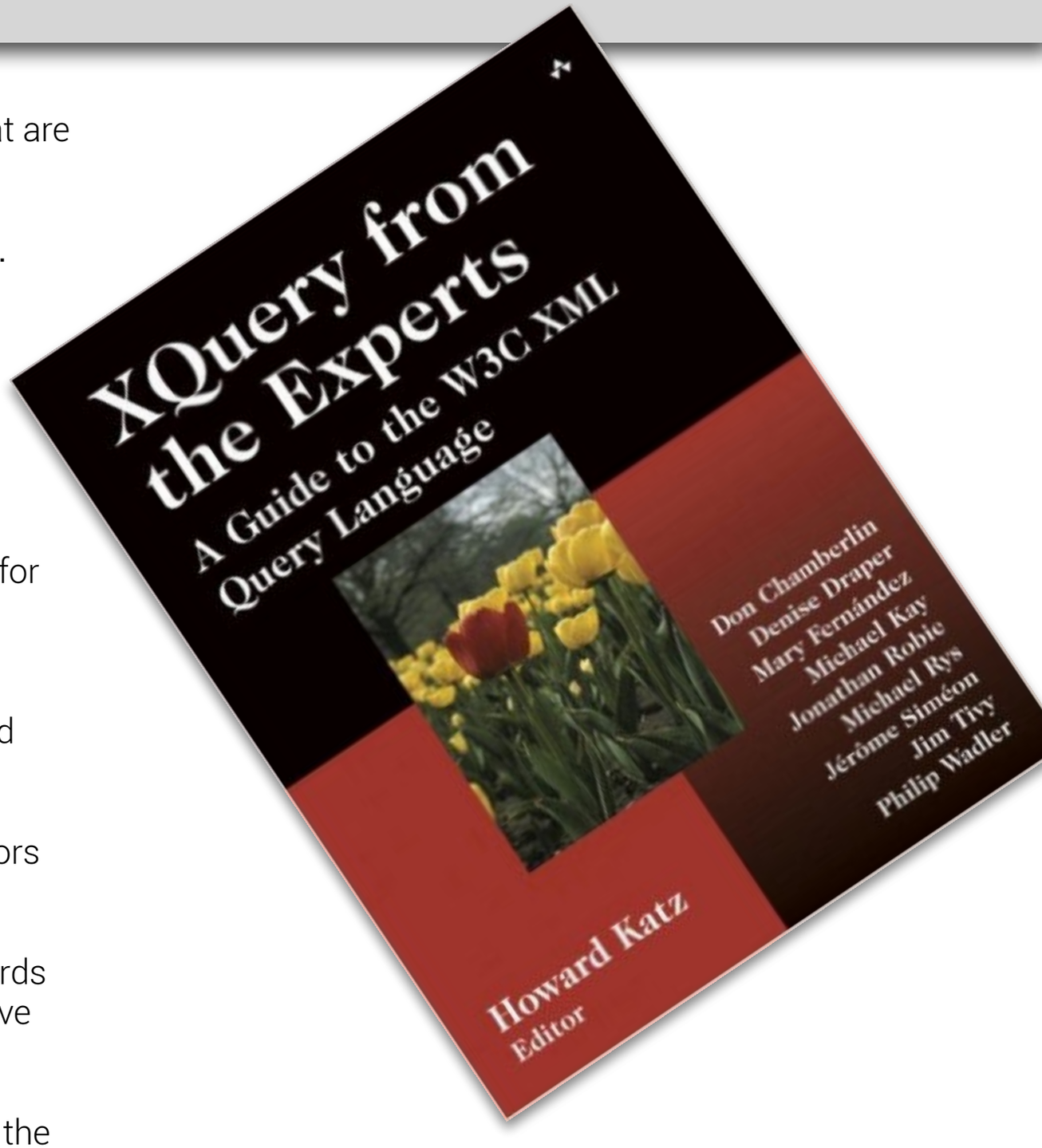
MANCHESTER 1824

The University of Manchester

Querying Data on the Web

Alvaro A A Fernandes
School of Computer Science
University of Manchester

Acknowledgements [1]



- Some of these slides mostly contain text and examples that are most often taken *verbatim* from the book:
  - XQuery from the Experts. D. Chamberlin, D. Draper, M. Fernández et al. Addison-Wesley, 2004. ISBN 0-321-18060-7. © Pearson Education Limited 2004
- In particular, from Chapter 1
  - XQuery: A Guided Tour. Jonathan Robie.
- These slides were put together from the above publication for educational purposes only.
- Any changes made either reflect updates to XQuery since publication or are the result of minor editing, adaptation and extension for use in teaching.
- The author of these slides is solely responsible for any errrors they contain.
- No claims are made by the author of these slides with regards to any form of intellectual property rights wherever the above authors are acknowledged.
- The author of these slides is very grateful to the authors of the works mentioned above.

Acknowledgements [2]

- Some of these slides mostly contain text, examples and materials that are most often taken *verbatim* from the book::
  - An Introduction to XML and Web Technologies. Anders Møller and Michael Schwartzbach. Addison-Wesley, 2006. ISBN 978-0-321-26966-9. http://www.brics.dk/ixwt/ © Pearson Education Limited 2006
- These slides were put together from the above publication for educational purposes only.
- Any changes made either reflect updates to XQuery since publication or are the result of minor editing, adaptation and extension for use in teaching.
- The author of these slides is solely responsible for any errrors they contain.
- No claims are made by the author of these slides with regards to any form of intellectual property rights wherever the above authors are acknowledged.
- The author of these slides is very grateful to the authors of the work above.

Acknowledgements [3]

- Some of these slides mostly contain text, examples and materials that are most often taken *verbatim* from the book:
  - XQuery. Priscilla Walmsley. O'Reilly, 2007. ISBN 978-0-596-00634-1. http://www.datypic.com/books/xquery/ © Priscilla Walmsley 2007
- These slides were put together from the above publication for educational purposes only.
- Any changes made either reflect updates to XQuery since publication or are the result of minor editing, adaptation and extension for use in teaching.
- The author of these slides is solely responsible for any errrors they contain.
- No claims are made by the author of these slides with regards to any form of intellectual property rights wherever the above author is acknowledged.
- The author of these slides is very grateful to the author of the work above.

Acknowledgements [4]

- Some of these slides mostly contain text and examples that are most often taken *verbatim* from the tutorial:
  - XQuery Tutorial. Peter Fankhauser, Philip Wadler. XML 2001, Orlando, 9-14 December 2001 http://homepages.inf.ed.ac.uk/wadler/papers/xquery-tutorial/xquery-tutorial.pdf
- These slides were put together from the above publication for educational purposes only.
- Any changes made either reflect updates to XQuery since publication or are the result of minor editing, adaptation and extension for use in teaching.
- The author of these slides is solely responsible for any errrors they contain.
- No claims are made by the author of these slides with regards to any form of intellectual property rights wherever the above authors are acknowledged.
- The author of these slides is very grateful to the authors of the work above.

Acknowledgements [5]

- Some of these slides mostly contain text and examples that are most often taken *verbatim* from:
  - Storing and Querying Large XML Instances. Christian Grün. Doctoral dissertation, University of Konstanz, 2010. http://kops.uni-konstanz.de/handle/123456789/6106
- These slides were put together from the above publication for educational purposes only.
- Any changes made either reflect updates to XQuery since publication or are the result of minor editing, adaptation and extension for use in teaching.
- The author of these slides is solely responsible for any errrors they contain.
- No claims are made by the author of these slides with regards to any form of intellectual property rights wherever the above authors are acknowledged.
- The author of these slides is very grateful to the author of the work above.


Universität Konstanz

Storing and Querying Large XML Instances

Christian Grün

Dissertation zur Erlangung des akademischen Grades Doktor der Naturwissenschaften (Dr. rer. nat.)

Fachbereich Informatik und Informationswissenschaft
Mathematisch-Naturwissenschaftliche Sektion
Universität Konstanz

Referenten:
Prof. Dr. Marc H. Scholl
Prof. Dr. Marcel Waldvogel
Tag der mündlichen Prüfung: 22. Dezember 2010

Acknowledgements [6]

- Some of these slides mostly contain text and examples that are most often taken *verbatim* from didactic material made available by **João Moura Pires, Carlos Damásio, Joaquim Aparício** and **Pedro Barahona**, all from **Universidade Nova de Lisboa**.

- These slides were put together from the above publication for educational purposes only.

- Any changes made either reflect updates to XQuery since publication or are the result of minor editing, adaptation and extension for use in teaching.

- The author of these slides is solely responsible for any errrors they contain.

- No claims are made by the author of these slides with regards to any form of intellectual property rights wherever the above authors are acknowledged.

- The author of these slides is very grateful to the authors mentioned above.

Acknowledgements [7]

- Some of these slides mostly contain text and examples that are most often taken *verbatim* from W3C documents mentioned in the slide titled **W3C Documents**.

Topics So Far

- We have revised the basic notions of what a database management system (DBMS) is, what role the languages a DBMS supports play, and how DBMS-centred applications are designed.
- We have adopted an approach to describing the internal architecture of DBMSs that allowed us to discuss the strengths and weaknesses of classical DBMSs and the recent trends in the way organizations use DBMS how architectures have been diversifying recently.
- We have revised the various kinds of query languages by looking into the relational calculi, a relational algebra and SQL
- We have looked in detail into logical optimization, i.e., heuristic-based rewriting of algebraic plans.
- We have explored some of the basic strategies for designing physical operators and for combining them into query evaluation plans (QEPs).
- We have briefly discussed some of the challenges in cost-based QEP selection, in particular the hard problem of choosing a join order.
- We have found out how data partitioning strategies, allied with the algebraic properties of query languages, make it possible to automatically parallelize QEP execution.

Learning Objectives So Far

- We have aimed to revise and reinforce undergraduate-level understanding of DBMS as software systems, rather than as software development components.
- We have aimed to start a postgraduate-level exploration of query languages.
- We have aimed to explore classical query processing, including optimization and evaluation, both centralized and parallel.
- We have aimed to be ready to use what we learned to exploring querying data on the Web, in the narrower sense, starting with XML/XQuery.

Topics This Week

- We will take a quick, example-driven tour of XQuery, to remind ourselves of the basic characteristics of the language.
- We will introduce and briefly explore an algebraic view of XQuery, and some equivalence laws that arise, on the basis of which one can conceive of a rewriting approach to logical optimization for the language.
- We will then study storage, optimization and evaluation of XQuery as implemented in one specific XML native DBMS, viz., BaseX.

Learning Objectives This Week

- Having completed our exploration of classical query processing, we are now ready to use what we have learnt and explore querying data on the Web, in the narrower sense, starting with XML/XQuery.
- We will aim to acquire a deeper understanding of some of the challenges arising in querying over large XML collections using XQuery.

Teaching Day 3 Outline

1. A Quick Tour of XQuery
   I. Capabilities, Uses and Processing Scenarios
   II. Contrasts with XPath, XSLT, SQL, XML Schema
   III. Examples
2. An Algebraic View of XQuery
   I. XQuery Select, Project, Join
   II. XQuery Core
   III. Translations
      i. XPath to XQuery FLWOR
      ii. XQuery FLWOR to XQuery Core
   IV. Equivalence Laws
3. The BaseX Storage Model
   I. XML Encoding
   II. Index Structures
4. Query Optimization in BaseX
   I. Overview: Non-Classical Capabilities, Query Processing Model, Query Engine Architecture
   II. Static Optimization
   III. Dynamic Optimization
5. Query Evaluation in BaseX
   I. General Evaluation Strategy
   II. Example Compiled Queries, Optimized Queries, and Evaluation Plans

Lecture 11

MANCHESTER 1824

The University of Manchester

A Quick Tour of XQuery

MANCHESTER 1824
The University of Manchester

XQuery: Capabilities, Uses and Processing Scenarios

XQuery Capabilities

- XQuery has a rich set of features that allow many different types of operations on XML data and documents, including:
  - Selecting information based on specific criteria
  - Filtering out unwanted information
  - Searching for information within a document or set of documents
  - Joining data from multiple documents or collections of documents
  - Sorting, grouping, and aggregating data
  - Transforming and restructuring XML data into another XML vocabulary or structure
  - Performing arithmetic calculations on numbers and dates
  - Manipulating strings to reformat text
  - Searching full text
  - Updating documents in store

This is out of our scope.

This is out of scope too.

from: XQuery. Priscilla Walmsley. O'Reilly, 2007. ISBN 978-0-596-00634-1.

XQuery Uses

- There are as many reasons to query XML as there are reasons to use XML.
- Some examples of common uses for the XQuery language are:
  - Extracting information from a relational database for use in a web service
  - Generating reports on data stored in a database for presentation on the Web as XHTML
  - Searching textual documents in a native XML database and presenting the results
  - Pulling data from databases or packaged software and transforming it for application integration
  - Combining content from traditionally non-XML sources to implement content management and delivery
  - Ad hoc querying of standalone XML documents for the purposes of testing or research

from: XQuery. Priscilla Walmsley. O'Reilly, 2007. ISBN 978-0-596-00634-1.

XQuery Processing Scenarios [1]

- XQuery's primary use is for querying bodies of XML content that are stored in databases.
- Some of the earliest XQuery implementations were in native XML database products.
- The term *native XML database* generally refers to a database that is designed for XML content from the ground up, as opposed to a traditionally relational database that has been extended to store XML content.
- Rather than being oriented around tables and columns, the XQuery data model is based on hierarchical documents and collections of documents.
- Native XML databases are most often used for narrative content and other data whose structure is less regular and whose content is less predictable than what one would typically store in a relational database.

from: XQuery. Priscilla Walmsley. O'Reilly, 2007. ISBN 978-0-596-00634-1.

XQuery Processing Scenarios [2]

- One example (among many) of a native XML database product is **BaseX** (http://basex.org/).
- XML database products provide the traditional capabilities of databases, such as data storage, indexing, querying, loading, extracting, backup, and recovery.
- Major relational database products (including by Oracle, IBM, and Microsoft) also have support for XML and XQuery.
- Early implementations of XML in relational databases involved storing XML in table columns as blobs or character strings and providing query access to those columns.
- However, these vendors have gradually blurred the line between native XML databases and relational databases with new features that allow one to store XML natively.

from: XQuery. Priscilla Walmsley. O'Reilly, 2007. ISBN 978-0-596-00634-1.

