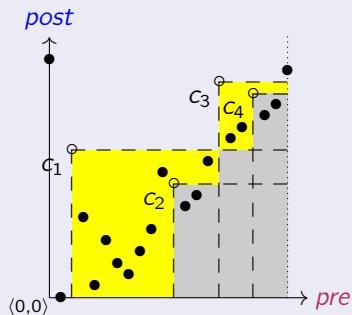
Location step  $(c_1, c_2, c_3, c_4)/\text{descendant}::\text{node}()$ . The pairs  $(c_1, c_2)$  and  $(c_3, c_4)$  are in descendant-relationship:

## Window overlap and coverage (descendant axis)

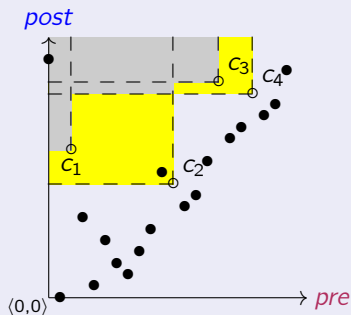


# Tree awareness: Window overlap, coverage

Axis window overlap  
(descendant axis)

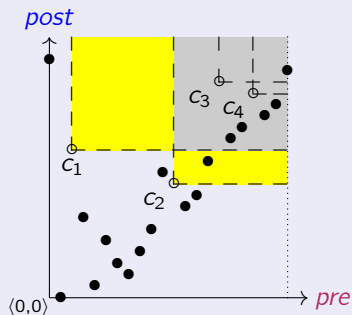


Axis window overlap  
(ancestor axis)

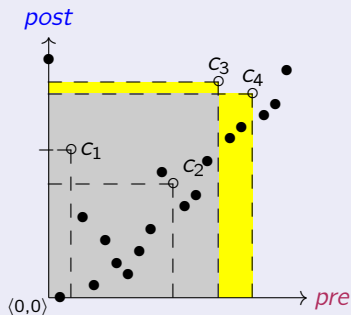


# Tree awareness: Window overlap, coverage

Axis window overlap  
(following axis)



Axis window overlap  
(preceding axis)



## Context node sequence pruning

We can turn these observations about axis window overlap and coverage into a simple strategy to **prune the initial context node sequence** for an XPath location step.

### Context node sequence pruning

Given  $cs/\alpha$ , determine minimal  $cs^- \subseteq cs$ , such that

$$cs/\alpha = cs^-/\alpha .$$

We will see that this minimization leads to axis step evaluation on the *pre/post* plane, which never emits duplicate nodes or out-of-order results.<sup>52</sup>

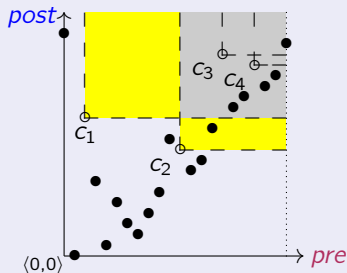
---

<sup>52</sup>The ancestor axis needs a bit more work here.

## Context node pruning: following axis

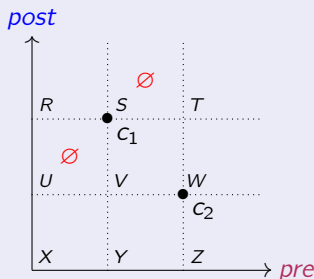
Once context pruning for the following axis is complete, all remaining context nodes relate to each other on the ancestor/descendant axes:

### Covering nodes $c_{1,2}$ in descendant relationship



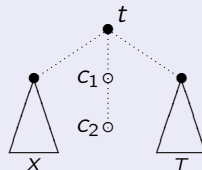
# Empty regions in the *pre/post* plane

Relating two context nodes ( $c_1, c_2$ ) on the plane



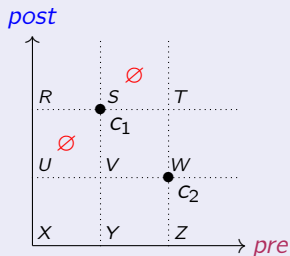
Empty regions?

Given  $c_{1,2}$  on the left, why are the regions  $U, S$  marked  $\emptyset$  guaranteed *to not hold any nodes*?



# Context pruning (following axis)

$(c_1, c_2)/\text{following}::\text{node}()$

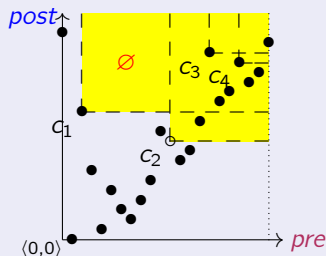


$$\begin{aligned}
 (c_1, c_2)/\text{following}::\text{node}() &\equiv S \cup T \cup W \\
 &\equiv T \cup W \\
 &\equiv (c_2)/\text{following}::\text{node}()
 \end{aligned}$$



# Context pruning (following axis)

## Context pruning (following axis)



## Context pruning (following axis)

Replace context node sequence  $cs$  by singleton sequence  $(c)$ ,  $c \in cs$ , with  $post(c)$  minimal.

## Context pruning (preceding axis)

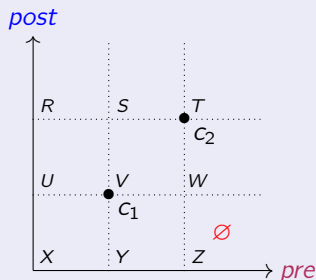
### Context pruning (preceding axis)

Replace context node sequence  $cs$  by singleton sequence  $(c)$ ,  $c \in cs$ , with  $pre(c)$  maximal.

- Regardless of initial context size, axes following and preceding yield simple **single region queries**.
- We focus on descendant and ancestor now.

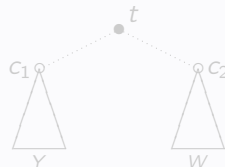
# More empty regions

Remaining context nodes  
 $c_1, c_2$  after pruning for  
descendant axis



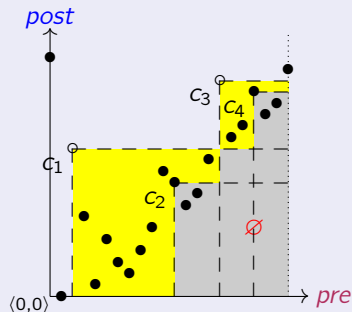
Empty region?

Why is region  $Z$  marked  $\emptyset$   
guaranteed to be empty?



# Context pruning (descendant axis)

## Context pruning (descendant axis)



- The region marked  $\emptyset$  above is a region of type  $Z$  (previous slide). In general, a **non-singleton sequence remains**.

# Context pre-processing: Pruning

*prune\_context*<sub>desc</sub>(*context* : TABLE(*pre*,*post*))

**begin**

result  $\leftarrow$  CREATE TABLE(*pre*,*post*);

*prev*  $\leftarrow$  0;

**foreach** *c* in *context* **do**

    /\* retain node only if *post* rank increases \*/

**if** *c.post* > *prev* **then**

        APPEND *c* TO *result*;

*prev*  $\leftarrow$  *c.post*;

/\* return new context table \*/

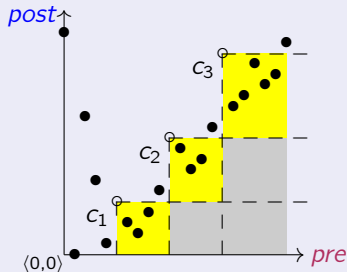
**return** *result*;

**end**

## “Staircases” in the *pre/post* plane

Note that after context pruning, the remaining context nodes form a **proper “staircase”** in the plane. (This is an important assumption in the following.)

### Context pruning & “staircase”



## Flashback: Intersecting ancestor paths

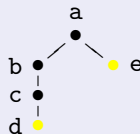
Even with pruning applied, duplicates and out-of-order results may still be generated due to **intersecting ancestor paths**.

- We have observed this before: apply function `ancestors( $c_1, c_2$ )` where  $c_1$  ( $c_2$ ) denotes the element node with tag  $d$  ( $e$ ) in the sample tree below.  
(Nodes  $c_{1,2}$  would *not* have been removed during pruning.)

### Simulate XPath ancestor via parent axis

```
declare function
  ancestors($n as node(*) as node(*)*
{ if (fn:empty($n)) then ()
  else (ancestors($n/..), $n/..)
}
```

### Sample tree



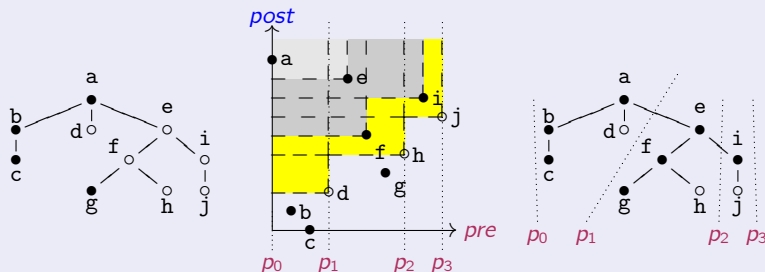
**Remember:** `ancestors((d,e))` yielded `(a,b,a,c)`.

# Separation of ancestor paths

**Idea:** try to **separate** the ancestor paths by defining suitable **cuts** in the XML fragment tree.

- Stop node-to-root traversal if a cut is encountered.

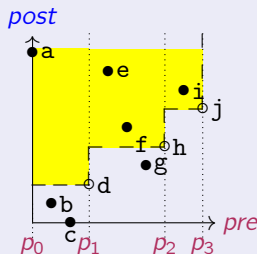
## Path separation (ancestor axis)





# Parallel scan along the *pre* dimension

## Separating ancestor paths



Scan partitions (intervals):  $[p_0, p_1)$ ,  $[p_1, p_2)$ ,  $[p_2, p_3)$ .

- Can scan in **parallel**. Partition results may be concatenated.
- Context pruning reduces numbers of partitions to scan.

# Basic Staircase Join (descendant)

$\sqcup_{\text{desc}}(\text{accel} : \text{TABLE}(\text{pre}, \text{post}), \text{context} : \text{TABLE}(\text{pre}, \text{post}))$

**begin**

$\text{result} \leftarrow \text{CREATE TABLE}(\text{pre}, \text{post});$

**foreach** successive pair  $(c_1, c_2)$  **in** context **do**

$\sqcup_{\text{scanpartition}}(c_1.\text{pre} + 1, c_2.\text{pre} - 1, c_1.\text{post}, <);$

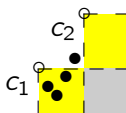
$c \leftarrow \text{last node in context};$

$n \leftarrow \text{last node in accel};$

$\sqcup_{\text{scanpartition}}(c.\text{pre} + 1, n.\text{pre}, c.\text{post}, <);$

**return** result;

**end**



## Partition scan (sub-routine)

*scanpartition*( $pre_1, pre_2, post, \theta$ )

**begin**

**for**  $i$  **from**  $pre_1$  **to**  $pre_2$  **do**

**if**  $accel[i].post \theta post$  **then**

            APPEND  $accel[i]$  TO *result*;

**end**

Notation  $accel[i]$  does not imply random access to document encoding:

- Access is strictly **forward sequential** (also *between* invocations of *scanpartition*( $\cdot$ )).