XQuery Processing Scenarios [3]

- Other XQuery processors are not packaged as a database product, but take the shape of a software component that works independently and is meant for use in, or with, encompassing software products.

- They may be used on physical XML documents stored as files on a file system or on the Web.

- They may also operate on XML data that is passed in memory from some other process.

- The most notable product in this category is **Saxon** (http://saxon.sourceforge.net/).

MANCHESTER 1824

The University of Manchester

XQuery: Contrasts with XPath, XSLT, SQL, XML Schema

XQuery: Design Desiderata

- The XQuery working group set out to design a language that would:
  - Be useful for both highly structured as well as semistructured documents
  - Be protocol-independent, allowing a query to be evaluated on any system with predictable results
  - Be declarative, rather than procedural
  - Be strongly typed, allowing queries to be compiled to identify possible errors and to optimize their evaluation
  - Allow querying across collections of documents
  - Use and share as much as possible with appropriate W3C recommendations, such as XML, Namespaces, XML Schema, and XPath

from: XQuery. Priscilla Walmsley. O'Reilly, 2007. ISBN 978-0-596-00634-1.

XQuery v. XPath [1]

- XPath started out as a language for selecting elements and attributes from an XML document while traversing its hierarchy and filtering out unwanted content.
- XPath 1.0 is a fairly simple yet useful recommendation that specifies path expressions and a limited set of functions.
- XPath 2.0 is much more than that, encompassing a wide variety of expressions and functions, not just path expressions.

from: XQuery. Priscilla Walmsley. O'Reilly, 2007. ISBN 978-0-596-00634-1.

XQuery v. XPath [2]

- XQuery 1.0 and XPath 2.0 overlap to a very large degree.
- They have the same data model and the same set of built-in functions and operators.
- XPath 2.0 is essentially a subset of XQuery 1.0.
- XQuery has a number of features that are not included in XPath, such as FLWOR (pronounced "flower") expressions (on which, more later) and XML constructors.
- This is because these features are not relevant to selecting, but instead are used for structuring or sorting query results.

from: XQuery. Priscilla Walmsley. O'Reilly, 2007. ISBN 978-0-596-00634-1.

XQuery v. XPath [3]

- The two languages are consistent in that any expression that is valid in both languages evaluates to the same value using both languages.
- Almost all XPath 1.0 expressions are still valid in XPath 2.0, with a few slight differences in the way values are handled.
- XPath 2.0 was built with the intention that it would be as backward-compatible with XPath 1.0 as possible.

XQuery v. XSLT [1]

- XSLT is a W3C language for transforming XML documents into other XML documents or, indeed, documents of any kind.
- There is a lot of overlap in the capabilities of XQuery and XSLT.
- In fact, the XSLT 2.0 standard is based upon XPath 2.0, so it has the same data model and supports all the same built-in functions and operators as XQuery, as well as many of the same expressions.

from: XQuery. Priscilla Walmsley. O'Reilly, 2007. ISBN 978-0-596-00634-1.

XQuery v. XSLT [2]

- XSLT implementations are generally optimized for transforming entire documents, therefore they load the entire input document into memory.
- XQuery is optimized for selecting fragments of data, e.g., from a database.
- It is designed to be scalable and to take advantage of database features such as indices for optimization.
- XQuery has a more compact non-XML syntax, which is sometimes easier to read and write (and embed in program code) than the XML syntax of XSLT.

from: XQuery. Priscilla Walmsley. O'Reilly, 2007. ISBN 978-0-596-00634-1.

XQuery v. XSLT [3]

- XQuery is designed to select from a collection of documents as opposed to a single document.
- Elegant iteration makes it easy to join information across (and within) documents.
- Also, XSLT 2.0 stylesheets can operate on multiple documents, but XSLT processors are not particularly optimized for this less common use case.

XQuery v. XSLT [4]

- Generally, when transforming an entire XML document from one XML vocabulary to another, it makes more sense to use XSLT.
- When your main focus is selecting a subset of data from an XML document or database, you should use XQuery.

from: XQuery. Priscilla Walmsley. O'Reilly, 2007. ISBN 978-0-596-00634-1.

XQuery v. SQL [1]

- XQuery borrows ideas from SQL, and many of the designers of XQuery were also designers of SQL.
- The line between XQuery and SQL may seem clear: XQuery is for XML, and SQL is for relational data.
- However, increasingly this line is blurred, because relational database vendors have provided XML frontends on their products and allowing XML to be stored in traditionally relational databases.

from: XQuery. Priscilla Walmsley. O'Reilly, 2007. ISBN 978-0-596-00634-1.

XQuery v. SQL [2]

- XQuery is unlikely to replace SQL for the highly structured data that is traditionally stored in relational databases.
- Most likely, the two will coexist, with XQuery being used to query less-structured data, or data that is destined for an XML-based application, and SQL continuing to be used for highly structured relational data.

from: XQuery. Priscilla Walmsley. O'Reilly, 2007. ISBN 978-0-596-00634-1.

XQuery v. SQL [3]

- Query engines underlying XQuery implementations are rather different from those underlying SQL implementations.
- XQuery query engines tend also to be more different amongst themselves than SQL query engines do.
- This is partly to do with technological maturity, and partly to do with the greater complexity of core XQuery, as well as, possibly, other reasons.
- The greater complexity stems from a more flexible data model, a more varied type system, and the need to use and share preexisting standards.

XQuery v. XML Schema [1]

- XML Schema is a W3C standard for defining schemas, that can be used to validate XML documents and to assign types to XML elements and attributes.
- XQuery uses the type system of XML Schema, which includes built-in types that represent common scalar datatypes such as decimal, date, and string.
- XML Schema also specifies a language for defining your own types based on the built-in types.

from: XQuery. Priscilla Walmsley. O'Reilly, 2007. ISBN 978-0-596-00634-1.

XQuery v. XML Schema [2]

- If an input document to a query has a schema, the types can be used when evaluating expressions on the input data.

- For example, if your `item` element has a `quantity` attribute, and you know from the schema that the type of the `quantity` attribute is `xs:integer`, you can perform sorts or other operations on that attribute's value without converting it to an integer in the query.

- This also has the advantages of allowing the processor to better optimize the query and to catch errors earlier.

from: XQuery. Priscilla Walmsley. O'Reilly, 2007. ISBN 978-0-596-00634-1.

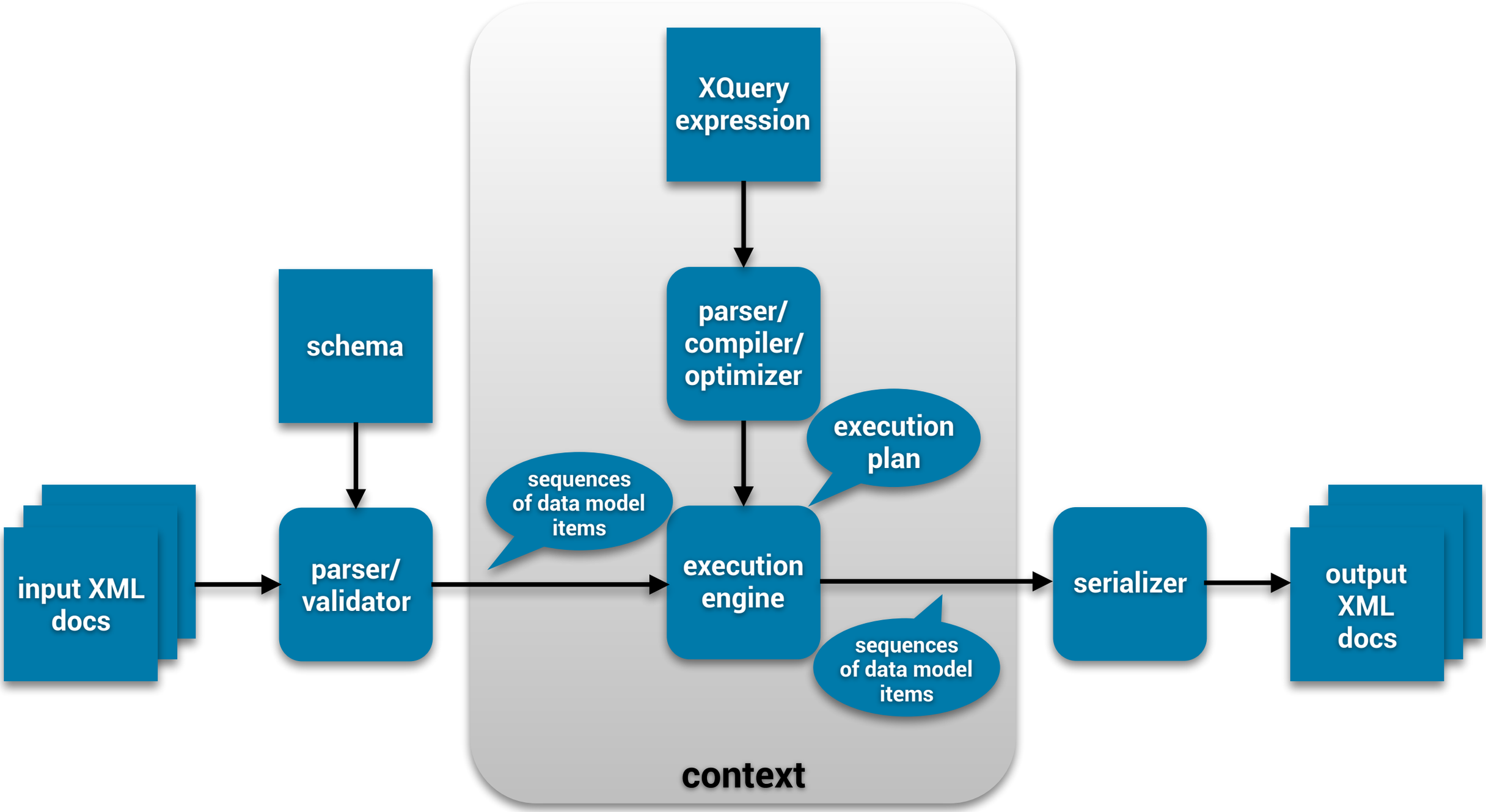
XQuery v. XML Schema [3]

- Note, however, that XQuery users are not required to use schemas.
- It is entirely possible to write a complete query with no mention of schemas or any of the schema types.
- However, a rich set of functions and operators are provided that generally operate on typed data, so it is useful to understand the type system and use the built-in types, even if no schema is present.

from: XQuery. Priscilla Walmsley. O'Reilly, 2007. ISBN 978-0-596-00634-1.

XQuery: Examples

A Simplified View of the XQuery Processing Model



A DTD for Bibliography Data

```
<!ELEMENT bib          (book* )>
<!ELEMENT book         (title,  (author+ | editor+ ), publisher, price )>
<!ATTLIST book         year CDATA  #REQUIRED >
<!ELEMENT author       (last, first )>
<!ELEMENT editor       (last, first, affiliation )>
<!ELEMENT title        (#PCDATA )>
<!ELEMENT last         (#PCDATA )>
<!ELEMENT first        (#PCDATA )>
<!ELEMENT affiliation  (#PCDATA )>
<!ELEMENT publisher    (#PCDATA )>
<!ELEMENT price        (#PCDATA )>
```

bib.xml: Example Bibliography Data

```
<bib>
    <book year="1994">
        <title>TCP/IP Illustrated</title>
        <author><last>Stevens</last><first>W.</first></author>
        <publisher>Addison-Wesley</publisher>
        <price>65.95</price>
    </book>
    <book year="1992">
        <title>Advanced Programming in the UNIX Environment</title>
        <author><last>Stevens</last><first>W.</first></author>
        <publisher>Addison-Wesley</publisher>
        <price>65.95</price>
        </book>
    <book year="2000">
        <title>Data on the Web</title>
        <author><last>Abiteboul</last><first>Serge</first></author>
        <author><last>Buneman</last><first>Peter</first></author>
        <author><last>Suciu</last><first>Dan</first></author>
        <publisher>Morgan Kaufmann Publishers</publisher>
        <price>65.95</price>
    </book>
    <book year="1999">
        <title>The Economics of Technology and Content for Digital TV</title>
        <editor>
           <last>Gerbarg</last>
           <first>Darcy</first>
           <affiliation>CITI</affiliation>
        </editor>
        <publisher>Kluwer Academic Publishers</publisher>
        <price>129.95</price>
    </book>
</bib>
```

Example: Path Expression [1]

Query

```
doc("books.xml")
```

This expression opens books.xml using the doc() function and returns its entire content.

Result

```
<!-- books.xml -->
<bib>
<book year="1994">
  <title>TCP/IP Illustrated</title>
  <author>
    <last>Stevens</last>
    <first>W.</first>
  </author>
  <publisher>Addison-Wesley</publisher>
  <price>65.95</price>
</book>
<book year="1992">
  <title>Advanced Programming in the UNIX Environment</title>
.
.
.
```

Example: Path Expression [2]

Query

```
doc("books.xml")/bib/book
```

This expression refines the previous one by using /bib to select the bib element at the top of the document, then using /book to select the book elements within the bib element. This path expression contains three steps.

Result

```
<book year="1994">
  <title>TCP/IP Illustrated</title>
  <author>
    <last>Stevens</last>
    <first>W.</first>
  </author>
  <publisher>Addison-Wesley</publisher>
  <price>65.95</price>
</book>
<book year="1992">
  <title>Advanced Programming in the UNIX Environment</title>
.
.
.
.
```

Example: Path Expression [3]

Query

```
doc("books.xml")//book
```

The same books could have been found by a query that uses the double slash, //, to select all of the book elements contained in the document, regardless of the level at which they are found.

Result

```
<book year="1994">
  <title>TCP/IP Illustrated</title>
  <author>
    <last>Stevens</last>
    <first>W.</first>
  </author>
  <publisher>Addison-Wesley</publisher>
  <price>65.95</price>
</book>
<book year="1992">
  <title>Advanced Programming in the UNIX Environment</title>
.
.
.
```

Example: Path Expression [4]

Query

```
doc("books.xml")/bib/*
```

The same books could have been found by a query that uses a wildcard, *, to select all of the elements contained in the document rooted by the bib element.

Result

```
<book year="1994">
  <title>TCP/IP Illustrated</title>
  <author>
    <last>Stevens</last>
    <first>W.</first>
  </author>
  <publisher>Addison-Wesley</publisher>
  <price>65.95</price>
</book>
<book year="1992">
  <title>Advanced Programming in the UNIX Environment</title>
.
.
.
```

Example: Path Expression [5]

Query

```
doc("books.xml")/bib/book/*
```

The wildcard here selects all of the elements contained in book elements.

Result

```
<title>TCP/IP Illustrated</title>
<author>
  <last>Stevens</last>
  <first>W.</first>
</author>
<publisher>Addison-Wesley</publisher>
<price>65.95</price>
<title>Advanced Programming in the UNIX Environment</title>
<author>
  <last>Stevens</last>
  <first>W.</first>
</author>
<publisher>Addison-Wesley</publisher>
<price>65.95</price>
.
.
.
```

Example: Path Expression [6]

Query

```
doc("books.xml")/bib//*
```

The wildcard here selects all of the elements contained the document (including the root one). It returns a longer sequence than previous queries.

Result

```
<title>TCP/IP Illustrated</title>
<author>
  <last>Stevens</last>
  <first>W.</first>
</author>
<publisher>Addison-Wesley</publisher>
<price>65.95</price>
<title>Advanced Programming in the UNIX Environment</title>
<author>
  <last>Stevens</last>
  <first>W.</first>
</author>
<publisher>Addison-Wesley</publisher>
<price>65.95</price>
.
.
.
```

Example: Path Expression [7]

Query

```
declare namespace dc="http://purl.org/dc/elements/1.1/";
                                    doc("books.xml")//dc:*
```

The wildcard here selects all of the elements that belong to the namespace declared in the prolog.

Assume that the first book in "books.xml" had been declared as below, i.e., with the title element referencing a namespace.

```
<book year="1994" xmlns:dcx="http://purl.org/dc/elements/1.1/">
    <dcx:title>TCP/IP Illustrated</dcx:title>
    <author><last>Stevens</last><first>W.</first></author>
    <publisher>Addison-Wesley</publisher>
    <price>65.95</price>
</book>
```

Result

```
<dcx:title xmlns:dcx="http://purl.org/dc/elements/1.1/">
   TCP/IP Illustrated
</dcx:title>
```

Example: Path Expression [8]

Query

```
doc("books.xml")//*:title
```

The wildcard here selects all of the title elements irrespective of namespace.

Assume that the first book in "books.xml" had been declared as below, i.e., with the title element referencing a namespace.

```
<book year="1994" xmlns:dcx="http://purl.org/dc/elements/1.1/">
    <dcx:title>TCP/IP Illustrated</dcx:title>
    <author><last>Stevens</last><first>W.</first></author>
    <publisher>Addison-Wesley</publisher>
    <price>65.95</price>
</book>
```

Result

```
<dcx:title xmlns:dcx="http://purl.org/dc/elements/1.1/">
  TCP/IP Illustrated
</dcx:title>
<title>
  Advanced Programming in the UNIX Environment
</title>
<title>
  Data on the Web
</title>
<title>
  The Economics of Technology and Content for Digital TV
</title>
```

Example: Path Expression [9]

Query

```
doc("books.xml")//*/@year
```

The wildcard here selects all of the year attributes in the input.

Result

```
year="1994"
year="1992"
year="2000"
year="1999"
```

Example: Path Expression [10]

Query

```
doc("books.xml")//*/@*
```

The wildcard here selects all of the attributes in the input. For the input given, the result happens to be the same as in the preceding query.

Result

```
year="1994"
year="1992"
year="2000"
year="1999"
```

Example: Path Expression [11]

Query

```
doc("books.xml")//text()
```

This query selects all text nodes. If we use, node() we select all nodes irrespective of their kind (i.e., element(), comment(), etc.).

Result

```
TCP/IP Illustrated
Stevens
W.
Addison-Wesley
65.95
Advanced Programming in the UNIX Environment
Stevens
W
Addison-Wesley
65.95
Data on the Web
Abiteboul
Serge
.
.
.
```

from: XQuery: A Guided Tour. Jonathan Robie. Ch. 1 in (D. Chamberlin, D. Draper, M. Fernández et al., 2004), and from material made available by Pires et al.

Example: Path Expression [12]

Query

```
doc("books.xml")/bib/book/author[last="Stevens"]
```

This query returns only authors for which last="Stevens" is true.

Result

```
<author>
  <last>Stevens</last>
  <first>W.</first>
</author>
<author>
  <last>Stevens</last>
  <first>W.</first>
</author>
```

Example: Path Expression [13]

Query

```
doc("books.xml")/bib/book/author/last[.="Stevens"]
doc("books.xml")/bib/book/author/last[data(.)="Stevens"]
doc("books.xml")/bib/book/author/last[text()="Stevens"]
```

Different queries, using predicates on an attribute, that return the same result.

Result

```
<last>Stevens</last>
<last>Stevens</last>
```

Example: Path Expression [14]

Query

```
doc("books.xml")/bib/book/author/last="Stevens"
```

This query asks whether there is an author for which last="Stevens" is true.

Result

```
true
```

Example: Path Expression [15]

Query

```
doc("books.xml")/bib/book/[author/last="Stevens"]
```

This query asks of each book whether among its authors there is one for which last="Stevens" is true.

Result

```
[
  true
]
[
  true
]
[
  false
]
[
  false
]
```

Example: Path Expression [16]

Query

```
doc("books.xml")/bib/book/author[1]
```

If a predicate contains a single numeric value, it is treated like a subscript. For instance, this expression returns the first author of each book.

Result

```
<author>
  <last>Stevens</last>
  <first>W.</first>
</author>
<author>
  <last>Stevens</last>
  <first>W.</first>
</author>
<author>
  <last>Abiteboul</last>
  <first>Serge</first>
</author>
```

Example: Path Expression [17]

Query

```
(doc("books.xml")/bib/book/author)[1]
```

Note that the expression author[1] will be evaluated for each book. If we want the first author in the entire document, we can use parentheses to force the desired precedence.

Result

```
<author>
  <last>Stevens</last>
  <first>W.</first>
</author>
```

Example: Path Expression [18]

Query

```
doc("books.xml")/bib/book/author[position()=1]
```

This query uses an explicit function rather than rely on an abbreviation.

Result

```
<author>
  <last>Stevens</last>
  <first>W.</first>
</author>
<author>
  <last>Stevens</last>
  <first>W.</first>
</author>
<author>
  <last>Abiteboul</last>
  <first>Serge</first>
</author>
```

from: XQuery: A Guided Tour. Jonathan Robie. Ch. 1 in (D. Chamberlin, D. Draper, M. Fernández et al., 2004), and from material made available by Pires et al.

Example: Path Expression [19]

Query

```
doc("books.xml")/bib/book/author[last()]
```

Since we cannot normally know the length, for the last position we use a function.

Result

```
<author>
  <last>Stevens</last>
  <first>W.</first>
</author>
<author>
  <last>Stevens</last>
  <first>W.</first>
</author>
<author>
  <last>Suciu</last>
  <first>Dan</first>
</author>
```

Example: Path Expression [20]

Query

```
doc("books.xml")/bib/book/author[position()=last()]
```

Without abbreviating, the preceding query becomes this expression, with the same result.

Result

```
<author>
  <last>Stevens</last>
  <first>W.</first>
</author>
<author>
  <last>Stevens</last>
  <first>W.</first>
</author>
<author>
  <last>Suciu</last>
  <first>Dan</first>
</author>
```

Example: Path Expression [21]

Query

```
doc("books.xml")/bib/book[last()]/author[last()]
```

This expression requests the last author of the last book.

Result

```
(: this returns the empty sequence, :)
(: as the last book has no authors, :)
(: instead, it has editors          :)
```

Example: Path Expression [22]

Query

```
doc("books.xml")/bib/book[last()-1]/author[last()]
```

This expression requests the last author of the penultimate book.

Result

```
<author>
  <last>Suciu</last>
  <first>Dan</first>
</author>
```

Example: Path Expression [23]

Query

```
(doc("books.xml")/bib/book/author)[last()]
```

This expression requests the last author to occur in "books.xml".

Result

```
<author>
  <last>Suciu</last>
  <first>Dan</first>
</author>
```

from: XQuery: A Guided Tour. Jonathan Robie. Ch. 1 in (D. Chamberlin, D. Draper, M. Fernández et al., 2004), and from material made available by Pires et al.

Example: Path Expression [24]

Query

```
doc("books.xml")/bib/book[author/last="Stevens"][1]
```

This expression filters by content and by position. Note the adjacent predicates, each enclosed in square brackets.

Result

```
<book year="1994">
  <title>TCP/IP Illustrated</title>
  <author>
    <last>Stevens</last>
    <first>W.</first>
  </author>
  <publisher>Addison-Wesley</publisher>
  <price>65.95</price>
</book>
```

Example: Path Expression [25]

Query

```
doc("books.xml")/bib/book[author/last="Stevens" and position()=1]
```

This expression also filters by content and by position but note that it does not necessarily return the same result as the preceding one.

Result

```
<book year="1994">
  <title>TCP/IP Illustrated</title>
  <author>
    <last>Stevens</last>
    <first>W.</first>
  </author>
  <publisher>Addison-Wesley</publisher>
  <price>65.95</price>
</book>
```

Example: Path Expression [26]

Query

```
doc("books.xml")/bib/book/@year
```

This query returns the year attribute of each book element.

Result

```
year="1994"
year="1992"
year="2000"
year="1999"
```

Example: Constructors [1]

Query

```
let $b :=
'<book year="1977">
  <title>Harold and the Purple Crayon
  </title>
  <author>
    <last>Johnson</last>
    <first>Crockett</first>
  </author>
  <publisher>HarperCollins Juvenile Books
  </publisher>
  <price>14.95</price>
</book>'
return
  typeswitch ($b)
     case    item() return ($b, 'is an item')
     default        return ($b, 'is not an item')
```

This query constructs an XML item using XML syntax.

Result

```
<book year="1977">
  <title>Harold and the Purple Crayon
  </title>
  <author>
    <last>Johnson</last>
    <first>Crockett</first>
  </author>
  <publisher>HarperCollins Juvenile Books
  </publisher>
  <price>14.95</price>
</book>
is an item
```

Example: Constructors [2]

Query

```
let $d := document{
<!-- books.xml -->,
<book year="1977">
  <title>Harold and the Purple Crayon
  </title>
  <author>
    <last>Johnson</last>
    <first>Crockett</first>
  </author>
  <publisher>HarperCollins Juvenile Books
  </publisher>
  <price>14.95</price>
</book>
}
return
  $d
```

This query constructs an XML document using the document() constructor.

Result

```
<!-- books.xml -->
<book year="1977">
  <title>Harold and the Purple Crayon
  </title>
  <author>
    <last>Johnson</last>
    <first>Crockett</first>
  </author>
  <publisher>HarperCollins Juvenile Books
  </publisher>
  <price>14.95</price>
</book>
```

Example: Constructors [3]

Query

```
<example>
  <p>  Here is a query.                          </p>
  <eg> doc("books.xml")//book[1]/title           </eg>
  <p>  Here is the result of the above query.    </p>
  <eg> {doc("books.xml")//book[1]/title}         </eg>
</example>
```

This query constructs illustrates how curly braces delimit expressions that are dynamically evaluated.

Result

```
<example>
  <p>  Here is a query.                          </p>
  <eg> doc("books.xml")//book[1]/title           </eg>
  <p>  Here is the result of the above query.    </p>
  <eg>
    <title>TCP/IP Illustrated</title>
  </eg>
</example>
```

Example: Constructors [4]

Query

```
<titles count="{ count(doc('books.xml')//title) }">
  {
    doc("books.xml")//title
  }
</titles>
```

This query creates a document that is a list of book titles whilst making the count of titles an attribute of the 'titles' element.

Result

```
<titles count="4">
  <title>TCP/IP Illustrated
  </title>
  <title>Advanced Programming in the UNIX …
  </title>
  <title>Data on the Web
  </title>
  <title>The Economics of Technology and …
</titles>
```

Example: Constructors [5]

Query

```
element title {
            "Harold and the Purple Crayon"
}
```

This query creates a 'title' element using a computed element constructor approach.

Result

```
<title>Harold and the Purple Crayon</title>
```

Example: Constructors [6]

Query

```
element book
{
  attribute year { 1977 },
  element author
  {
   element first { "Crockett" },
   element last { "Johnson" }
  },
   element publisher {"HarperCollins Juvenile Books"},
   element price { 14.95 }
}
```

This query creates a 'book' element, with nested attribute and node elements using computed element constructors.

Result

```
<book year="1977">
  <author>
    <first>Crockett</first>
    <last>Johnson</last>
  </author>
  <publisher>HarperCollins Juvenile Books
  </publisher>
  <price>14.95</price>
</book>
```

Example: Combining and Restructuring [1]

Query

```
for    $b in doc("books.xml")//book
where  $b/@year = "2000"
return $b/title
```

This query returns the title of each book that was published in the year 2000.

Result

```
<title>Data on the Web</title>
```

Example: Combining and Restructuring [2]

Query

```
for    $b in doc("books.xml")//book
let    $c := $b/author
return
  <book>{
     $b/title,
     <count>{
        count($c) }
     </count>}
  </book>
```

This query returns the title of each book with the number of authors of the book.

Result

```
<book>
  <title>TCP/IP Illustrated</title>
  <count>1</count>
</book>
<book>
  <title>Advanced Programming … <title>
  <count>1</count>
</book>
<book>
  <title>Data on the Web</title>
  <count>3</count>
</book>
<book>
  <title>The Economics of Technology … </title>
  <count>0</count>
</book>
```

Example: Combining and Restructuring [3]

Query

```
for      $t in doc("books.xml")//title
order by $t
return   $t
```

This query returns an ordered list of 'title' elements. Ascending order is the default.