# Basic Staircase Join (ancestor)

$\sqcup_{\text{anc}}(\text{accel} : \text{TABLE}(\text{pre}, \text{post}), \text{context} : \text{TABLE}(\text{pre}, \text{post}))$

**begin**

$\text{result} \leftarrow \text{CREATE TABLE}(\text{pre}, \text{post});$

$c \leftarrow \text{first node in context};$

$n \leftarrow \text{first node in accel};$

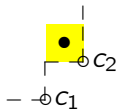
$\text{scanpartition}(n.\text{pre}, c.\text{pre} - 1, c.\text{post}, >);$

**foreach** successive pair  $(c_1, c_2)$  **in** context **do**

$\text{scanpartition}(c_1.\text{pre} + 1, c_2.\text{pre} - 1, c_2.\text{post}, >);$

**return** result;

**end**




## Basic Staircase Join: Summary

- The operation of **staircase join** is perhaps most closely described as **merge join** with a **dynamic range predicate**: the join predicate traces the staircase boundary:
  - ▶  $\Join$  scans the *accel* and *context* tables and populates the *result* table sequentially in document order,
  - ▶  $\Join$  scans both tables once for an entire context sequence,
  - ▶  $\Join$  never delivers duplicate nodes.
- $\Join$  works correctly only if *prune\_context*( $\cdot$ ) has previously been applied.
  - ▶ *prune\_context*( $\cdot$ ) may be **inlined** into  $\Join$ , thus performing context pruning *on-the-fly*.



# Pruning on-the-fly

 `desc(accel:TABLE(pre, post), context:TABLE(pre, post))`

**begin**

`result`  $\leftarrow$  CREATE TABLE(*pre*, *post*);

`c`<sub>1</sub>  $\leftarrow$  first node in context;

**while** (`c`<sub>2</sub>  $\leftarrow$  next node in context) **do**

**if** `c`<sub>2</sub>.*post* < `c`<sub>1</sub>.*post* **then**

        | /\* prune \*/

**else**

        | scanpartition(`c`<sub>1</sub>.*pre* + 1, `c`<sub>2</sub>.*pre* - 1, `c`<sub>1</sub>.*post*, <);

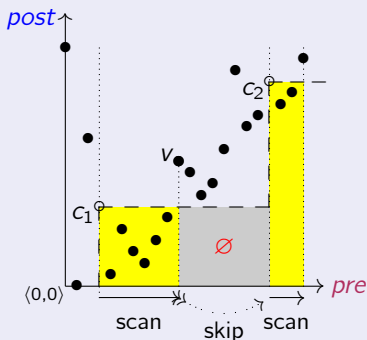
        | `c`<sub>1</sub>  $\leftarrow$  `c`<sub>2</sub>;

**return** `result`;

**end**

# Skip ahead, if possible

$(c_1, c_2)/\text{descendant}::\text{node}()$



- While scanning the partition associated with  $c_{1,2}$ :
  - $v$  is outside staircase boundary, thus not part of the result.
  - No node beyond  $v$  in result ( $\emptyset$ -region of type  $Z$ ).
- ⇒ Can **terminate scan early and skip ahead** to  $pre(c_2)$ .

## Skipping for the descendant axis

*scanpartition*<sub>desc</sub>(*pre*<sub>1</sub>, *pre*<sub>2</sub>, *post*)

```
begin
  for i from pre1 to pre2 do
    if accel[i].post < post then
      | APPEND accel[i] TO result;
    else
      | /* on the first offside node, terminate scan */ break;
  end
end
```

**Note:** keyword **break** transfers control out of innermost enclosing loop (cf. C, Java).



# Effectiveness of skipping

- Enable skipping in *scanpartition*(·). Then, for each node in *context*, we either
  - ① hit a node to be copied into table *result*, or
  - ② encounter an offside node (node *v* on slide 511) which leads to a skip to a known *pre* value ( $\rightarrow$  positional access).
- To produce the final result,  $\Join$  thus never touches more than

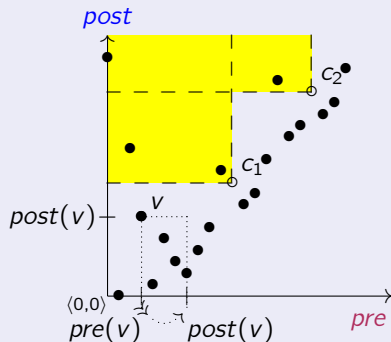
$$| \textit{context} | + | \textit{result} |$$

nodes in the plane (without skipping:  $| \textit{context} | + | \textit{accel} |$ ).

- ▶ In practice:  $> 90\%$  of nodes in table *accel* are skipped.

# Skipping for the ancestor axis

## Skipping over the subtree of $v$



Encounter  $v$  outside staircase boundary



$v$  and subtree below  $v$  in preceding axis of context node.

## How far to skip?

Conservative estimate:

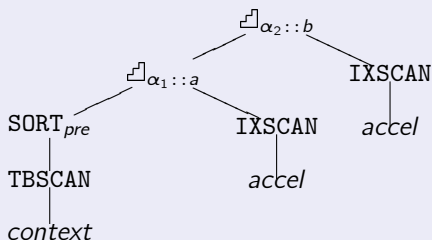
$$\text{size}(v) \geq \text{post}(v) - \text{pre}(v)$$

# Injecting $\Join$ into PostgreSQL

**PostgreSQL** (<http://postgresql.org/>): Conventional disk-based RDBMS, SQL interface.

- Detection of  $\Join$  applicability on SQL level (self-join with conjunctive range selection on columns of type `tree`<sup>53</sup>).

## Algebraic query plan for two-step XPath location path



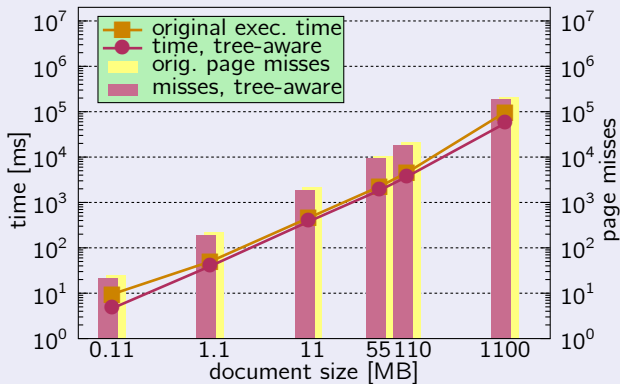
<sup>53</sup>PostgreSQL is highly extensible, also permits introduction of new column types.

# Injecting $\bowtie$ into PostgreSQL

- Create clustered ascending B-tree index on column *pre* of table *accel*.
  - ▶ Standard no-frills PostgreSQL B-tree index, entered with search predicates of the form  $pre \underset{?}{\geq} c.pre$  (*c* context node).
  - ▶ B-tree on column *pre* also used for **skipping**.
- Following performance figures obtained on a 2.2 GHz Dual Intel™ Pentium 4, 2 GB RAM, PostgreSQL 7.3.3.
  - ▶ Compares  $\bowtie$ -enabled (tree-aware) PostgreSQL with vanilla PostgreSQL instance.
  - ▶ Evaluate XPath location path `/descendant::a/ $\alpha$ ::b` on document instances of up to 1.1 GB serialized size.

# Injecting $\bowtie$ into PostgreSQL

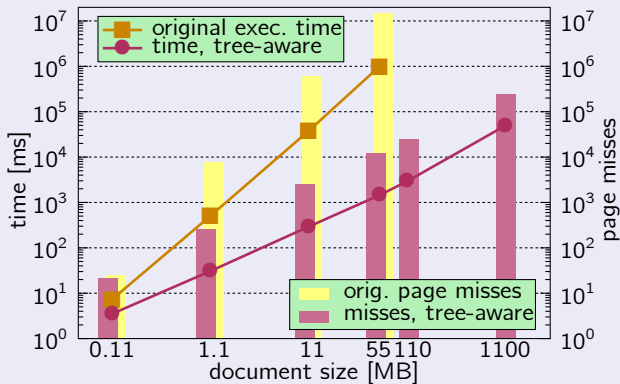
/descendant::a/descendant::b



# Injecting $\alpha$ into PostgreSQL

**For  $\alpha = \text{descendant}$  observe:**

- For *both* PostgreSQL instances, query evaluation time grows **linearly** with the input XML document size (since the results size grows linearly).
- For the original instance, this is due to **window shrink-wrapping** (expressible at the SQL level).

Injecting  $\lambda$  into PostgreSQL`/descendant::a/ancestor::b`

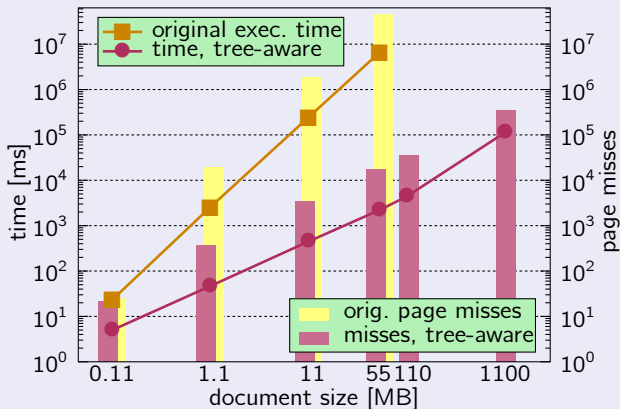
# Injecting $\bowtie$ into PostgreSQL

- **For  $\alpha \in \{\text{ancestor, preceding, following}\}$  observe:**
  - ▶ For the  $\bowtie$ -**enabled** PostgreSQL instance, query evaluation time grows **linearly** with the input XML document (and result) size.  
For the **original** instance, query evaluation time grows **quadratically** ( $| \text{accel} |$  scans of table *accel* performed).
  - ▶ Original instance is incapable of completing experiment in reasonable time ( $> 15$  mins for XML input size of 55 MB).
- **Generally:**
  - ▶ The number of **buffer page misses** (= necessary I/O operations) determines evaluation time.