Result

```
<title>Advanced Programming in … </title>
<title>Data on the Web</title>
<title>TCP/IP Illustrated</title>
<title>The Economics of Technology and … </title>
```

Example: Combining and Restructuring [4]

Query

```
for      $a in doc("books.xml")//author
order by $a/last  descending,
         $a/first descending
return   $a
```

One can also have more than one ordering field and ordering direction.

Result

```
<author>
  <last>Suciu</last>
  <first>Dan</first>
</author>
<author>
  <last>Stevens</last>
  <first>W.</first>
</author>
<author>
  <last>Stevens</last>
  <first>W.</first>
</author>
<author>
  <last>Buneman</last>
  <first>Peter</first>
</author>
<author>
  <last>Abiteboul</last>
  <first>Serge</first>
</author>
```

from: XQuery: A Guided Tour. Jonathan Robie. Ch. 1 in (D. Chamberlin, D. Draper, M. Fernández et al., 2004), and from material made available by Pires et al.

Example: Combining and Restructuring [5]

Query

```
let      $b := doc("books.xml")//book
for      $t in distinct-values($b/title)
let      $a1 := $b[title=$t]/author[1]
order by $a1/last, $a1/first
return   $b[title=$t]/title
```

Note that there is no need for the data used in ordering to appear in the `return` clause.

```
let      $b := doc("books.xml")//book
for      $t in distinct-values($b/title)
let      $a1 := $b[title=$t]/author[1]
stable
order by $a1/last  empty least,
         $a1/first empty least
return   $b[title=$t]/title
```

Using 'stable' and'empty least' avoids differences in implementation whto come to the fore when two values are ordered the same and when empty sequences appear.

Result

```
<title>The Economics of Technology … </title>
<title>Data on the Web</title>
<title>TCP/IP Illustrated</title>
<title>Advanced Programming in the … </title>
```

from: XQuery: A Guided Tour. Jonathan Robie. Ch. 1 in (D. Chamberlin, D. Draper, M. Fernández et al., 2004), and from material made available by Pires et al.

Example: Combining and Restructuring [6]

Query

```
for      $a in doc("books.xml")//author
order by $a/last  descending,
         $a/first descending
return   $a
```

One can also have more than one ordering field and ordering direction.

Result

```
<author>
  <last>Suciu</last>
  <first>Dan</first>
</author>
<author>
  <last>Stevens</last>
  <first>W.</first>
</author>
<author>
  <last>Stevens</last>
  <first>W.</first>
</author>
<author>
  <last>Buneman</last>
  <first>Peter</first>
</author>
<author>
  <last>Abiteboul</last>
  <first>Serge</first>
</author>
```

from: XQuery: A Guided Tour. Jonathan Robie. Ch. 1 in (D. Chamberlin, D. Draper, M. Fernández et al., 2004), and from material made available by Pires et al.

Example: Combining and Restructuring [7]

Query

```
let $authors := for $a in doc("books.xml")//author
                order by $a/last, $a/first
                return   $a
return $authors/last
```

This query does not return the author's last names in alphabetical order, because the / in $authors/last sorts the last elements in document order.

This query does return the author's last names in alphabetical order.

```
for $a in doc("books.xml")//author
order by $a/last, $a/first
return $a/last
```

Result

```
<last>Stevens</last>
<last>Stevens</last>
<last>Abiteboul</last>
<last>Buneman</last>
<last>Suciu</last>
```

```
<last>Abiteboul</last>
<last>Buneman</last>
<last>Stevens</last>
<last>Stevens</last>
<last>Suciu</last>
```

from: XQuery: A Guided Tour. Jonathan Robie. Ch. 1 in (D. Chamberlin, D. Draper, M. Fernández et al., 2004), and from material made available by Pires et al.

Example: Combining and Restructuring [8]

Query

```
for $b in doc("books.xml")//book
return
   <quote>
     {
      $b/title,
      $b/price
     }
   </quote>
```

This query uses the `return` clause to structure a new sequence of 'quote' elements.

Result

```
<quote>
  <title>TCP/IP Illustrated</title>
  <price>65.95</price>
</quote>
<quote>
  <title>Advanced Programming in … </title>
  <price>65.95</price>
</quote>
<quote>
  <title>Data on the Web</title>
  <price>65.95</price>
</quote>
<quote>
  <title>The Economics of Technology and … </title>
  <price>129.95</price>
</quote>
```

Example: Combining and Restructuring [9]

Query

```
for $a in doc("books.xml")//author
return
  <author>
   {
    string($a/first), string($a/last)
   }
  </author>
```

This query uses the `return` clause to combine the first and last names of an author into a single string.

Result

```
<author>W. Stevens</author>
<author>W. Stevens</author>
<author>Serge Abiteboul</author>
<author>Peter Buneman</author>
<author>Dan Suciu</author>
```

Example: Combining and Restructuring [10]

Query

```
for $t at $i in doc("books.xml")//title
return
  <title pos="{$i}">
    {
     string($t)
    }
  </title>
```

One can access the position of an item in the sequence from which it came using the `at` positional variable.

Result

```
<title pos="1">TCP/IP Illustrated</title>
<title pos="2">Advanced Programming … </title>
<title pos="3">Data on the Web</title>
<title pos="4">The Economics of … </title>
```

Example: Combining and Restructuring [11]

Query

```
doc("books.xml")//author/last
```

This query returns duplicate elements.

This query removes duplicates but, in order to do so, it must extract values from nodes.

```
distinct-values(doc("books.xml")//author/last)
```

```
for $l in
  distinct-values(doc("books.xml")//author/last)
return <last>{ $l }</last>
```

This query filters and reconstrucs the elements.

Result

```
<last>Stevens</last>
<last>Stevens</last>
<last>Abiteboul</last>
<last>Buneman</last>
<last>Suciu</last>
```

```
Stevens
Abiteboul
Buneman
Suciu
```

```
<last>Stevens</last>
<last>Abiteboul</last>
<last>Buneman</last>
<last>Suciu</last>
```

Example: Combining and Restructuring [12]

Query

```
for $t in doc("books.xml")//title,
    $e in doc("reviews.xml")//entry
where $t = $e/title
return
    <review-comments>
     { $t,
       $e/remarks
     }
    </review-comments>
```

This is a join query.

Result

```
<review-comments>
  <title>TCP/IP Illustrated</title>
  <remarks>Excellent technical content.</remarks>
</review-comments>
<review-comments>
  <title>TCP/IP Illustrated</title>
  <remarks>Really hard.</remarks>
</review-comments>
<review-comments>
  <title>Data on the Web</title>
  <remarks>Very good, but superseded.</remarks>
</review-comments>
```

Example: Combining and Restructuring [13]

Query

```
for $t in doc("books.xml")//title
return
  <book-and-review>
    { $t
    }
    { for $e in doc("reviews.xml")//entry
      where $e/title = $t
      return $e/remarks
    }
  </book-and-review>
```

This acts like a left outer join query. It returns an element for every book. If the book has reviews, it returns them grouped together.

Result

```
<book-and-review>
  <title>TCP/IP Illustrated</title>
  <remarks>Excellent technical content.</remarks>
  <remarks>Really hard.</remarks>
</book-and-review>
<book-and-review>
  <title>Advanced Programming in … </title>
</book-and-review>
<book-and-review>
  <title>Data on the Web</title>
  <remarks>Very good, but superseded.</remarks>
</book-and-review>
<book-and-review>
  <title>The Economics of … </title>
</book-and-review>
```

Lecture 12

MANCHESTER 1824

The University of Manchester

An Algebraic View of XQuery

XQuery: Select, Project, Join

Sample Schema: DTD v. XQuery Types

```
<!ELEMENT BOOKS  (BOOK*)>
<!ELEMENT BOOK   (AUTHOR+, TITLE, PRICE?, ISBN?)>
<!ATTLIST BOOK   YEAR CDATA #OPTIONAL>
<!ELEMENT AUTHOR (#PCDATA)>
<!ELEMENT TITLE  (#PCDATA)>
<!ELEMENT PRICE  (#PCDATA)>
<!ELEMENT ISBN   (#PCDATA)>
```

DTD

XQuery types

```
element BOOKS  { BOOK* }
element BOOK   { @YEAR?,
                 AUTHOR+, TITLE, PRICE?, ISBN? }
attribute YEAR { xsd:string  }
element AUTHOR { xsd:string  }
element TITLE  { xsd:string  }
element PRICE  { xsd:integer }
element ISBN   { xsd:string  }
```

from XQuery Tutorial. Peter Fankhauser, Philip Wadler. 2001.

Sample Data: XML v. XQuery Data Instances

XML document

```
<BOOKS>
  <BOOK YEAR="2013">
    <AUTHOR>Garcia-Molina</AUTHOR>
    <AUTHOR>Ullman</AUTHOR>
    <AUTHOR>Widom</AUTHOR>
    <TITLE>Database Systems: The Complete Book</TITLE>
    <PRICE>54.99</PRICE>
    <ISBN>129202447X</ISBN>
  </BOOK>
  <BOOK YEAR="1994">
    <AUTHOR>Stevens</AUTHOR>
    <TITLE>TCP/IP Illustrated</TITLE>
    <PRICE>32.24</PRICE>
    <ISBN>0321336313</ISBN>
</BOOK>
<BOOK YEAR="1992">
  <AUTHOR>Stevens</AUTHOR>
  <TITLE>Advanced Programming in the UNIX Environment</TITLE>
  <PRICE>41.79</PRICE>
  <ISBN>0321637739</ISBN>
</BOOK>
```

books.xml

```
<BOOK YEAR="2000">
    <AUTHOR>Abiteboul</AUTHOR>
    <AUTHOR>Buneman</AUTHOR>
    <AUTHOR>Suciu</AUTHOR>
    <TITLE>Data on the Web</TITLE>
    <PRICE>50.99</PRICE>
    <ISBN>155860622X</ISBN>
 </BOOK>
 <BOOK YEAR="2007">
    <AUTHOR>Walmsley</AUTHOR>
    <TITLE>XQuery</TITLE>
    <PRICE>26.80</PRICE>
    <ISBN>0596006349</ISBN>
  </BOOK>
</BOOKS>
```

XQuery data instances

```
element BOOK {
    attribute YEAR { "2000" },
    element AUTHOR { "Abiteboul" },
    element AUTHOR { "Buneman" },
    element AUTHOR { "Suciu" },
    element TITLE  { "Data on the Web" },
    element PRICE  { 50.99 },
    element ISBN   { "155860622X" }
}
…
```

XPath for Projection and Selection

```
let $root := doc("books.xml")
return
  (: return the title of every book published before 2000
  :)
  $root/BOOKS/BOOK[@YEAR < 2000]/TITLE

(:
=

<TITLE>TCP/IP Illustrated</TITLE>
<TITLE>Advanced Programming in the UNIX Environment</TITLE>
:)
```

XQuery FLWOR for Selection and Structuring

```
let $root := doc("books.xml")
return
  (: return the year and title of every book published before 2000
  :)
  for    $book in $root/BOOKS/BOOK
  where  $book/@YEAR < 2000
  return
    <BOOK>
    { $book/@YEAR,
      $book/TITLE
    }
    </BOOK>

(:
=

<BOOK YEAR="1994">
  <TITLE>TCP/IP Illustrated</TITLE>
</BOOK>
<BOOK YEAR="1992">
  <TITLE>Advanced Programming in the UNIX Environment</TITLE>
</BOOK>

:)
```

XQuery FLWOR for Grouping and Restructuring

```
let $root := doc("books.xml")
return
  (: return each author name followed by the title of every book s/he published
  :)
  for $author in distinct-values($root/BOOKS/BOOK/AUTHOR)
  return
    <AUTHOR NAME="{ $author }">
    {
      $root/BOOKS/BOOK[AUTHOR = $author]/TITLE
    }
    </AUTHOR>

(:
=

<AUTHOR NAME="Garcia-Molina">
  <TITLE>Database Systems: The Complete Book</TITLE>
</AUTHOR>
<AUTHOR NAME="Ullman">
  <TITLE>Database Systems: The Complete Book</TITLE>
</AUTHOR>
<AUTHOR NAME="Widom">
  <TITLE>Database Systems: The Complete Book</TITLE>
</AUTHOR>
<AUTHOR NAME="Stevens">
  <TITLE>TCP/IP Illustrated</TITLE>
  <TITLE>Advanced Programming in the UNIX Environment</TITLE>
</AUTHOR>
…
:)
```

XQuery FLWOR for Joining and Restructuring

```
let $amazon   := doc("amazon.xml"), $fatbrain := doc("fatbrain.xml")
return
  (: return the title and respective prices of
     every book that costs more in Amazon than in Fatbrain
  :)
  for $b1 in $amazon/BOOKS/BOOK, $b2 in $fatbrain/BOOKS/BOOK
  where $b1/ISBN  = $b2/ISBN and $b1/PRICE > $b2/PRICE
  return
    <BOOK>
    { $b1/TITLE }
      <AT-AMAZON>  {$b1/PRICE}</AT-AMAZON>
      <AT-FATBRAIN>{$b2/PRICE}</AT-FATBRAIN>
    </BOOK>

(:
=

<BOOK>
  <TITLE>Database Systems: The Complete Book</TITLE>
  <AT-AMAZON>
    <PRICE>54.99</PRICE>
  </AT-AMAZON>
  <AT-FATBRAIN>
    <PRICE>45.68</PRICE>
  </AT-FATBRAIN>
</BOOK>
…

:)
```

- The default is to sort in document order and eliminate duplicates (SIDOAED).
- Wrapping the outermost **`for`** with **`unordered(.)`** gives the optimizer freedom.
- Placing **`order by`** before **`return`** allows one to impose value-based orderings.