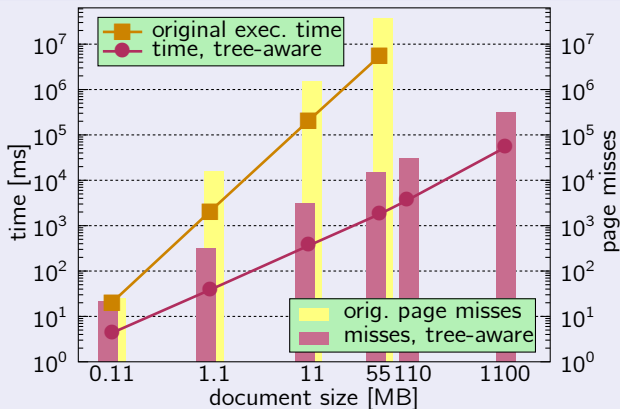
# Injecting $\bowtie$ into PostgreSQL

/descendant::a/preceding::b



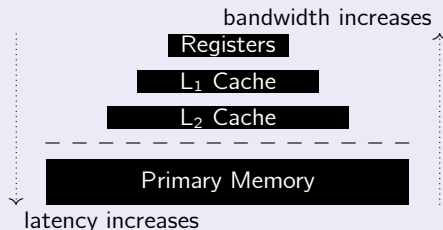
# Injecting $\bowtie$ into PostgreSQL

/descendant::a/following::b



# MonetDB/XQuery: Targetting modern CPU/memory architectures

## Memory Hierarchy



- Computation performed with CPU registers only.
- **Cache miss may escalate:**  $L_1 \rightarrow L_2 \rightarrow \text{RAM}$ , data transport all the way back:  $L_1 \leftarrow L_2 \leftarrow \text{RAM}$ .
- Data transport in **cache line granularity**.

# CPU/cache characteristics

## Intel™ Dual Pentium 4 (Xeon)<sup>54</sup>

### CPU/Cache Characteristics

Clock frequency		2.2 GHz
L <sub>1</sub> /L <sub>2</sub> cache size		8 kB/512 kB
L <sub>1</sub> /L <sub>2</sub> cache line size	$LS_{L_1}/LS_{L_2}$	32 byte/128 byte
L <sub>1</sub> miss latency	$L_{L_1}$	28 cycles $\hat{=}$ 12.7 ns
L <sub>2</sub> miss latency	$L_{L_2}$	387 cycles $\hat{=}$ 176 ns

- For this CPU, a **full cache miss** implies a **stall of the CPU** for  $28 + 387 = 415$  cycles (cy).



<sup>54</sup>Measure these characteristics for your CPU with Stefan Manegold's *Calibrator*, <http://monetdb.cwi.nl/Calibrator/>.

# Staircase join: Wrap-up

- Standard  $B^+$ -tree implementation suffices to support  $\Join$ .
  - ▶ A **single**  $B^+$ -tree indexes the *pre/post* plane as well as the context node sequence.
  - ⇒ Less index pages compete for valuable buffer space.
- $\Join$  derives pruning and skipping information from the plane itself, using **simple integer arithmetic and comparisons**.
  - ▶ Simple  $\Join$  logic leads to **simple memory access pattern and control flow**.
  - ⇒ Branches in inner  $\Join$  loops are highly predictable, facilitating **speculative execution** in the CPU.

Predictable branches?

Explain why!

# Part XVIII

## Relational XQuery Compilation

# Outline of this part

## 48 Where We Are

## 49 XQuery Core

- Restricted XQuery Subset
- Normalization

## 50 Typing

- Type-Based Simplifications

## 51 XQuery Compilation


- Representing Sequences
- Target Language

## 52 Compiling FLWORs

- Example
- Representation Issues
- Relational Algebra for FLWOR Blocks
- Nested Iterations
- Resulting Relational Algebra Plans

## Where we are

We have been discussing an infrastructure for the **relational** representation of XML documents:

- a **relational tree encoding**  $\mathcal{E}$ , the XPath Accelerator,
- support for efficient **XPath** location step processing using its *pre/post* numbering scheme,
- possibilities to enhance relational DBMSs by a specialized, and **tree aware** processing algorithm, **Staircase Join** .

We will now focus on the translation of **XQuery** expressions into relational execution plans.

- We will discuss the translation of a **subset** of XQuery.
- The compiler will emit expressions over a (rather restricted) classical variant of **relational algebra**.



# Source language: XQuery Core

## Supported XQuery Core Dialect


- literals
- sequences ( $e_1, e_2$ )
- variables ( $\$v$ )
- `let...return`
- `for...return`
- `for...[at  $\$v$ ...]return`
- `if...then...else`
- `typeswitch...case...default`
- `element {...} {...}`
- `text {...}`
- XPath ( $e/\alpha$ )
- function application
- document order ( $e_1 \ll e_2$ )
- node identity ( $e_1 \text{ is } e_2$ )
- arithmetics (+, -, \*, idiv)
- `fn:doc()`
- `fn:root()`
- `fn:data()`
- `fn:distinct-doc-order()`
- `fn:count()`
- `fn:sum()`
- `fn:empty()`
- `fn:position()`
- `fn:last()`

# XQuery Core

**XQuery Core** removes “syntactic sugar” from the XQuery surface syntax without sacrificing expressiveness.

- XQuery Core expressions tend to be significantly more verbose than their XQuery equivalents.

Nevertheless, an XQuery compiler benefits:

- 1 Implicit XQuery **semantics** is made **explicit**, and
  - 2 **less constructs** need to be treated in the compiler.
- The process of turning XQuery expressions into XQuery Core is referred to as **normalization**.  
Normalization and XQuery Core are defined in the  XQuery 1.0 and XPath 2.0 Formal Semantics.<sup>55</sup>

---

<sup>55</sup><http://www.w3.org/TR/xquery-semantics/>

## Normalization: Simpler constructs

- In XQuery surface syntax, `for` clauses may bind an arbitrary number  $n$  of variables. In XQuery Core,  $n$  is fixed to be 1.
- Further, there is no `where` clause in XQuery Core.

### Multi-variable `for` into nested single-variable `for` loops

```
for $v_1 in e_1, $v_2 in e_2, ..., $v_n in e_n  
where p  
return e
```

≡

```
for $v_1 in e_1 return  
  for $v_2 in e_2 return  
    ⋮  
    for $v_n in e_n return  
      if (p) then e else ()
```

## Normalization: Less constructs

### ✎ No `some` (`every`) quantifier in XQuery Core

While the XQuery surface syntax supports the existential (universal) quantifier `some` (`every`), no such support is present in XQuery Core.

How can

$$\begin{array}{l} \text{some } \$v \text{ in } e_1 \text{ satisfies } e_2 \\ (\text{every } \$v \text{ in } e_1 \text{ satisfies } e_2) \end{array}$$

be equivalently expressed in XQuery Core? (Hint: use `fn:empty.`)

# Normalization: Implicit to explicit semantics

## Implicit Semantics

Consider the simple XQuery path expression

`/a/b[@c >= 42]` .

In XQuery Core, **implicit semantics** is made **explicit**:

- 1 The context node of the **absolute path** (starting with `/...`) is the root node of the current context node (`.`).
- 2 A **multi-step path** is broken into single steps. An XPath **predicate** is turned into a conditional expression.
- 3 General comparison `>=` has **existential semantics**.
- 4 The comparison operands need to be **atomized**.

# Normalization

## Normalized expression for /a/b[@c >= 42]

```
for $_v0 := fn:root(.) return
  for $_v1 in $_v0/child::a return
    for $_v2 in $_v1/child::b return
      if (some $_v3 in fn:data($_v2/attribute::c) satisfies
          some $_v4 in fn:data(42) satisfies
            op:ge($_v3, $_v4))
      then $_v2
      else ()
```

### Note:

- Both `some...in...satisfies` still non-normalized. Normalization of path steps more complex (see below).
- Builtin function `op:ge` (**g**reater or **e**qual) implements a generic (overloaded) variant of the comparison operator `ge`.

## Normalization of location steps

In the [W3C](#) Formal Semantics documents, normalization is formally defined in terms of function  $\llbracket e \rrbracket$  which maps XQuery expression  $e$  into Core (in a bottom-up fashion).

### Normalize an XPath location step

$$\llbracket e_1/e_2 \rrbracket$$

=

```
fs:distinct-doc-order(  
  let $fs:context as node()* :=  $\llbracket e_1 \rrbracket$  return  
    let $fs:last := fn:count($fs:context) return  
      for $fs:dot at $fs:position in $fs:context return  $\llbracket e_2 \rrbracket$   
)
```

- Names (functions, variables) introduced by the normalization are located in namespace `fs`, unreachable by XQuery surface queries.

# More normalization rules

## Further cases for $\llbracket \cdot \rrbracket$

$$\begin{aligned}\llbracket . \rrbracket &= \$fs:dot \\ \llbracket last() \rrbracket &= \$fs:last \\ \llbracket position() \rrbracket &= \$fs:position\end{aligned}$$

$$\begin{aligned}\llbracket nt \rrbracket &= child::nt \\ \llbracket e_1 // e_2 \rrbracket &= \llbracket e_1 / descendant-or-self::node() / e_2 \rrbracket\end{aligned}$$

$$\begin{aligned}\llbracket following-sibling::nt \rrbracket^{56} \\ = \\ \llbracket let \$e := . return \\ \$e / parent::node() / child::nt[. >> \$e] \rrbracket\end{aligned}$$

<sup>56</sup>Used only if the XQuery processor does not provide builtin support for XPath axes beyond parent, child, descendant(or-self), attribute.



# Static typing

- The resulting normalized XQuery Core queries include many obvious (and not so obvious) hooks for simplification.
- Such opportunities for simplification are largely detectable once the Core query has been **statically typed**.
  - ▶ **Static typing** assigns a **sequence type** to any subexpression of a given Core query.
  - ▶ To achieve this, the static typing process traverses the Core expression tree bottom-up.
  - ▶ **Static** typing does *not* depend on the actual XML input data—only on the query itself (and imported schemas).

## Sequence types (recap)

XQuery uses **sequence types** to describe the type of item sequences:

### Sequence types $t$ (simplified)

```

 $t$  ::= empty-sequence()
      |  $item\ occ$ 

 $occ$  ::= + | * | ? |  $\epsilon$ 
 $item$  ::= atomic | node | item()
 $node$  ::= element() | element( $name$  [ ,  $tyname$  , ] ) | ...
 $name$  ::= * | QName
 $tyname$  ::= QName
 $atomic$  ::= integer | string | double | ...
  
```

## Typing rules

In the **W3C** XQuery Formal Semantics, static typing is defined in terms of **inference rules**.

### Typing a conditional expression

$$\frac{E \vdash e_1 : \text{xs:boolean} \quad E \vdash e_2 : t_2 \quad E \vdash e_3 : t_3}{E \vdash \text{if } (e_1) \text{ then } e_2 \text{ else } e_3 : (t_2 \mid t_3)}$$

- The premise of an inference rule may be empty (*facts*).
- Read  $e : t$  as “*expression e has type t*”.
- **Environment**  $E$ <sup>57</sup> contains a mapping of variables to types; for  $\$v \dots$ , let  $\$v \dots$ , some  $\$v$ /every  $\$v \dots$  enrich the environment:  $E$  becomes  $E + \{v \mapsto t\}$ .

<sup>57</sup>Named *statEnv* in the **W3C** document.

# Typing rules and the environment

## Constants (no premise)

$$\frac{}{E \vdash 42 : \text{xs:integer}} \quad \frac{}{E \vdash \text{"foo"} : \text{xs:string}} \quad \dots$$

## Variable binding (let)

$$\frac{E \vdash e_1 : t_1 \quad E + \{v \mapsto t_1\} \vdash e_2 : t_2}{E \vdash \text{let } \$v := e_1 \text{ return } e_2 : t_2}$$

## Variable reference

$$\frac{}{E + \{v \mapsto t\} \vdash \$v : t}$$

# Type inference: Example

A complete type inference ( $E' = E + \{x \mapsto \text{int}\}$ )

$$\begin{array}{c}
 \frac{}{E \vdash 42 : \text{int}} \quad \frac{\frac{E' \vdash \$x : \text{int} \quad E' \vdash 0 : \text{int}}{E' \vdash \$x \text{ gt } 0 : \text{bool}} \quad \frac{E' \vdash \langle a \rangle : \text{elem}(a) \quad E' \vdash \$x : \text{int}}{E' \vdash \text{if } (\$x \text{ gt } 0) \text{ then } \langle a \rangle \text{ else } \$x : (\text{elem}(a) \mid \text{int})}}{E \vdash \text{let } \$x := 42 \text{ return } \text{if } (\$x \text{ gt } 0) \text{ then } \langle a \rangle \text{ else } \$x : (\text{elem}(a) \mid \text{int})}
 \end{array}$$

- Note how environment  $E$  (and its enrichment  $E'$ ) are passed top-down while the inference of the type  $\text{elem}(a) \mid \text{int}$  proceeds bottom-up.

## Static types vs. dynamic types

The XQuery static typing discipline is **conservative** in the sense that the static types overestimate the actual types occurring during query evaluation (the latter are also called **dynamic types**).

### Dynamic type

```
let $x := 42 return
if ($x gt 0) then <a/> else $x    →    <a/> : elem(a)
```

Dynamic types (here: `elem(a)`) are always **subtypes** of the static types inferred at compile time:

$$\text{elem}(a) <: \text{elem}(a) \mid \text{int}$$

[If  $t <: t'$ , then  $t'$  accepts all values accepted by  $t$  (and possibly more).]

# Static typing

## Statically typed expression [type annotations]

```

for $_v0 [node()] := fn:root(.) [node()]
return
  for $_v1 [element(a)] in $_v0/child::a [element(a)*]
  return
    for $_v2 [element(b)] in $_v1/child::b [element(b)*]
    return
      if (
        some $_v3 [xs:integer] in
          fn:data($_v2/attribute::c [attribute(c)]) [xs:integer]
          satisfies some $_v4 [xs:integer] in
            fn:data(42) [xs:integer] satisfies
              op:ge($_v3, $_v4)) [xs:boolean]
      then $_v2 [element(b)]
      else () [empty-sequence()]
[element(b)?]

```

# Static typing and XML-Schema


## Notes:

- The static type `xs:integer` for the subexpression

```
fn:data($_v2/attribute::c)
```

may only be derived if the **schema attribute declaration** `attr c {xs:integer}` is in scope. (Which type would be inferred otherwise?)

- Likewise, if **schema element declarations** `elem a { $\tau_1$ }` and `elem b { $\tau_2$ }` are in scope, we can type the two XPath location steps more rigidly (and gain).
- The other way round, in specific cases, static typing may make validation (at runtime) unnecessary.

(Research!) 



# Type-based simplifications

## Single-item iteration

If the type of  $e_1$  denotes a **single item**, then

$$\begin{aligned}\text{for } \$v \text{ in } e_1 \text{ return } e_2 &\equiv \text{let } \$v := e_1 \text{ return } e_2 \\ \text{some } \$v \text{ in } e_1 \text{ satisfies } e_2 &\equiv \text{let } \$v := e_1 \text{ return } e_2 \\ \text{every } \$v \text{ in } e_1 \text{ satisfies } e_2 &\equiv \text{let } \$v := e_1 \text{ return } e_2\end{aligned}$$

## ✎ “Empty” iteration

If the type of  $e_1$  is `empty-sequence()`, then

$$\begin{aligned}\text{for } \$v \text{ in } e_1 \text{ return } e_2 &\equiv \\ \text{some } \$v \text{ in } e_1 \text{ satisfies } e_2 &\equiv \\ \text{every } \$v \text{ in } e_1 \text{ satisfies } e_2 &\equiv\end{aligned}$$

## Type-based simplification

- Apply *single item iteration* simplification.
- Specialize `op:ge` (no overloading anymore).

### First simplification steps

```

let $_v0 [node()] := fn:root(.) [node()]
return
  for $_v1 [element(a)] in $_v0/child::a [element(a)*]
  return
    for $_v2 [element(b)] in $_v1/child::b [element(b)*]
    return
      if (
        let $_v3 [xs:integer] :=
          fn:data($_v2/attribute::c [attribute(c)]) [xs:integer]
        let $_v4 [xs:integer] := fn:data(42) [xs:integer]
        return op:integer-ge($_v3, $_v4)) [xs:boolean]
      then $_v2 [element(b)]
      else () [empty-sequence()]
[element(b)?]

```

# Type-based simplification

- `fn:data()` on atomic values is the identity.
- Unfold `let` bindings (but only if this is safe to do).



## More simplification steps

```
for $_v1 [element(a)] in fn:root()/child::a [element(a)*]
return
  for $_v2 [element(b)] in $_v1/child::b [element(b)*]
  return
    if (op:integer-ge(fn:data($_v2/attribute::c), 42))
    then $_v2 [element(b)]
    else () [empty-sequence()]
[element(b)?]
```

- For **holistic** XPath location step implementations, it might be more efficient to “stitch” the path steps together again.  
(For  $\sqsubset$ -based step evaluation, the above is just fine.)

## When types get in the way

Static typing may be used to improve XQuery expressions at **compile time**. Since data is not available at this point, typing is conservative. This can get in the way.

### Static typing gets in the way (ice-warning.xq)

```
for $w in $weather-reports/weather-report
return
  if (($w/temp [element(temp)*] * 0.9) < 2.5)
  then <ice-warning> { $w/@* } </ice-warning>
  else ()
```

```
$ XQuery ice-warning.xq
TYPE ERROR: no variant of function op:times accepts
the given argument type(s): double*; decimal
$
```

## When types get in the way

In principle, the XQuery compiler could derive the type annotation `element(temp)` for subexpression `$w/temp` from the type of `$w` and a corresponding XML Schema: perform “location step” on schema type.

### Possible fixes:

- 1 User shares her schema knowledge with the compiler:

```
$w/temp[1] [element(temp)] * 0.9
```

- 2 User asserts that path expression yields exactly one node. System checks at runtime:<sup>58</sup>

```
fn:exactly-one($w/temp) [element(temp)] * 0.9
```

---

<sup>58</sup> Also available: `fn:zero-or-one()`, `fn:one-or-more()`.

# XQuery compilation

Two **principal data structures** form the backbone of the XQuery data model:

## ① **Ordered, unranked trees of nodes**

We know how to map these into the relational domain. A node  $v$  in such a tree is representable by  $pre(v)$  (which may be used as a key in the  $pre|post$ ,  $pre|kind$ , ... tables to explore  $v$ 's containing tree.

## ② **Ordered, finite sequences of items** ( $i_1, i_2, \dots, i_n$ )

An item either is a **node** or an **atomic value** of an XML Schema simple type  $\tau_s$ . Note:  $\tau_s$  might not be available in the database back-end. Maintaining sequence order in a relational back-end calls for extra effort and care.

# Representing items and sequences

Let  $i$ ,  $i_k$  denote XQuery **items** (atomic values, nodes):

$(i_1, i_2, \dots, i_n)$

<i>pos</i>	<i>item</i>
1	$i_1$
2	$i_2$
$\vdots$	$\vdots$
$n$	$i_n$

$i$

<i>pos</i>	<i>item</i>
1	$i$

$()$

<i>pos</i>	<i>item</i>

- Item  $i$  and singleton sequence  $(i)$  share representation.
- Issues of polymorphism in column ***item*** not addressed here.

## (Explicit) Sequence positions

The maintenance of **explicit sequence positions** in column `pos` may seem costly—but it is mandatory to properly implement XQuery sequence order:

- In arbitrary XQuery expressions, sequence order does *not* coincide with document order.
- Sequences may contain non-node items (and nodes).
- For sequences of type `node()*` (nodes only) in document order, we may derive `pos` from `item` (see below).
- Once the query has been mapped to the system's **physical algebra**, intermediate results (tables) are ordered. This physical order may coincide with `pos`.



# Target code: A dialect of relational algebra

## Operators

$\sigma_a$	row selection
$\pi_{a,b:c}$	projection/renaming
$\varrho_{a:(b,\dots,c)}/d$	<b>row numbering</b>
$- \times -$	Cartesian product
$- \bowtie_p -$	join
$- \dot{\cup} -$	disjoint union
$- \setminus -$	difference
$\delta$	duplicate elimination
$\odot_{a:(b,\dots,c)}$	apply $\circ \in \{*, =, <, \dots\}$

- Column names denoted by  $a, b, c, \dots$
- Last row: the algebra contains operators  $\otimes, \oplus, \otimes, \oslash, \dots$

# Relational algebra dialect

- This dialect of relational algebra has been chosen to be efficiently implementable by standard database kernels.
- A small library of **simple support routines**, ideally implemented in or close to the database kernel, complete the target language.
  - ▶ Support routines provide shorthands for “*micro plans*” recurring in the algebraic plans emitted by the XQuery compiler.

## Support routines (excerpt)

⌋	<i>staircase join</i> (XPath evaluation)	ROOT	support for <code>fn:root</code>
$\varepsilon$	element node construction	DOC	support for <code>fn:doc</code>
$\tau$	text node construction	SUM	support for <code>fn:sum</code>

# Relational algebra dialect

## Row selection

**Row selection**  $\sigma$  does *not* support predicates as arguments. Instead,  $\sigma_a$  (Boolean column  $a$ ), selects all rows with column  $a = \text{true}$ :

$$\sigma_a \left( \begin{array}{|c|c|} \hline a & b \\ \hline \text{true} & 3 \\ \text{false} & 2 \\ \text{true} & 1 \\ \text{true} & 5 \\ \text{false} & 4 \\ \text{true} & 7 \\ \hline \end{array} \right) = \begin{array}{|c|c|} \hline a & b \\ \hline \text{true} & 3 \\ \text{true} & 1 \\ \text{true} & 5 \\ \text{true} & 7 \\ \hline \end{array}$$

Predicate evaluation is lifted onto the level relational algebra itself, using the  $\odot$  operators.