XQuery Core

XQuery Core

- XQuery Core is a simplified syntactic subset of XQuery.
- The main features of XQuery Core are:
  - only one iterator variable in a **`for`** clause iteration;
  - no **`where`** clause;
  - only simple path expressions of the form *iteratorVariable/Axis::NodeTest*;
  - only simple **`element`** and **`attribute`** constructors;
  - **`order by`**;
  - function calls.

from XQuery Tutorial. Peter Fankhauser, Philip Wadler. 2001.

Formal Static and Dynamic Semantics [1]

- XQuery Core has a static semantics in the form of type inference rules.
- These map XQuery expressions to types and specify under what conditions an expression is well-typed.
- The semantics is called static, because ill-typed expressions are identified at query-analysis time, i.e., before the query is evaluated.

from http://www.w3.org/TR/2001/WD-query-semantics-20010607/

Formal Static and Dynamic Semantics [2]

- XQuery Core has a dynamic, or operational, semantics in the form of value inference rules.
- These map XQuery expressions to values.
- A dynamic semantics guarantees that every expression can be reduced to a value and thus serves as the basis for a query interpreter or compiler.

from http://www.w3.org/TR/2001/WD-query-semantics-20010607/

Formal Static and Dynamic Semantics [3]

- We will not pursue the formal aspects of the XQuery Core type system.
- We will focus on the way XQuery Core expressions can be written as a kind of comprehension called a monad.
- We will not pursue the theoretical aspects of monads.

Formal Static and Dynamic Semantics [4]

- We will look into algebraic laws that allow us to map from one expression to an equivalent expression.
- This allows us to envisage how an optimizer for XQuery can make use of such laws to derive heuristically more efficient-to-evaluate expressions from less efficient ones
- We will also look into how an evaluator for XQuery would be able to make use of such laws to derive the value of an expression.

The Four **C**s of XQuery Core

- **Closure**
  - Every input is an XML node sequence and every output is an XML node sequence
- **Compositionality**
  - Expressions are composed of expressions whose evaluation have no side-effects.
- **Correctness**
  - There are both dynamic (at query evaluation time) and static (query compilation time) formal semantics.
- **Completeness**
  - XQuery syntax can be completely expressed.

from XQuery Tutorial. Peter Fankhauser, Philip Wadler. 2001.

Translations

XPath to XQuery FLWOR

Translating XPath Projection into XQuery FLWOR

```
let $root := doc("books.xml")
return
  empty
  (
    ( $root/BOOKS/BOOK/AUTHOR )
    except
    ( for $dot1 in $root/BOOKS return
        for $dot2 in $dot1/BOOK return
          $dot2/AUTHOR
    )
  )

(:
=

true
:)
```

- In this and the following slides, we aim to state the equivalences in a form one could test in, say, an XQuery engine like BaseX.

- Note that use of comments to suggest what the outcome of the evaluation is.

- For equivalence of expressions e and *e′* that return sequences, these slides use the form **`empty(`***e* **`except`** *e′***`)`** that, when evaluated, returns *true*.

from XQuery Tutorial. Peter Fankhauser, Philip Wadler. 2001.

Associativity in XPath

```
let $root := doc("books.xml")
return
  empty
  (
    ( $root/BOOKS/(BOOK/AUTHOR) )
    except
    ( ($root/BOOKS/BOOK)/AUTHOR )
  )
(:
=

true
:)
```

Associativity in XQuery

```
let $root := doc("books.xml")
return
  empty
  (
    ( for $dot1 in $root/BOOKS
      return
        for $dot2 in $dot1/BOOK
        return
          $dot2/AUTHOR
    )
    except
    ( for $dot2 in
        ( for $dot1 in $root/BOOKS
          return
            $dot1/BOOK
        )
      return
        $dot2/AUTHOR
    )
  )

(:
=

true
:)
```

Translating XPath Selection into XQuery FLWOR

```
let $root := doc("books.xml")
return
  empty
  (
    ( $root/BOOKS/BOOK[@YEAR < 2000]/TITLE
    )
    except
    ( for    $book in $root/BOOKS/BOOK
      where  $book/@YEAR < 2000
      return $book/TITLE
    )
  )

(:
=

true
:)
```

The University of Manchester

Translations

XQuery FLWOR into XQuery Core

Translating XQuery FLWOR into XQuery Core [1]

```
let $root := doc("books.xml")
return
not
  ( empty
      ( $root/BOOKS/BOOK[@YEAR < 2000]
      )
  )
=
  (      some $year in $root/BOOKS/BOOK/@YEAR
    satisfies $year < 2000
  )

(:
=

true
:)
```

- The translation of XQuery FLWOR expressions into XQuery Core has three main stages:
  - Firstly, a comparison is mapped to an existential quantifier.
  - Secondly, an existential quantifier is mapped to an iteration with `where`-based selection.
  - Thirdly, a `where`-based selection is mapped to a conditional expression.
- This and the next slides exemplify each of the above stages.

Translating XQuery FLWOR into XQuery Core [2]

```
let $root := doc("books.xml")
return
  (      some $year in $root/BOOKS/BOOK/@YEAR
    satisfies $year < 2000
  )
=
  not
    ( empty
        ( for    $year in $root/BOOKS/BOOK/@YEAR
          where  $year < 2000
          return $year
        )
    )

(:
=

true
:)
```

Translating XQuery FLWOR into XQuery Core [3]

```
let $root := doc("books.xml")
return
  empty
  (
    ( for    $year in $root/BOOKS/BOOK/@YEAR
      where  $year < 2000
      return $year
    )
    except
    (   for $year in $root/BOOKS/BOOK/@YEAR
        return
          if   ($year < 2000)
          then $year
          else ()
    )
  )
```

Translating XQuery FLWOR into XQuery Core [4]

```
let $root := doc("books.xml")
return
  empty
  (
    ( $root/BOOKS/BOOK[@YEAR < 2000]/TITLE
    )
    except
    ( for $book in $root/BOOKS/BOOK
      return
        if ( not
               ( empty
                   ( for $year in $book/@YEAR
                     return if   ($year < 2000)
                            then $year
                            else ()
                   )
               )
           )
        then $book/TITLE
        else ()
    )
  )

(:
=

true
:)
```

An example of the complete process

Translating XML Construction into XQuery Core

```
let $root := doc("books.xml")
return
  for $x in ( for   $book in $root/BOOKS/BOOK
              where $book/@YEAR < 2000
              return
                <BOOK>
                { $book/@YEAR,
                  $book/TITLE
                }
                </BOOK>
            )
  return
    some $y in ( for   $book in $root/BOOKS/BOOK
                 where $book/@YEAR < 2000
                 return
                   element BOOK
                     { attribute YEAR
                         { data($book/@YEAR)
                         },
                       $book/TITLE
                     }
               )
    satisfies deep-equal($x,$y)

(:
=

true … true

:)
```

- Note that in this slide, we cannot use the form **`empty(`***e* **`except`** *e′***`)`** to show equivalence of outcome because node identity makes the elements returned by the two expressions irremediably distinct.
- The form we use instead is **`for $x in`** *e* **`return some $y in`** *e′* **`satisfies deep-equal($x,$y)`** that, when evaluated, returns one *true* value for each member of e and *e′*.

The Importance of SIDOAED [1]

```
let $doc :=
   <WARNING>
    <P>Do <EM>not </EM>press the button, computer will <EM>explode!</EM>
    </P>
   </WARNING>
return
   ( "1. All children nodes of the root node",
     " ",
     $doc//*,
     " ",
     "2. All strings in sequence order",
     " ",
     for $x in $doc//*
     return $x/text(),
     " ",
     "3. All strings in document order",
     " ",
     for $x in $doc//*/text()
     return $x
   )
```

- We abbreviate the phrase *sorted in document order and eliminated duplicates* as SIDOAED (pronounce it 'sidled' if you wish).
- Consider the document passed as input.
- Assume it to be an HTML fragment, but this in itself is not important: it is textual coherence that is important here.
- The three queries to the left illustrate the importance of SIDOAED. The results are shown in the next slide.

The Importance of SIDOAED [2]

```
(:
=

1. All children nodes of the root node

<P>Do <EM>not </EM>press the button, computer will <EM>explode!</EM>
</P>
<EM>not </EM>
<EM>explode!</EM>

2. All strings in sequence order

Do
press the button, computer will
not
explode!

3. All strings in document order

Do
not
press the button, computer will
explode!

:)
```

- Clearly, if the textual content has not been SIDOAED it could be scrambled, and this could be semantically significant.

A Complete Complex Translation of XQuery into XQuery Core

```
let $books   := doc("books.xml"),
    $reviews := doc("reviews.xml")
return
  for $x in
    ( for   $b in $books/BOOKS/BOOK,
            $r in $reviews/REVIEWS/REVIEW
      where $b/TITLE = $r/TITLE
      return
        <BOOK>
        {
           $b/TITLE,
           $b/AUTHOR,
           $r/REVIEW
        }
        </BOOK>
    )
```

```
return some $y in
( for $b in (
    for $dot in $books return
      for $dot in $dot/child::BOOKS return $dot/child::BOOK
  ) return
    for $r in (
      for $dot in $reviews return
        for $dot in $dot/child::REVIEWS return $dot/child::REVIEW
    ) return
      if (
        not(empty(
          for $v1 in (
            for $dot in $b return $dot/child::TITLE
          ) return
            for $v2 in (
              for $dot in $r return $dot/child::TITLE
            ) return
              if ($v1 eq $v2) then $v1 else ()
        ))
      ) then (
        element BOOK {
          for $dot in $b return $dot/child::TITLE ,
          for $dot in $b return $dot/child::AUTHOR ,
          for $dot in $r return $dot/child::REVIEW
        }
      )
      else ()
)
satisfies deep-equal($x,$y)
```

MANCHESTER 1824

The University of Manchester

Equivalence Laws

Equivalence Laws [1]: Empty

```
   for $v in () return e
=  (empty)
   ()
```

strings in *italics* are place-holders

v for variables, *e* for expressions, both possibly subscripted

from XQuery Tutorial. Peter Fankhauser, Philip Wadler. 2001.

Equivalence Laws [2]: Sequence

```
 for $v in (e1 , e2) return e3
= (sequence)
  (for $v in e1 return e3) , (for $v in e2 return e3)
```

from XQuery Tutorial. Peter Fankhauser, Philip Wadler. 2001.

Equivalence Laws [3]: Data

```
  data(element a { d })
= (data)
   d
```

Equivalence Laws [4]: Left Unit

```
  for $v in e return $v
= (left unit)
   e
```

Equivalence Laws [5]: Right Unit

```
 for $v in e1 return e2
= (right unit), if e1 is a singleton
  let $v := e1 return e2
```

Equivalence Laws [6]: Associativity

```
  for $v1 in e1 return (for $v2 in e2 return e3)
= (associative)
  for $v2 in (for $v1 in e1 return e2) return e3
```

from XQuery Tutorial. Peter Fankhauser, Philip Wadler. 2001.

Equivalence Laws [7]: Let

```
 let $v := e1 return e2
= (let), i.e., variable replacement by expression
  e2[e1/$v]
```

an alternative notation is $v -> e_1

Seven Equivalence Laws (Collected)

```
  for $v in () return e
= (empty)
  ()

  for $v in (e1 , e2) return e3
= (sequence)
  (for $v in e1 return e3) , (for $v in e2 return e3)

  data(element a { d })
= (data)
  d
```

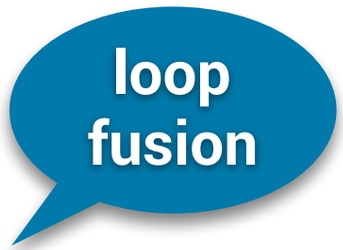
```
  for $v in e return $v
= (left unit)
  e

  for $v in e1 return e2
= (right unit), if e1 is a singleton
  let $v := e1 return e2

  for $v1 in e1 return (for $v2 in e2 return e3)
= (associative)
  for $v2 in (for $v1 in e1 return e2) return e3

  let $v := e1 return e2
= (let), i.e., variable replacement by expression
  e2[$v -> e1]
```

Using the Equivalence Laws for Evaluation

```
  for $x in (<A>1</A>,<A>2</A>) return <B>{data($x)}</B>
= (sequence)
  for $x in <A>1</A> return <B>{data($x)}</B> , for $x in <A>2</A> return <B>{data($x)}</B>
= (right unit)
  let $x := <A>1</A> return <B>{data($x)}</B> , let $x := <A>2</A> return <B>{data($x)}</B>
= (let)
  <B>{data(<A>1</A>)}</B> , <B>{data(<A>2</A>)}</B>
= (data)
  <B>1</B>,<B>2</B>
```

Using the Equivalence Laws for Optimization

```
  let $b := for $x in $a return <B>{ data($x) }</B>
  return for $y in $b return <C>{ data($y) }</C>
= (let)
  for $y in (
      for $x in $a return <B>{ data($x) }</B>
    ) return <C>{ data($y) }</C>
= (associative)
  for $x in $a return
      (for $y in <B>{ data($x) }</B> return <C>{ data($y) }</C>)
= (right unit)
  for $x in $a return <C>{ data(<B>{ data($x) }</B>) }</C>
= (data)
  for $x in $a return <C>{ data($x) }</C>
```



Lecture 13

MANCHESTER 1824

The University of Manchester

BaseX: A Native XML DBMS

XQuery Engines

- There are many XQuery engines.
- For a (not necessarily up-to-date) reference list, see:

  http://www.w3.org/XML/Query/#implementations
- The diversity of approaches and strategies amongst XQuery engines is quite large.
- The slides that follow focus on BaseX:

  http://basex.org/
- It is described in (Grün, 2010).
- The focus is on storage and querying strategies, principally on the latter.

The BaseX XQuery Engine

- When optimizing queries, BaseX does not directly target XQuery Core.
- It takes a more pragmatical approach of relying on indexing and rewrites so as to make the most of the indices it builds.
- It is also noteworthy for adopting techniques more often used for optimizing general-purpose programming, rather than query, languages.
- This approach is based on the argument that XQuery has features of both query and general-purpose programming languages.