# Relational algebra dialect

## Applying operator $\circ$ via $\odot$

$$\oplus_{c:(a,b)} \left( \begin{array}{|c|c|} \hline a & b \\ \hline 0 & 3 \\ 40 & 2 \\ 41 & 1 \\ 5 & 5 \\ -4 & 4 \\ 35 & 7 \\ \hline \end{array} \right) = \begin{array}{|c|c|c|} \hline a & b & c \\ \hline 0 & 3 & 3 \\ 40 & 2 & 42 \\ 41 & 1 & 42 \\ 5 & 5 & 10 \\ -4 & 4 & 0 \\ 35 & 7 & 42 \\ \hline \end{array}$$

$$\otimes_{c:(a,b)} \left( \begin{array}{|c|c|} \hline a & b \\ \hline 0 & 3 \\ 40 & 2 \\ 41 & 1 \\ 5 & 5 \\ -4 & 4 \\ 35 & 7 \\ \hline \end{array} \right) = \begin{array}{|c|c|c|} \hline a & b & c \\ \hline 0 & 3 & \text{true} \\ 40 & 2 & \text{false} \\ 41 & 1 & \text{false} \\ 5 & 5 & \text{false} \\ -4 & 4 & \text{true} \\ 35 & 7 & \text{false} \\ \hline \end{array}$$

# Relational algebra dialect

## Predicates on the algebraic level

Formulate the selection  $\sigma_{a>b \wedge c=42}(e)$  on the algebraic level using operators  $\oplus, \otimes, \dots$  ( $e$  denotes a relation containing columns  $a, b, c$ ). You will also need  $\times$ .

(You will understand why this is considered a simple, “*assembly-style*” relational algebra.)

# Relational algebra dialect

## Column projection and renaming

**Column projection**  $\pi$  is *not* required to remove duplicate rows after column removal. **Explicit duplicate removal** is performed by  $\delta$ .

(Note: also renames column  $a$  into  $c$ .)

$$\pi_{c:a} \left( \begin{array}{|c|c|} \hline a & b \\ \hline 0 & 10 \\ 1 & 10 \\ 2 & 10 \\ 2 & 20 \\ 2 & 30 \\ 3 & 10 \\ \hline \end{array} \right) = \begin{array}{|c|} \hline c \\ \hline 0 \\ 1 \\ 2 \\ 2 \\ 2 \\ 3 \\ \hline \end{array} ; \quad \delta \left( \begin{array}{|c|} \hline c \\ \hline 0 \\ 1 \\ 2 \\ 2 \\ 2 \\ 3 \\ \hline \end{array} \right) = \begin{array}{|c|} \hline c \\ \hline 0 \\ 1 \\ 2 \\ 3 \\ \hline \end{array}$$

- **Note:** In the plans emitted by the XQuery compiler,  $\delta$  is seldomly necessary as the projection list includes key columns.

# Relational algebra dialect

Order is prevalent in XQuery but **row order** has *no* meaning in the relational model.

- Reflect order on the level of the relational model by means of explicit **pos** columns.
- Derive these columns via the **row numbering**  $\varrho$  operator.

## Row numbering operator $\varrho$

$$\varrho_{a:(b,c)}(e)$$

Use order criteria (columns)  $b, c$  to order the rows of  $e$ , attach new **densely numbered**  $(1, 2, \dots)$  column  $a$  reflecting this order.

$$\varrho_{a:(b,c)/d}(e)$$

As before, but perform the numbering for each **group of rows** with identical  $d$  values (numbering in each group starts from 1).

# Relational algebra dialect

## Row numbering via $\varrho$

Suppose the evaluation of an XPath location step yields a one-column ( $pre$ ) relation (node identifiers). Use  $\varrho$  to derive **sequence order from document order**:

$$\varrho_{pos:(pre)} \left( \begin{array}{c|c} & pre \\ \hline & 100 \\ & 12 \\ & 13 \\ & 6 \\ & 2 \\ & 212 \end{array} \right) = \begin{array}{c|c} pos & pre \\ \hline 5 & 100 \\ 3 & 12 \\ 4 & 13 \\ 2 & 6 \\ 1 & 2 \\ 6 & 212 \end{array}$$



# Relational algebra dialect

## Grouped row numbering

$$\varrho_{c:(a)/b} \left( \begin{array}{|c|c|} \hline a & b \\ \hline 3 & 1 \\ 4 & 2 \\ 1 & 2 \\ 8 & 1 \\ 8 & 2 \\ \hline \end{array} \right) = \begin{array}{|c|c|c|} \hline c & a & b \\ \hline 1 & 3 & 1 \\ 2 & 8 & 1 \\ \hline 2 & 4 & 2 \\ 1 & 1 & 2 \\ 3 & 8 & 2 \\ \hline \end{array}$$

**Note:** if  $b$  is key or constant,  $\varrho_{c:(a)/b}$  may be simplified.

$\varrho_{c:(a)/b}(e)$  is expressible using the SQL/OLAP amendment to SQL:1999

## Grouped row numbering in SQL

```
SELECT a,b,DENSE_RANK() OVER
      (PARTITION BY b ORDER BY a) AS c
FROM e
```

# Relational algebra dialect

- As we will see,  $\varrho$  operators are pervasive in the query plans emitted by the XQuery compiler.
- Since all conceivable implementations of  $\varrho$  rely on a **blocking sort**, the compiler will try to remove/simplify occurrences of  $\varrho$ .
- In particular cases, however, *physical row order* and the order criteria of  $\varrho$  coincide. This renders  $\varrho$  almost a no-op.

## Physical row order and $\varrho$

Suppose the database delivers the rows of  $e$  in  $(b, c)$  order. Which of the following  $\varrho$  instances require a blocking sort?

①  $\varrho_{a:(b,c)}(e)$

②  $\varrho_{a:(b)}(e)$

③  $\varrho_{a:(b)/c}(e)$

④  $\varrho_{a:(c)/b}(e)$

## Relational algebra dialect

There are further properties of the emitted plans, which facilitate their efficient evaluation by the database kernel:

- All joins are **equi-joins** only ( $\rightarrow$  use *merge join* or *hash join* internally):

$$e_1 \bowtie_{a_1=a_2} e_2$$

- All union operations consume **disjoint** operands only ( $\rightarrow$  simply concatenate rows internally):

$$e_1 \dot{\cup} e_2$$

- All difference operators process **keys** only ( $\rightarrow$  index-only operation):

$$\pi_k(e_1) \setminus \pi_k(e_2)$$

# Support routines

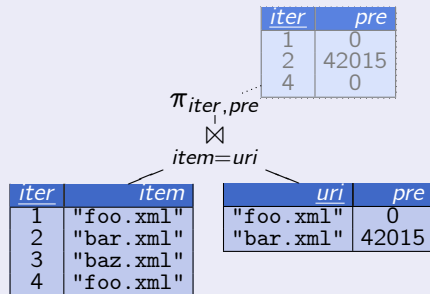
- Emitted plans may refer to a small number of **support routines** which primarily encapsulate access to
  - ① tables maintained by the relational **XML fragment encoding** (*i.e.*, the tables needed to implement  $\mathcal{E}$ ), and to
  - ② tables needed to maintain **persistent XML documents** in the database.
- The support routines consume and return tables just like relational operators.
  - ▶ For efficiency reasons, these routines are **implemented next to/inside the database kernel**.
  - ▶ Their semantics, however, is equivalent to specific algebraic expressions (**relational micro plans**).

# Support routines

## Access to persistent XML documents

The database maintains a table *uri|pre* mapping XML document URIs to preorder ranks of document nodes.

Routine D0C accepts a whole table of URIs and performs the mapping for each of these.



Routine D0C encapsulates the join and the access to the *uri|pre* table. (Table *iter|item* is the only argument to D0C.)

# Compiling FLWORS

XQuery Core is designed around an **iteration primitive**, the **for-return** construct.

- A for loop iterates the evaluation of loop body  $e$  for successive **bindings** of the loop variable  $\$v$ :

$$\begin{aligned} &\text{for } \$v \text{ in } (i_1, i_2, \dots, i_n) \text{ return } e \\ &\quad \equiv \\ &(e[i_1/\$v], e[i_2/\$v], \dots, e[i_n/\$v]) \end{aligned}$$

where  $e[i/\$v]$  denotes the consistent replacement of all free occurrences of  $\$v$  in  $e$  by item  $i$ .

- In principle, in XQuery it is semantically sound to evaluate all iterations of  $e$  in **parallel** or in **arbitrary order** (as long as the final result sequence is correctly ordered).

# Example: Compiling FLWORS

## Parallel/arbitrary evaluation of for loop body

```
for $x in (1,2,3) return $x*10 gt 15
```

≡

```
((($x*10 gt 15)[1/$x],  
  ($x*10 gt 15)[2/$x],  
  ($x*10 gt 15)[3/$x])
```

≡

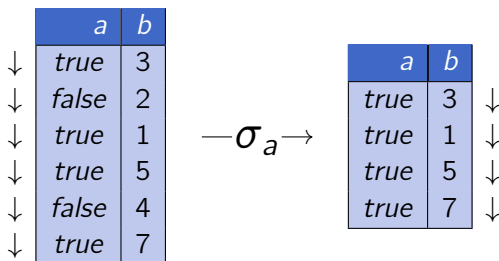
```
(1*10 gt 15,  
 2*10 gt 15,  
 3*10 gt 15)
```

↓

```
(false,true,true)
```

# The relational query processing mode

This iterative nature of evaluation does *not* fit too well with the relational query processing mode.



$\Rightarrow$  Consume **bulk of tuples**, produce **bulk of tuples** (code locality).



# Variable representation

**Relational representation of XQuery variables:** collect the *bindings* of all iterations into a *single relation*.

$$\begin{aligned} &\text{for } \$v \text{ in } (i_1, i_2, \dots, i_n) \text{ return } e \\ &\quad \equiv \\ &\quad (e[i_1/\$v], e[i_2/\$v], \dots, e[i_n/\$v]) \end{aligned}$$

Representation of  $(i_1, i_2, \dots, i_n)$ :

<i>pos</i>	<i>item</i>
1	$i_1$
2	$i_2$
$\vdots$	$\vdots$
$n$	$i_n$

Derive  $\$v$  as follows:

<i>iter</i>	<i>pos</i>	<i>item</i>
1	1	$i_1$
2	1	$i_2$
$\vdots$	$\vdots$	$\vdots$
$n$	1	$i_n$

## Iterated item sequences: *iter|pos|item* tables

Such 

<i>iter</i>	<i>pos</i>	<i>item</i>
-------------	------------	-------------

 tables will be pervasive in this XQuery compilation scheme: the relational plan for any compiled XQuery subexpression will yield a relation of this form.

The *iter|pos|item* representation of item sequences

<i>iter</i>	<i>pos</i>	<i>item</i>
⋮	⋮	⋮
<i>i</i>	<i>p</i>	<i>x</i>
⋮	⋮	⋮

“In the *i*th iteration, the item at position *p* has value *x*.”

# The *iter|pos|item* representation

## The *iter|pos|item* representation

What is the *iter|pos|item* representation of the **result** of the for loop below?

```
for $x in (1,2,3,4)
return if ($x mod 2 eq 0) then -10 else (10,$x)
```

Result:

<i>iter</i>	<i>pos</i>	<i>item</i>

# Deriving variable representations

## Deriving Variables

Suppose we wrap the former query in another for loop:

```
for $y in (for $x in (1,2,3,4) return
           if ($x mod 2 eq 0) then -10 else (10,$x))
return $y * 5
```

Devise an algebraic query that derives the representation of variable \$y from its bindings:<sup>59</sup>

<i>iter</i>	<i>pos</i>	<i>item</i>
1	1	10
1	2	1
2	1	-10
3	1	10
3	2	3
4	1	-10

$$\begin{array}{|c|} \hline \textit{pos} \\ \hline 1 \\ \hline \end{array} \times \pi_{\textit{iter}:\textit{inner}, \textit{item}}(\varrho_{\textit{inner}:(\textit{iter}, \textit{pos})}(\cdot))$$

<i>iter</i>	<i>pos</i>	<i>item</i>

<sup>59</sup>Remember: an XQuery variable is always bound to a *single* item.

# Iteration scopes

The principal idea of the compilation scheme is to compile any subexpression in dependence of the **iteration scope**  $s_i$  it appears in.

- The outermost “iteration scope” is  $s_0$ .  
Note: in  $s_0$ , *no* actual iteration is performed (any top-level expression is evaluated exactly once).
- A new **iteration scope** is opened for every for–return construct:

Outermost scope  $s_0$  and iteration scope  $s_1$

```
 $s_0$  [ for $x in (k, ..., 5, ..., 2, 1)  
     $s_1$  [ return $x * 5
```

# Iteration scopes

## Flat iteration

$$s_0 \left[ \begin{array}{l} \text{for } \$x \text{ in } (k, \dots, 5, \dots, 2, 1) \\ s_1 \left[ \text{return } \$x * 5 \end{array} \right.$$

Encoding of subexpressions in their respective scopes:

In  $s_0$ :

$(k, \dots, 2, 1)$

<i>pos</i>	<i>item</i>
1	$k$
$\vdots$	$\vdots$
$k-1$	2
$k$	1

In  $s_1$ :

$\$x$

<i>iter</i>	<i>pos</i>	<i>item</i>
1	1	$k$
$\vdots$	$\vdots$	$\vdots$
$k-1$	1	2
$k$	1	1

In  $s_1$ :

5

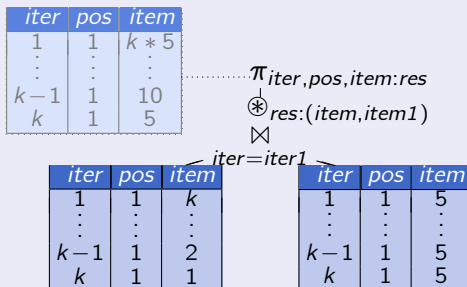
<i>iter</i>	<i>pos</i>	<i>item</i>
1	1	5
$\vdots$	$\vdots$	$\vdots$
$k-1$	1	5
$k$	1	5

# Relational algebra evaluates FLWOR block

Input: XQuery

```
for $x in (k,...,5,...,2,1)
  return $x * 5
```

Output: Relational Algebra



# Loop lifting

Subexpressions are compiled in dependence of the **iteration scope**  $s_i$ —represented as unary relation  $loop(s_i)$ —in which they occur.

## XQuery Iteration

$s_0$   $\left[ \begin{array}{l} \text{for } \$v \text{ in } (i_1, i_2, \dots, i_n) \\ \quad s_1 \left[ \text{return } e \end{array} \right.$

$loop(s_0)$

<i>iter</i>
1

$loop(s_1)$

<i>iter</i>
1
$\vdots$
$n$

- Item "a" in scope  $s_1$ :

<i>iter</i>	<i>pos</i>	<i>item</i>
1	1	"a"
$\vdots$	$\vdots$	$\vdots$
$n$	1	"a"

- Sequence ("a", "b") in  $s_1$ :

<i>iter</i>	<i>pos</i>	<i>item</i>
1	1	"a"
1	2	"b"
$\vdots$	$\vdots$	$\vdots$
$n$	1	"a"
$n$	2	"b"



## Loop lifting

Much like the static typing process, we may formally specify a bottom-up compilation procedure in terms of **inference rules**. The rules collectively define the “*compiles to*” function  $\Rightarrow$ .

- The inference rules rely on
  - ① an **variable environment**  $\Gamma$  mapping variable names to algebraic plans, and
  - ② relation *loop* encoding the **current iteration scope**.

### Compilation rule for constant item *i*

$$\frac{}{\Gamma; loop \vdash i \Rightarrow loop \times \begin{array}{|c|c|} \hline pos & item \\ \hline 1 & i \\ \hline \end{array}}$$

## More compilation rules

### Compile arithmetics (here: `op:plus`)

$$\frac{\Gamma; loop \vdash e_1 \Rightarrow q_1 \quad \Gamma; loop \vdash e_2 \Rightarrow q_2}{\Gamma; loop \vdash \text{op:plus}(e_1, e_2) \Rightarrow \pi_{iter, pos, item:res}(\oplus_{res:(item, item')}(q_1 \bowtie_{iter=iter'} \pi_{iter':iter, item':item}(q_2)))}$$

### Compile `let` binding

$$\frac{\Gamma; loop \vdash e_1 \Rightarrow q_1 \quad \Gamma + \{v \mapsto q_1\}; loop \vdash e_2 \Rightarrow q_2}{\Gamma; loop \vdash \text{let } \$v := e_1 \text{ return } e_2 \Rightarrow q_2}$$

### Compile variable reference

$$\frac{}{\{\dots, v \mapsto q, \dots\}; loop \vdash \$v \Rightarrow q}$$

# Nested iteration scopes

## Nested for iterations

$s_0$   $\left[ \begin{array}{l} \text{for } \$v_0 \text{ in } (10, 20) \\ \quad s_1 \left[ \begin{array}{l} \text{for } \$v_1 \text{ in } (100, 200) \\ \quad s_2 \left[ \text{return } \$v_0 + \$v_1 \end{array} \right. \end{array} \right. \end{array} \right.$

$loop(s_0)$	$loop(s_1)$	$loop(s_2)$
iter	iter	iter
1	1	1
	2	2
		3
		4

Derive  $\$v_0, \$v_1$  as before (uses row numbering operator  $\rho$ ):

$\$v_0$  in  $s_1$ :

iter	pos	item
1	1	10
2	1	20

$\$v_1$  in  $s_2$ :

iter	pos	item
1	1	100
2	1	200
3	1	100
4	1	200

Variable  $\$v_0$  in scope  $s_2$ ?



# Nested iteration scopes

## Nested for iterations

$$s_0 \left[ \begin{array}{l} \text{for } \$v_0 \text{ in } (10, 20) \\ s_1 \left[ \begin{array}{l} \text{for } \$v_1 \text{ in } (100, 200) \\ s_2 \left[ \text{return } \$v_0 + \$v_1 \end{array} \right. \end{array} \right. \end{array}$$

Capture the semantics of **nested iteration** in an additional relation *map*:

<i>map</i>	
<i>inner</i>	<i>outer</i>
1	1
2	1
3	2
4	2

Read tuple  $\langle i, o \rangle$  as:

"If the outer for loop is in its *o*th iteration, the inner for loop is iterated the *i*th time."

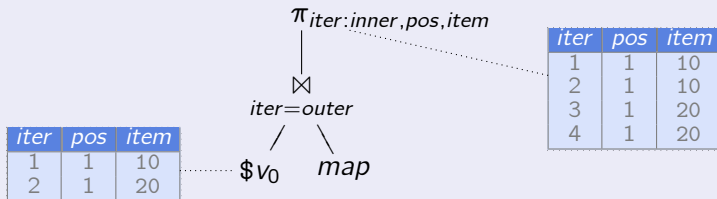
# Nested iteration scopes

## Nested for iterations

$$s_0 \left[ \begin{array}{l} \text{for } \$v_0 \text{ in } (10,20) \\ s_1 \left[ \begin{array}{l} \text{for } \$v_1 \text{ in } (100,200) \\ s_2 \left[ \text{return } \$v_0 + \$v_1 \end{array} \right. \end{array} \right.$$

map	
inner	outer
1	1
2	1
3	2
4	2

## Representation of $\$v_0$ in $s_2$



# FLWOR evaluation in scope $s_2$

## Nested for iterations

```

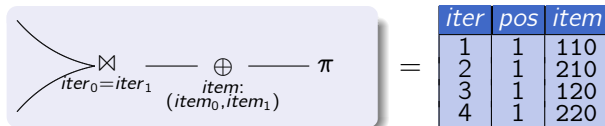
for $v_0 in (10,20)
  for $v_1 in (100,200)
     $s_2$  [ return $v_0 + $v_1
  
```

$\$v_0$

$iter_0$	$pos_0$	$item_0$
1	1	10
2	1	10
3	1	20
4	1	20

$\$v_1$

$iter_1$	$pos_1$	$item_1$
1	1	100
2	1	200
3	1	100
4	1	200



## Back-mapping to enclosing scopes

On the previous slide, note that the result of the iteration is **represented with respect to the innermost scope**  $s_2$ :

<i>iter</i>	<i>pos</i>	<i>item</i>
1	1	110
2	1	210
3	1	120
4	1	220

We can re-use the *map* relation to map this result back into  $s_1$  and finally back into  $s_0$ .