MANCHESTER 1824

The University of Manchester

BaseX: A Native XML DBMS

Storage

The BaseX Approach to Storage

- The next few slides describe an encoding of XML data that is elegant and lends itself to efficient implementation.
- It, as is required, preserves document order and can be constructed in linear time.
- It makes it simple to compute, given a context node, the membership of the various axes in location paths.
- This is crucial, of course, for XPath selection.
- Besides this encoding, the slides briefly look at index structures.

MANCHESTER 1824

The University of Manchester

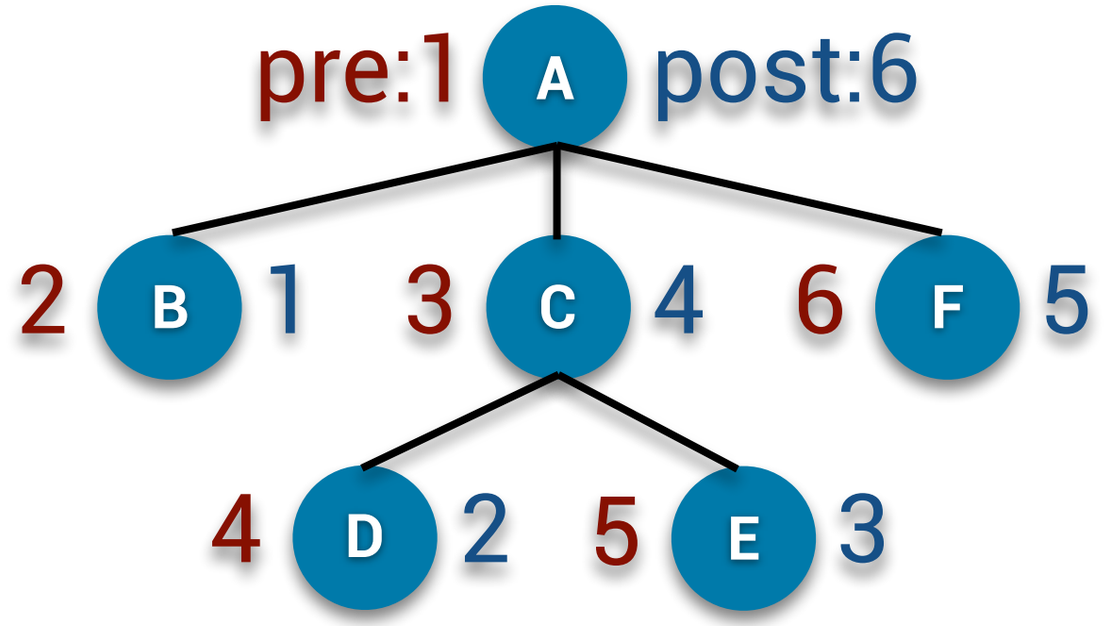
Storage: XML Encoding

An XML Encoding [1]

XML document in textual form

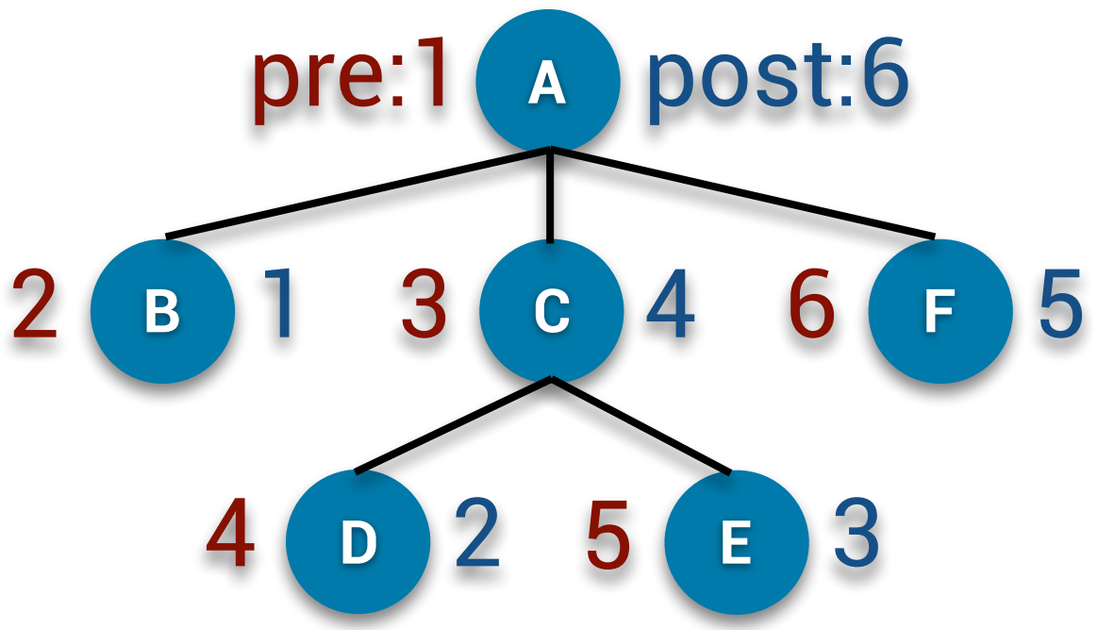
```
<A>
   <B/>
   <C>
      <D/><E/>
   </C>
   <F/>
</A>
```

XML node tree, with pre- and post-order traversals

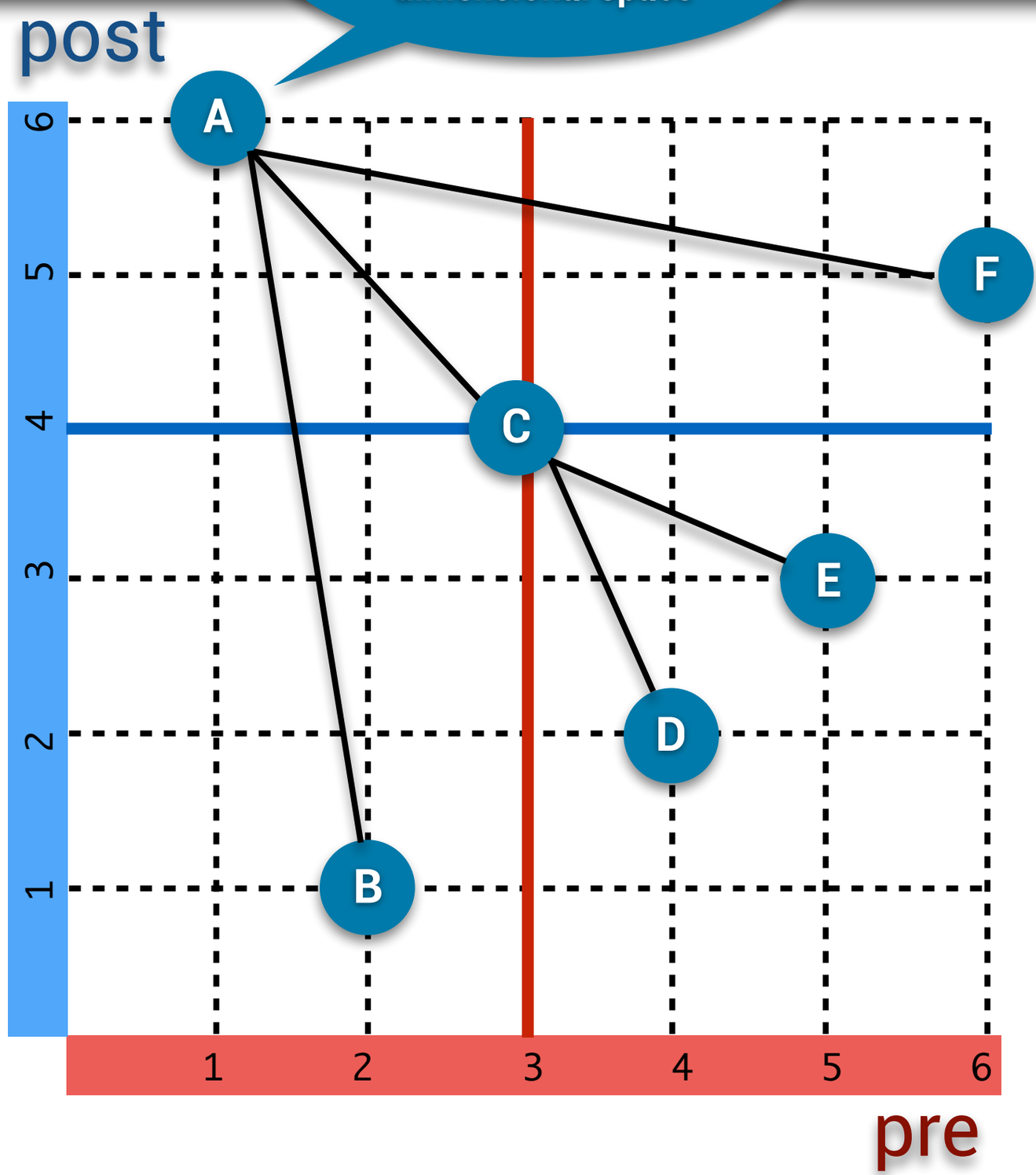


An XML Encoding [2]

XML node tree, with pre- and post-order traversals



The pre- and post-order traversals mapped to a two-dimensional space



Computing the Axes from the Encoding [1]

- Imagine the origin of the graph moving to lie on the coordinates of a given node *n* (e.g., C in the running example).
- This gives rise to quadrant regions, relative to *n*.
- Let (with the obvious interpretation) the quadrants be referred to as:
  - northwest (NW)
  - northeast (NE)
  - southeast (SE) and
  - southwest (SW)

Computing the Axes from the Encoding [2]

- Then it follows that:
  - the ancestors of *n* are in the NW quadrant (e.g., A),
  - the descendants of *n* are in the SE quadrant (e.g., DE),
  - the following nodes from *n* (excluding descendants) are in the NE quadrant (e.g., F),
  - the preceding nodes from *n* (excluding ancestors) are in the SW quadrant (e.g., B).

Computing the Axes from the Encoding [3]

- The contents of each quadrant can therefore be computed from the *(pre, pos)* coordinates of *n* as follows:

1. *ancestor(n) = NW(n) =*
   { n' | pre(n') < pre(n) ∧ post(n') > post(n) }
2. *descendant(n) = SE(n) =*
   { n' | pre(n') > pre(n) ∧ post(n') < post(n) }
3. *following(n) = NE(n) =*
   { n' | pre(n') > pre(n) ∧ post(n') > post(n) }
4. *preceding(n) = SW(n) =*
   { n' | pre(n') < pre(n) ∧ post(n') < post(n) }

Computing the Axes from the Encoding [4]

- Not all relationships between XML nodes can be determined exclusively with *pre* and *post*.
- The *level* is another property that represents the depth of a node within a tree.
- The level of a node is therefore the length of the path from the root to the node.
- It can be used to evaluate four more XPath axes as shown in a forthcoming slide.

Computing the Axes from the Encoding [5]

- Note that the self axis is trivial.
- Note also that the attribute axis is special in that the order of attributes is implementation-specific but poses no distinct challenges.
- Note finally that storing a direct reference to the parent is easy and yields significant benefits in avoiding recomputation with respect to using the level as shown in the next slide.

Computing the Axes from the Encoding [6]

- The parent (resp., a child) *n'* of a node *n* is an ancestor (resp., descendant), whose level is smaller (resp., larger) by one.
- The following (resp., preceding) siblings *n'* of a node *n* are those following (resp., preceding) nodes of that have the same parent node (i.e., are at the same level).

1. *parent(n) =*
   { n' ∈ ancestor(n) | level(n') = level(n) − 1 }
2. *children(n) =*
   { n' ∈ descendant(n) | level(n') = level(n) + 1 }
3. *following-sibling(n) =*
   { n' ∈ following(n) | parent(n) = parent(n') }
4. *preceding-sibling(n) =*
   { n' ∈ preceding(n) | parent(n) = parent(n') }

Computing the Axes from the Encoding [7]

- Although this encoding is fast to compute, it is brittle with respect to updates, which may cause complete recomputation of the encoding.
- To mitigate this (by delaying its impact), one might leave gaps in the encoding (i.e., empty positions/cells), assuming one can compute the number of descendants (referred to as the *size*) from existing properties of the encoding.