### Representation of this result in scopes $s_1$ and $s_0$ ?

In  $s_1$ :

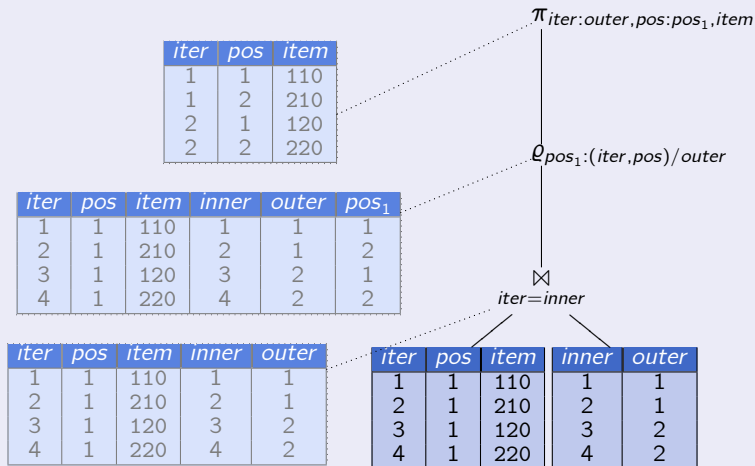
<i>iter</i>	<i>pos</i>	<i>item</i>
1	1	110
1	2	210
2	1	120
2	2	220

In  $s_0$ :

<i>iter</i>	<i>pos</i>	<i>item</i>
1	1	110
1	2	210
1	3	120
1	4	220

# Back-mapping to enclosing scopes

## Back-mapping from scope $s_2$ to $s_1$





# Back-mapping to enclosing scopes

A further, identical, back-mapping step on this result yields the final result in the outermost scope  $s_0$ .

Of course, this second back-mapping step needs to use the *map* relation between scopes  $s_0$  and  $s_1$ .



 Relation *map* between scopes  $s_0$  and  $s_1$ ?

<i>inner</i>	<i>outer</i>
1	1
2	1

# Compiling for $\$v$ in $e_1$ return $e_2$

## XQuery Iteration

$$s_i \left[ \begin{array}{l} \text{for } \$v \text{ in } e_1 \\ \quad s_{i+1} \left[ \text{return } e_2 \end{array} \right.$$

### Summary of for-return compilation scheme:

- ➊ Compute relation *map* between current scope  $s_i$  and new iteration scope  $s_{i+1}$ .
  - (a) Derive representation of  $\$v$  from result of  $e_1$ .
  - (b) Derive new *loop* relation from representation of  $\$v$ .
- ➋ Compile  $e_2$  in a variable environment  $\Gamma$  where all variables have been mapped into scope  $s_{i+1}$  and  $\$v$  is visible.
- ➌ Perform back-mapping of result of  $e_2$  into scope  $s_i$ .

# Compiling for \$v\$ in \$e\_1\$ return \$e\_2\$

## Compiling for-return

$$\begin{array}{c}
 \Gamma; loop \vdash e_1 \Rightarrow q_1 \quad q_v \stackrel{\textcircled{1} (a)}{=} \frac{\text{pos}}{1} \times \pi_{iter:inner,item}(\varrho_{inner:(iter,pos)}(q_1)) \\
 loop_v \stackrel{\textcircled{1} (b)}{=} \pi_{iter}(q_v) \quad map \stackrel{\textcircled{1}}{=} \pi_{outer:iter,inner}(\varrho_{inner:(iter,pos)}(q_1)) \\
 \Gamma_v \equiv \Gamma \left[ x \mapsto \pi_{iter:inner,pos,item}(q_x \mathbin{\boxtimes}_{iter=outer} map) / x \mapsto q_x \right] + \{v \mapsto q_v\} \\
 \Gamma_v; loop_v \vdash e_2 \Rightarrow q_2 \quad \textcircled{2} \\
 \hline
 \Gamma; loop \vdash \text{for } \$v \text{ in } e_1 \text{ return } e_2 \Rightarrow \textcircled{3} \\
 \pi_{iter:outer,pos:pos_1,item}(\varrho_{pos_1:(iter,pos)/outer}(q_2 \mathbin{\boxtimes}_{iter=inner} map))
 \end{array}$$

**Note:** numbers in  $\bigcirc$  refer to previous slide.

## Compiling for $\$v$ in $e_1$ return $e_2$

Note that the `for-return` compilation rule indicates that the resulting algebra tree will contain numerous **identical subtrees**.

- Such opportunities for **sharing common algebraic subexpressions** may be discovered *after* compilation: *common subexpression elimination* (CSE).
- Alternatively, the compiler may already make sharing explicit and emit a **directed acyclic graph (DAG) of algebraic operators** instead of an algebra tree.

We follow the latter approach.

# Compiling nested FLWOR blocks

## XQuery FLWOR Block

$$s_0 \left[ \begin{array}{l} \text{for } \$x \text{ in } (100, 200, 300) \text{ return} \\ s_1 \left[ \begin{array}{l} \text{for } \$y \text{ in } (30, 20) \text{ return} \\ s_2 \left[ \text{if } (\$x \text{ eq } \$y * 10) \text{ then } \$x \text{ else } () \end{array} \right] \end{array} \right]$$

Encoding of **invariable** sub-expressions is **denormalized** in inner scopes (*i.e.*, item sequence value independent of iteration):

10 in  $s_2$ :

iter	pos	item
1	1	10
2	1	10
3	1	10
4	1	10
5	1	10
6	1	10

(30, 20) in  $s_1$ :

iter	pos	item
1	1	30
1	2	20
2	1	30
2	2	20
3	1	30
3	2	20

[illegible]

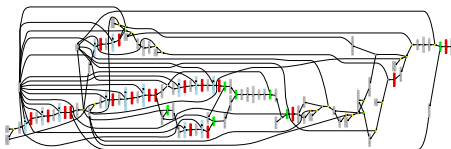
# Compiling complex queries

## XMark query Q8

```
for $p in fn:doc("auction.xml")/site/people/person
return
  let $a := for $t in fn:doc("auction.xml")/site/
              closed_auctions/closed_auction
              return if (fn:data($t/buyer/person/text()) =
                        fn:data($p/id/text()))
                      then $t
                      else ()
  return <item>{ <person>{ $p/name/text() }</person>,
                text { fn:count($a) }
              }
</item>
```

⇒ Compiled into a DAG of 120 operators, significant sharing.

- Equivalent tree has  $\approx 2,000$  operator nodes.





## Compiling conditional expressions

The compilation of **conditional expressions** `if ( $e_1$ ) then  $e_2$  else  $e_3$`  fits nicely into the compilation framework.

### Iterated evaluation of `if ( $e_1$ ) then $e_2$ else $e_3$`

```
for $x in 1 to 4
  return if ($x mod 2 = 0)
    then "even"
    else "odd"
```

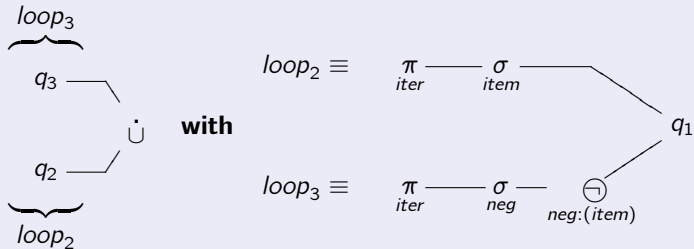
- Here, the `if-then-else` is evaluated in four iterations  $iter \in \{1, 2, 3, 4\}$ .
  - The `then` branch is evaluated in iterations  $\{2, 4\}$ .
- ⇒ Consequently, the `else` branch is evaluated in iterations  $\{1, 2, 3, 4\} \setminus \{2, 4\} = \{1, 3\}$ .

# Compiling conditional expressions

## XQuery conditional expression

if ( $e_1$ ) then	$e_2$	else	$e_3$
$\Downarrow$	$\Downarrow$		$\Downarrow$
$q_1$	$q_2$		$q_3$

## Equivalent algebraic code



$\ominus$  denotes the algebra's Boolean negation operator.

# Compiling conditional expressions

## Compiling if-then-else

$$\begin{array}{c}
 \Gamma; loop \vdash q_1 \Rightarrow e_1 \\
 loop_2 \equiv \pi_{iter}(\sigma_{item}(q_1)) \quad loop_3 \equiv \pi_{iter}(\sigma_{neg}(\ominus_{neg:(item)}(q_1))) \\
 \Gamma; loop_2 \vdash e_2 \Rightarrow q_2 \quad \Gamma; loop_3 \vdash e_3 \Rightarrow q_3 \\
 \hline
 \Gamma; loop \vdash \text{if } (e_1) \text{ then } e_2 \text{ else } e_3 \Rightarrow q_2 \dot{\cup} q_3
 \end{array}$$

### Note:

- Note that the **then** and **else** branches are compiled with different *loop* relations.
- $q_{2,3}$  do not contribute to the overall result for those iterations missing in  $loop_{2,3}$  respectively.
- Operator  $\dot{\cup}$  is guaranteed to union disjoint inputs.

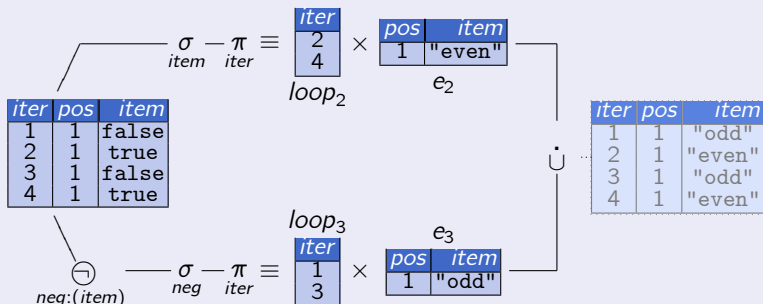
# Evaluation of conditional expressions

## XQuery conditional expression

```

for $x in 1 to 4
s1 [ return if (  $\underbrace{\$x \bmod 2 = 0}_{e_1}$  ) then  $\underbrace{"even"}_{e_2}$  else  $\underbrace{"odd"}_{e_3}$  ]
  
```

## Evaluation

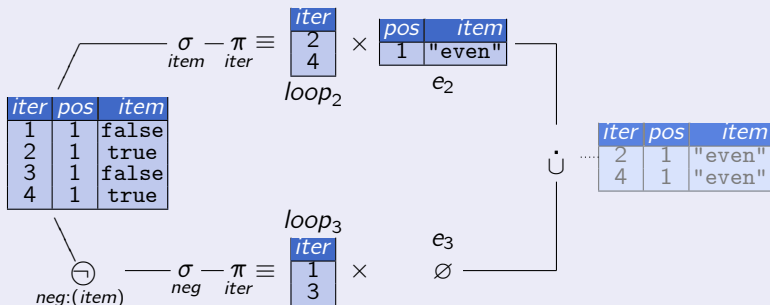


# Compiling the FLWOR where clause

## Normalized XQuery FLWOR block

$$\left[ \begin{array}{l} \text{for } \$x \text{ in } 1 \text{ to } 4 \\ \text{where } \$x \bmod 2 = 0 \\ \text{return "even"} \end{array} \right] \equiv \left[ \begin{array}{l} \text{for } \$x \text{ in } 1 \text{ to } 4 \\ \text{return if } \$x \bmod 2 = 0 \\ \quad \text{then "even" else } () \end{array} \right]$$

## Evaluation



## “Missing” iterations and ()

- Note how the intermediate result on the previous slide encodes the **empty sequence** in terms of **missing iter values**:

<i>iter</i>	<i>pos</i>	<i>item</i>
2	1	"even"
4	1	"even"

$\Rightarrow$  Evaluation in the first and third iterations yielded ().  $e, () = (), e = e$ .

- Clearly, encoding () by *absence* (of *iter* values) requires additional information about *all* iterations which have been evaluated. This is exactly what relation *loop* provides:

$$\begin{array}{c} \textit{loop} \\ \begin{array}{|c|} \hline \textit{iter} \\ \hline 1 \\ 2 \\ 3 \\ 4 \\ \hline \end{array} \end{array} \setminus \pi_{\textit{iter}} \left( \begin{array}{|c|c|c|} \hline \textit{iter} & \textit{pos} & \textit{item} \\ \hline 2 & 1 & \text{"even"} \\ 4 & 1 & \text{"even"} \\ \hline \end{array} \right) = \begin{array}{|c|} \hline \textit{iter} \\ \hline 1 \\ 3 \\ \hline \end{array}$$

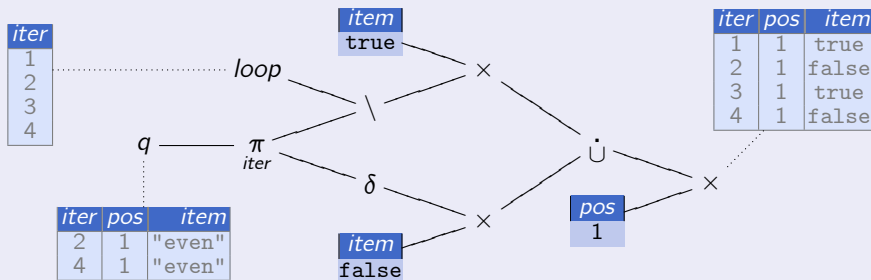


# “Missing” iterations and ()

Insert call to `fn:empty()` in loop body

```
for $x in (1 to 4)
  return fn:empty(if ($x mod 2 = 0) then "even"
                  else ())
```

Evaluation of `fn:empty(e)` with  $e \Rightarrow q$





# Table of Contents I

<b>1.1</b>	<b>Welcome</b> .....	<b>4</b>
<b>1.2</b>	<b>Overview</b> .....	<b>5</b>
	XML .....	5
	XML and Databases .....	7
<b>1.3</b>	<b>Organization</b> .....	<b>12</b>
<b>2.4</b>	<b>Markup Languages</b> .....	<b>17</b>
	Early Markup .....	17
	An Application of Markup: A Comic Strip Finder .....	22
<b>3.5</b>	<b>Formalization of XML</b> .....	<b>32</b>
	Elements .....	33
	Attributes .....	36
	Entities .....	37
<b>3.6</b>	<b>Well-Formedness</b> .....	<b>38</b>
	Context-free Properties .....	39
	Context-dependent Properties .....	46
<b>3.7</b>	<b>XML Text Declarations</b> .....	<b>48</b>
	XML Documents and Character Encoding .....	49

# Table of Contents II

	Unicode .....	50
	XML and Unicode .....	56
<b>3.8</b>	<b>The XML Processing Model .....</b>	<b>57</b>
	The XML Information Set .....	59
	More XML Node Types .....	65
<b>4.9</b>	<b>DOM Level 1 (Core) .....</b>	<b>71</b>
<b>4.10</b>	<b>DOM Example Code .....</b>	<b>74</b>
<b>4.11</b>	<b>DOM—A Memory Bottleneck .....</b>	<b>78</b>
<b>5.12</b>	<b>SAX Events .....</b>	<b>85</b>
<b>5.13</b>	<b>SAX Callbacks .....</b>	<b>87</b>
<b>5.14</b>	<b>SAX and the XML Tree Structure .....</b>	<b>89</b>
<b>5.15</b>	<b>SAX and Path Queries .....</b>	<b>95</b>
	Path Query Evaluation .....	96
<b>5.16</b>	<b>Final Remarks on SAX .....</b>	<b>100</b>
<b>6.17</b>	<b>Valid XML .....</b>	<b>103</b>
<b>6.18</b>	<b>DTDs—Document Type Definitions .....</b>	<b>106</b>
	Element Declaration .....	107
	Attribute Declaration .....	113
	Crossreferencing via ID and IDREF .....	116

# Table of Contents III

	Other DTD Features .....	120
	A “Real Life” DTD—GraphML .....	121
	Concluding remarks on DTDs .....	126
<b>6.19</b>	<b>XML Schema .....</b>	<b>128</b>
	Some XML Schema Constructs .....	129
	Other XML Schema Concepts .....	133
<b>6.20</b>	<b>Validating XML Documents Against DTDs .....</b>	<b>134</b>
	Regular Expressions .....	136
	Evaluating Regular Expressions (Matching) .....	139
	Plugging It All Together .....	152
<b>7.21</b>	<b>Querying XML Documents .....</b>	<b>155</b>
	Overview .....	155
<b>7.22</b>	<b>The XQuery Data Model .....</b>	<b>158</b>
	The XQuery Type System .....	158
	Node Properties .....	163
	Items and Sequences .....	167
	Atomic Types .....	171
	Automatic Type Assignment (Atomization) .....	173
	Node Types .....	174

# Table of Contents IV

	Node Identity .....	176
	Document Order .....	177
<b>8.23</b>	<b>XPath—Navigational access to XML documents .....</b>	<b>181</b>
	Context .....	181
	Location steps .....	184
	Navigation axes .....	185
	Examples .....	190
<b>8.24</b>	<b>XPath Semantics .....</b>	<b>193</b>
	Document order & duplicates .....	193
	Predicates .....	200
	Atomization .....	204
	Positional access .....	208
<b>9.25</b>	<b>XSLT—An XML Presentation Processor .....</b>	<b>223</b>
	Separating content from style .....	224
	XSL Stylesheets .....	227
	XSLT Templates .....	229
	Examples .....	235
	Conflict Resolution and Modes in XSLT .....	242
	More on XSLT .....	247

# Table of Contents V

<b>10.26</b>	<b>XQuery—Declarative querying over XML documents</b>	<b>250</b>
	Introduction	250
	Preliminaries	253
<b>10.27</b>	<b>Iteration (FLWORS)</b>	<b>259</b>
	For loop	259
	Examples	261
	Variable bindings	264
	where clause	266
	FLWOR Semantics	271
	Variable bindings	278
	Constructing XML Fragments	279
	User-Defined Functions	297
<b>11.28</b>	<b>Mapping Relational Databases to XML</b>	<b>309</b>
	Introduction	309
	Wrapping Tables into XML	311
	Beyond Flat Relations	315
	Generating XML from within SQL	318
<b>11.29</b>	<b>Some XML Benchmarking Data Sets</b>	<b>320</b>
<b>12.30</b>	<b>Mapping XML to Databases</b>	<b>326</b>

# Table of Contents VI

	Introduction .....	326
<b>12.31</b>	<b>Relational Tree Encoding .....</b>	<b>334</b>
	Dead Ends .....	334
	Node-Based Encoding .....	340
	Working With Node-Based Encodings .....	345
<b>12.32</b>	<b>XPath Accelerator Encoding .....</b>	<b>347</b>
	Tree Partitions and XPath Axes .....	347
	Pre-Order and Post-Order Traversal Ranks .....	350
	Relational Evaluation of XPath Location Steps .....	354
<b>12.33</b>	<b>Path-Based Encodings .....</b>	<b>362</b>
	Motivation .....	362
	Data Guides .....	363
	Skeleton Extraction and Compression .....	364
	Data Vectors .....	371
	Skeleton Compression and Semi-Structured Data .....	373
	Improving Skeleton Compression .....	381
<b>13.34</b>	<b>Index Support .....</b>	<b>388</b>
	Overview .....	388
	Hierarchical Node IDs and B <sup>+</sup> Trees .....	389

# Table of Contents VII

	<i>Pre/Post</i> Encoding and B <sup>+</sup> Trees .....	390
	<i>Pre/Post</i> Encoding and R Trees .....	393
	More on Physical Design Issues .....	395
<b>14.35</b>	<b>Scan Ranges</b> .....	<b>405</b>
	descendant Axis .....	405
<b>14.36</b>	<b>Stretched <i>Pre/Post</i> Plane</b> .....	<b>409</b>
<b>14.37</b>	<b>XPath Symmetries</b> .....	<b>420</b>
<b>15.38</b>	<b>Updating XML Trees</b> .....	<b>427</b>
	Update Specification .....	427
	XUpdate .....	429
<b>15.39</b>	<b>Impact on XPath Accelerator Encoding</b> .....	<b>431</b>
<b>15.40</b>	<b>Impacts on Other Encoding Schemes</b> .....	<b>435</b>
<b>16.41</b>	<b>Serialization</b> .....	<b>446</b>
	Problem .....	446
	Serialization & <i>Pre/Post</i> Encoding .....	447
<b>16.42</b>	<b>Shredding (<math>\mathcal{E}</math>)</b> .....	<b>460</b>
<b>16.43</b>	<b>Completing the <i>Pre/Post</i> Encoding Table Layout</b> .....	<b>472</b>
<b>17.44</b>	<b>XPath Accelerator—Tree aware relational XML representation</b> .....	<b>481</b>

# Table of Contents VIII

	Enhancing Tree Awareness .....	481
<b>17.45</b>	<b>Staircase Join .....</b>	<b>490</b>
	Tree Awareness .....	490
	Context Sequence Pruning .....	493
	Staircases .....	502
<b>17.46</b>	<b>Injecting ↗ into PostgreSQL .....</b>	<b>515</b>
<b>17.47</b>	<b>Outlook: More on Performance Tuning in MonetDB/XQuery .....</b>	<b>523</b>
<b>18.48</b>	<b>Where We Are .....</b>	<b>528</b>
<b>18.49</b>	<b>XQuery Core .....</b>	<b>529</b>
	Restricted XQuery Subset .....	529
	Normalization .....	531
<b>18.50</b>	<b>Typing .....</b>	<b>537</b>
	Type-Based Simplifications .....	545
<b>18.51</b>	<b>XQuery Compilation .....</b>	<b>550</b>
	Representing Sequences .....	551
	Target Language .....	553
<b>18.52</b>	<b>Compiling FLWORs .....</b>	<b>566</b>
	Example .....	566
	Representation Issues .....	569



# Table of Contents IX

Relational Algebra for FLWOR Blocks .....	575
Nested Iterations .....	579
Resulting Relational Algebra Plans .....	590