Computing the Axes from the Encoding [8]

- Indeed, the size of a node can be computed from its pre/pos coordinates and its level as follows:

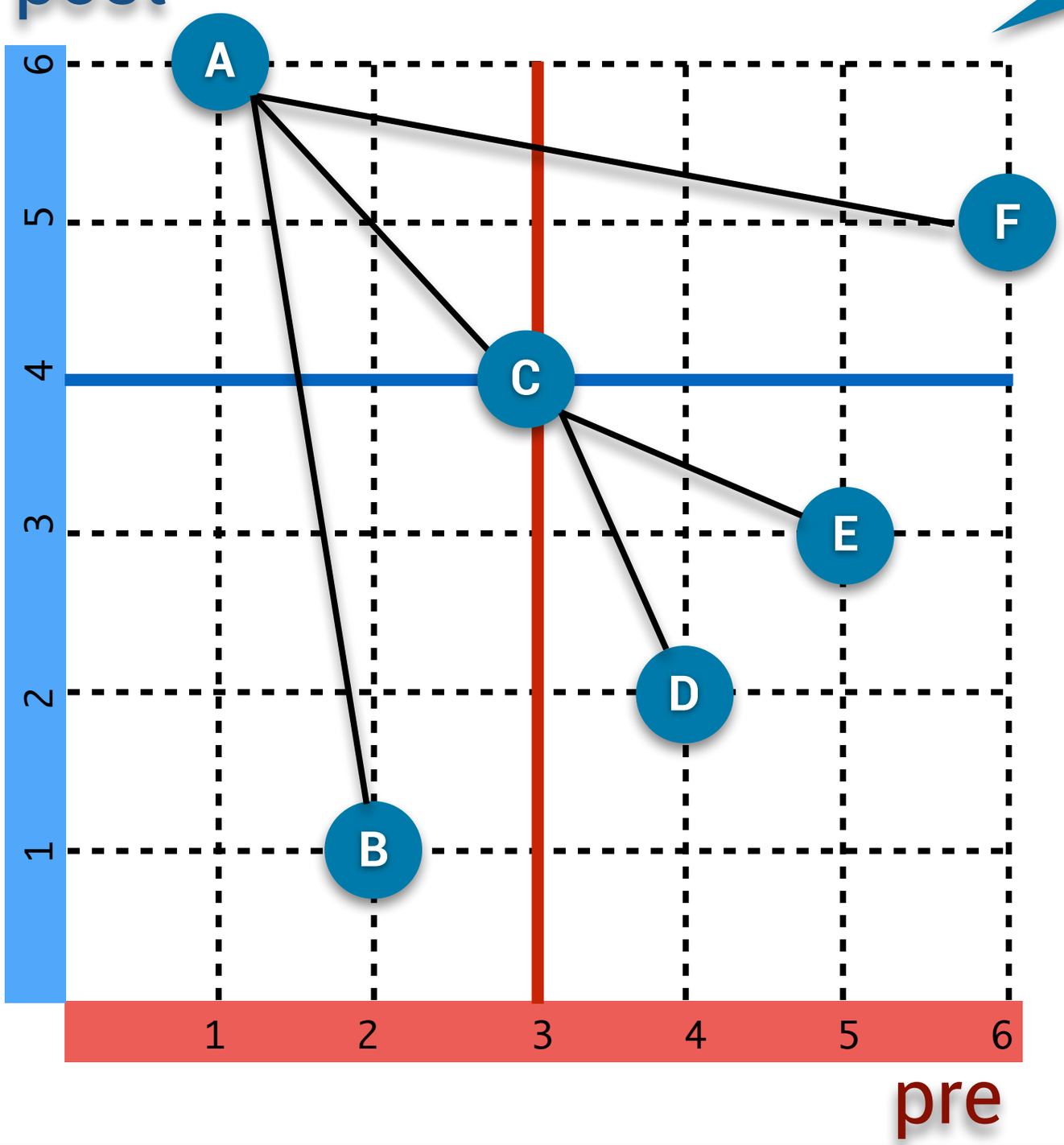
size(n) = post(n) − pre(n) + level(n)

- Note that if one stores the pre, post and level values of a node, one can reconstruct the original document in order whilst making it potentially efficient to compute the XPath axes.

- One way of achieving this is to map the information generated by the encoding onto a relation *E*.

An XML Encoding [3]

The pre- and post-order traversals mapped to a two-dimensional space



We can map the two-dimensional space onto a relation and add more dimensions (i.e., level, values, etc.)

E					
node	pre	post	level	…	V
A	1	6	0	…	()
B	2	1	1	…	()
C	3	4	1	…	()
D	4	2	2	…	()
E	5	3	2	…	()
F	6	5	1	…	()

Values can be inlined, possibly in compressed form, or one may place a pointer to a (possibly compressed) value table.

from Storing and Querying Large XML Instances. Christian Grün

Using the Relational View of the Encoding

ancestor("C")

==

$\pi_{e'.node}(E \text{ as } e \bowtie_{\sigma\ e.node="C" \wedge e'.pre < e.pre \wedge e'.post > e.post} E \text{ as } e')$

==

```
SELECT e2.node
  FROM e AS e1, e AS e2
 WHERE e1.node = "C"
   AND e2.pre  < e1.pre
   AND e2.post > e1.post;
```

=

{A}

E					
node	pre	post	level	…	V
A	1	6	0	…	()
B	2	1	1	…	()
C	3	4	1	…	()
D	4	2	2	…	()
E	5	3	2	…	()
F	6	5	1	…	()

An XML Encoding [4]

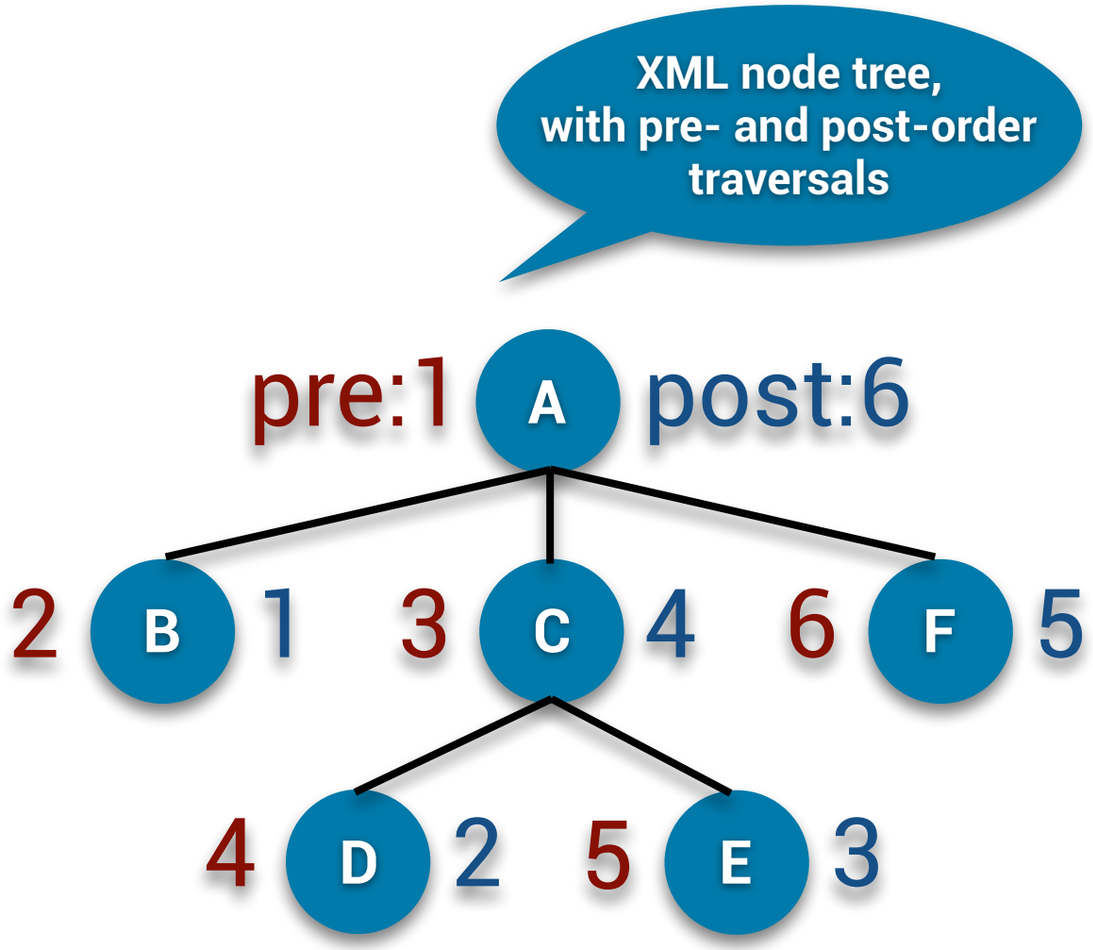
- In fact, for performance reasons, XML query engines that use this kind of encoding tend not to store the post value.
- In MonetDB, XML nodes are mapped to a *pre/size/level* triple.
- BaseX uses *pre* and *size* but *dist* rather than *level*, where *dist* is the distance between the position of the node's parent and the position of the node in the pre-order traversal, i.e.,

dist(n) = pre(parent(n))-pre(n)

An XML Encoding [5]

- The *dist* attribute allows access to the parents and ancestors of a node in constant time.
- Using relative distance also has the benefit that it has update invariant properties.
- For example, it does not need to be recomputed if a subtree is moved from one place to another either in the same or in another document.
- The next slide shows the BaseX encoding for the running example.

An XML Encoding [6]



The BaseX pre/dist/size of the XML node tree.

E					
node	pre	dist	size	…	V
A	1	0	5	…	()
B	2	1	0	…	()
C	3	2	2	…	()
D	4	1	0	…	()
E	5	1	0	…	()
F	6	5	0	…	()

MANCHESTER 1824

The University of Manchester

Storage: Index Structures

Index Structures [1]

- In relational databases, most of the concern is to index values.
- For XML data, besides values, there is also a concern with tags that identify elements and attributes and with the tree structure of the data.
- In BaseX, the three kinds of indices are available:
  - Value indices
  - Name indices
  - Path summaries

from Storing and Querying Large XML Instances. Christian Grün

Index Structures [2]

- A value index is created for every text node.
- Several size reduction and compression strategies are used.
- For example, strings above a certain length are not indexed on the grounds that they are less frequently occurring in queries.

Index Structures [3]

- For another example, the position of a node in the pre-order traversal (i.e., its *pre* value) is compressed.
- The resulting index is conservative in terms of memory footprint, but update operations are not supported.
- In the absence of updates this is not a major shortcoming, whilst the performance benefit of holding the index in memory can be significant.

Index Structures [4]

- XML data is thought of as very verbose.
- This has motivated alternative representations, such as JavaScript Object Notation (JSON), that aim to be more concise.
- The main reason for XML verbosity is the extreme repetition of tag and attribute names.

from Storing and Querying Large XML Instances. Christian Grün

Index Structures [5]

- One approach to mitigate the problem is to replace variable-length names with numeric identifiers in the process of storing them in an associative array structure.
- BaseX takes advantage of name indices to store statistical information that the optimizer may need in choosing execution plans.
- Name indexes can be updated but the freshness of statistics deteriorates over time with updates.

Index Structures [6]

- A name index collates information on tag and attribute names irrespective of where they occur in the document.
- A path summary complements a name index by collating distinct element and attribute paths that occur in the document.
- A path summary can be computed statically from a schema (either a DTD or an XML Schema specification) if it is available for a document.
- Computing a path summary dynamically is, of course, more expensive.

Index Structures [7]

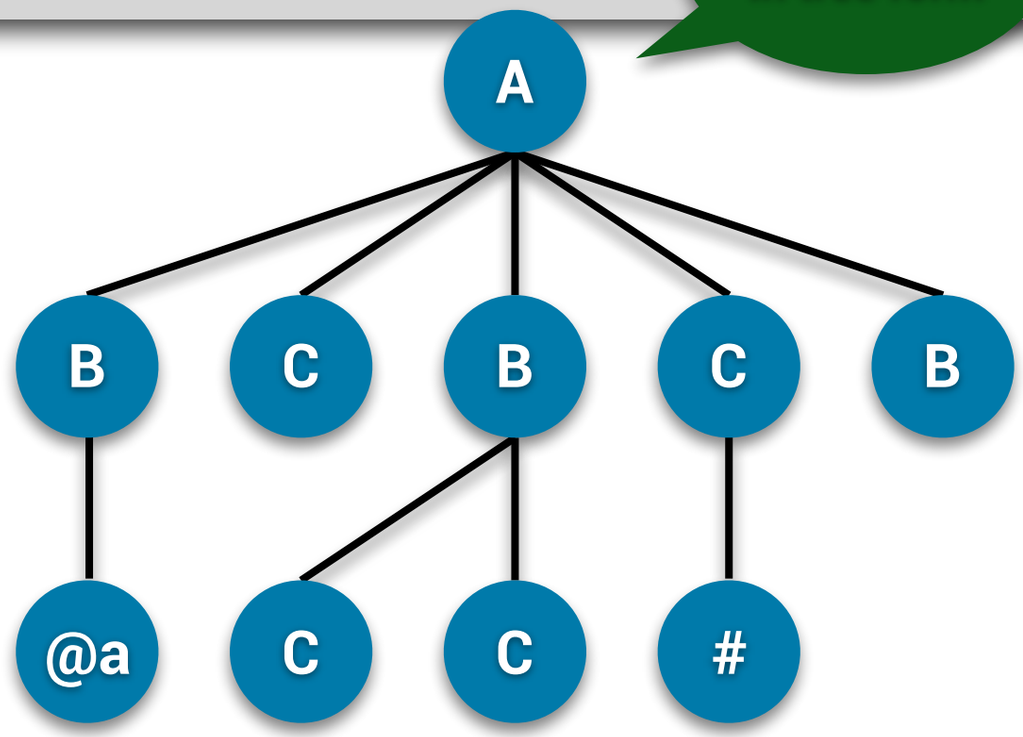
- However:
  - Path summaries computed from documents are more accurate, as they reflect only actually occurring paths.
  - The construction overhead is small if the path summary is built when the document is being scanned for storage.
- The next slide shows an example of a path summary for a simple XML document.

Index Structures [8]

XML document in textual form

```
<A>
   <B a="0"/>
   <C/>
   <B><C/><C/></B>
   <C>text</C>
   <B/>
</A>
```

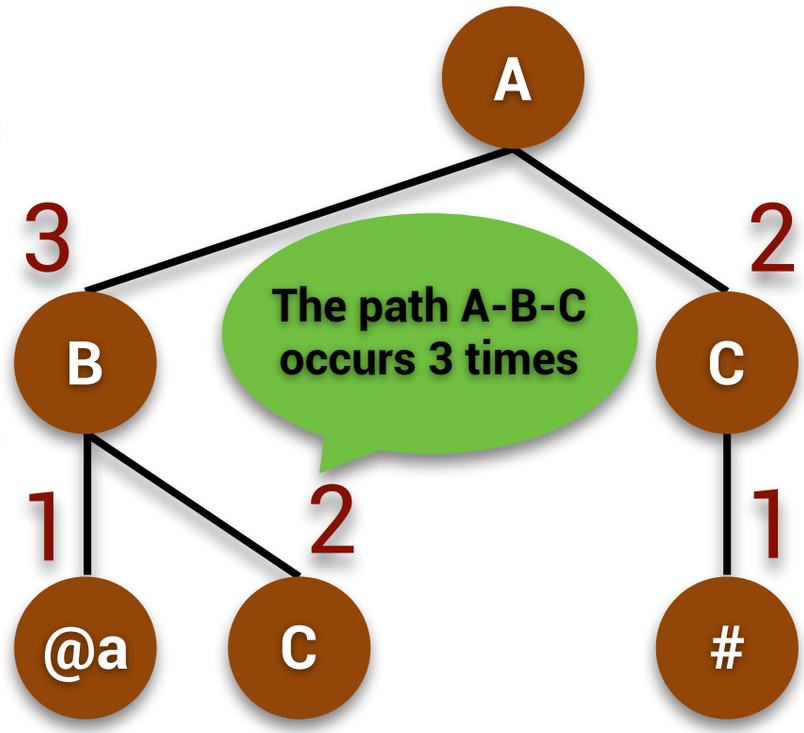
XML document in tree form



Path Summary with cardinalities

The path A-B occurs 3 times

The path A-B-C occurs 3 times



Lecture 14

MANCHESTER 1824

The University of Manchester

BaseX: A Native XML DBMS

Query Optimization

XQuery Non-Classical Capabilities [1]

- Classical query languages, like relational algebras and relational calculi, tend to limit themselves to declaratively specifying content retrieval.

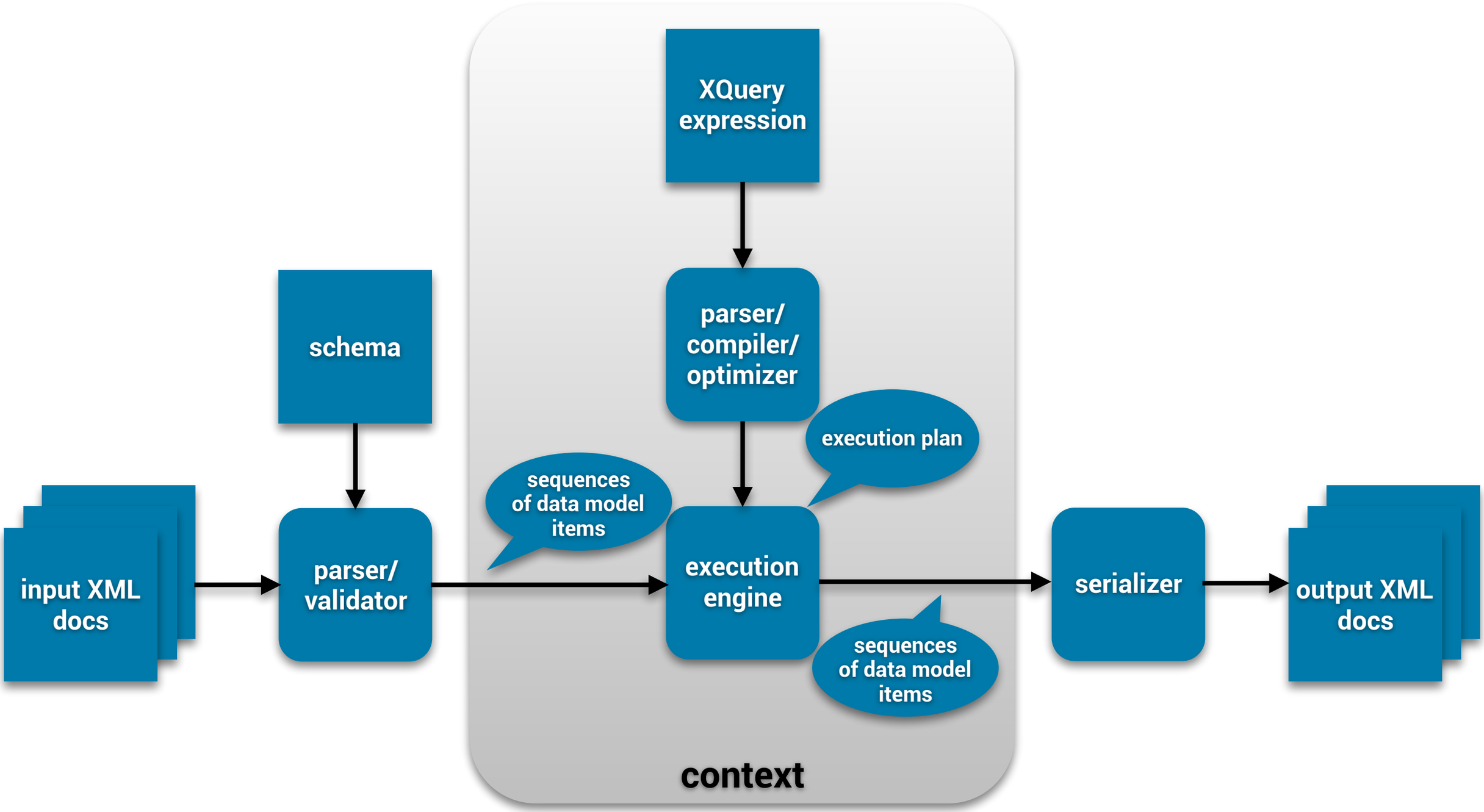
- For practical purposes, more is often needed.

- XQuery, like SQL, is more versatile than classical query languages.

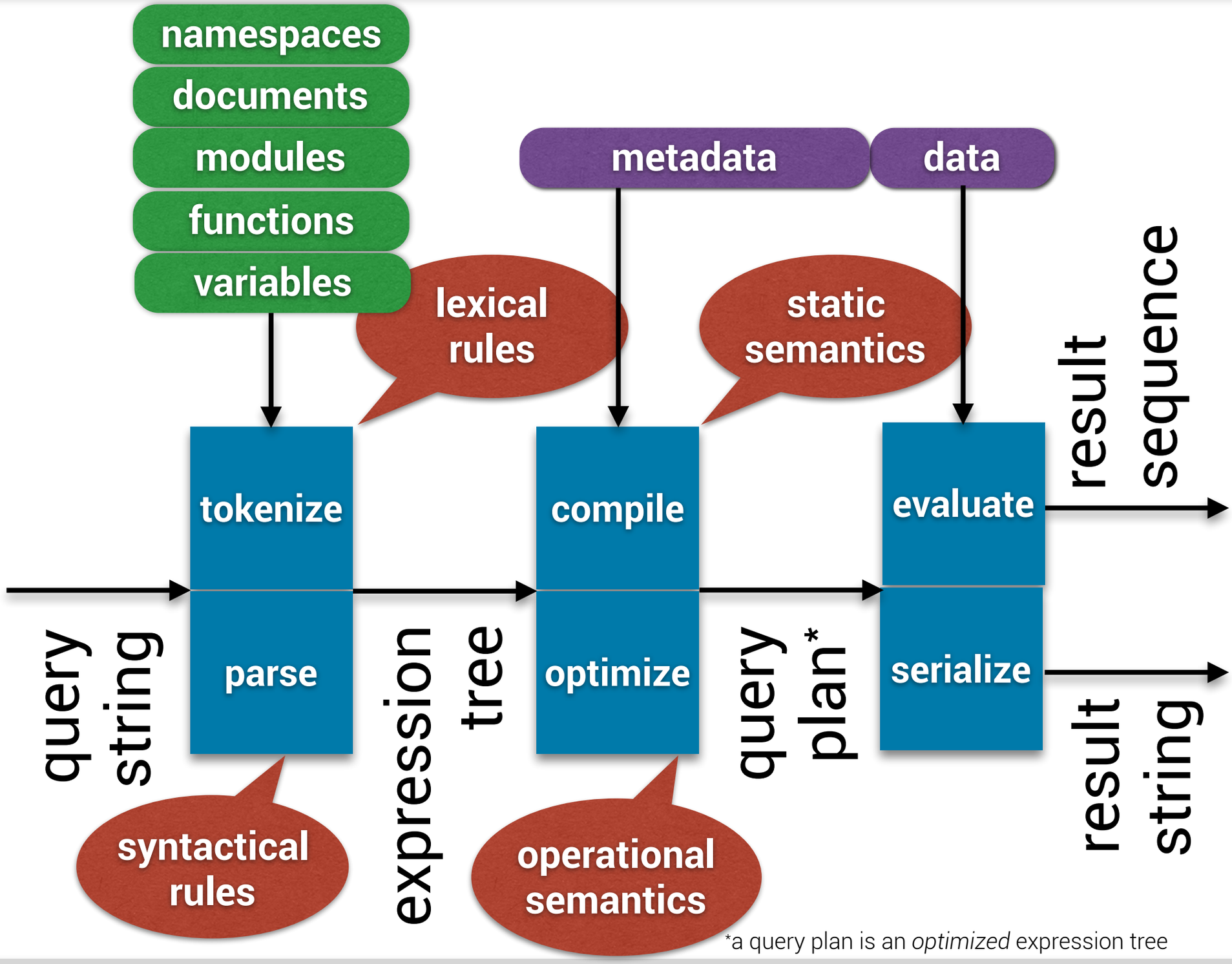
XQuery Non-Classical Capabilities [2]

- Non-classical capabilities of XQuery include:
  - New content, as XML fragments, can be created.
  - New documents can be created.
  - Modules collecting functions (that can be passed as parameters and include recursive ones) can enrich a query.
- These capabilities make XQuery optimization more challenging than that of classical query languages.

A Simplified View of the XQuery Processing Model



A Simplified View of the Query Engine Architecture



*a query plan is an *optimized* expression tree

An Overview of the Compilation/Optimization Process [1]

- BaseX does tokenization and parsing together on the grounds that the lexical components of XQuery are hard to disambiguate independently from the syntactical context.

- BaseX also does compilation and (static) optimization together as it does not use the classical separation between logical and physical optimization (based, respectively, on rewriting and cost-based plan selection).

- Finally, BaseX does not enforce a strict separation of evaluation and serialization if/when using pipelined evaluation.

An Overview of the Compilation/Optimization Process [2]

- The BaseX compilation and optimization process consists of:
  - type checking the expression tree
  - applying semantics-preserving transformations on the expression tree that yield a query plan that is, heuristically, more efficient to evaluate
- Among the more significant transformations that the example query engine applies are:
  - Pre-evaluation of operations with statically-computable values
  - Rewriting of location paths and FLWOR expressions
  - Rewriting of predicates to rely on indices

An Overview of the Compilation/Optimization Process [3]

- The classical separation between logical and physical optimization has not been followed very often in the XML/XQuery context.
- There have been cases in which the classical separation has been pursued, e.g., in the query engines developed for:
  - the Lore (http://ilpubs.stanford.edu:8090/view/projects/lore/) project, and
  - the TIMBER (http://wwweb.eecs.umich.edu/db/timber/) project.

An Overview of the Compilation/Optimization Process [4]

- Most recent implementations have tended to not enforce the distinction and to adopt a programming-language as opposed to database-language view of the XQuery compilation/ optimization process.
- This is consistent with the fact that XQuery has non-classical capabilities that make it, in certain cases, more like a general purpose programming language than SQL (without procedural extensions) does.

from Storing and Querying Large XML Instances. Christian Grün

An Overview of the Compilation/Optimization Process [5]

- It is also the case that XQuery optimization tends to be more implementation-specific than for classical query languages.
- Two examples of this are:
  - Reverse axis steps are normally expensive if the processing model is a streaming (or push-based, or event- driven model) one, as with SAX, but not if a reference to the parent is available with the node.
  - Fragment construction is normally expensive but not as much if most representations can be held in primary memory.
- BaseX does use a direct reference to node parents and aims to remain in primary memory in most cases.

Static Optimizations in BaseX

Constant Folding

- Many sub-expressions in a query are values, or have values as arguments.
- As such, they will always evaluate to the same result and can be pre-evaluated once at compile time.
- This process is known as *constant folding*.
- For maximum benefit, constant folding needs to be recursively evaluated on all expressions of the query tree.
- It can also be applied to certain functions (e.g., `doc()` and `collection()`)

Constant Propagation

- *Constant propagation* is related to folding: whenever a variable turns out to have a constant value, it can be statically bound to all its references in the query.
- Recognition of static variables is a straightforward operation in most functional languages, as all global variables are immutable and thus constant.
- Some XQuery expressions, such as value comparisons, return an empty sequence if one of the arguments yields an empty sequence, and implementations are free to choose if the remaining arguments are ignored or evaluated.

Variable/Function Inlining

- Variable expressions can also be bound to their references if they have not yet been evaluated to values.
- In XQuery, variables and functions without arguments are mostly similar, except that variables cannot be recursively called.
- Consequently, functions without function calls and arguments can be treated equally to variables.
- If variable and functions calls are replaced by their declarations (i.e., *inlined*), the resulting code may be subject to further local optimizations.

Dead Code Elimination [1]

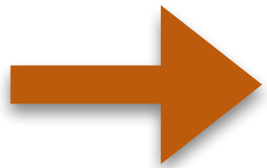
- After the expression tree has been simplified and rewritten, there may remain sub-expressions that will have no effect on the query results, or are not accessed at all.
- Eliminating sub-trees will reduce the tree size and, even more important, reduce evaluation time by avoiding the execution of irrelevant operations.
- As shown, inlining will make variable declarations within FLWOR expressions obsolete, and the subsequent removal of all declarations in questions will speed up execution, as the repeated process of binding values to variables can be skipped.

Dead Code Elimination [2]

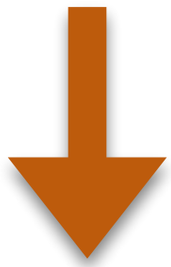
- If no declarations are left, the complete FLWOR expression is replaced by the RETURN expression.
- If WHERE has been specified, the return value is wrapped by a conditional expression.
- In conditional expressions, branches can be discarded if they are known to be never accessed.
- If the main condition yields true, the else branch can be eliminated, and vice versa.
- If both branches yield the same result (which can happen after numerous simplifications of the original code), the condition itself need not be executed.

An Extended Example

```
let $x := 5
let $y := $x - 2
return
  if ($x < 1 + 2 * 3)
  then $x + $y
  else $x - $y
```



- pre-evaluating (2 * 3)
- pre-evaluating (1 + 6)
- rewriting ($x_0 < 7)
- inlining $x_0
- pre-evaluating (5 - 2)
- pre-evaluating 5 < 7.0
- pre-evaluating if(true()) then (5 + $y_1) else (5 - $y_1)
- inlining $y_1
- pre-evaluating (5 + 3)
- simplifying flwor expression



8

Static Typing [1]

- To reduce the number of runtime errors, type checking can be used at compile time to validate if an operation will always be unsuccessful.
- If both types of the arithmetic operators are numeric, they can always be added, subtracted, etc., whereas values of type xs:string and xs:integer will always result in a type error.
- Also, as such operators return an empty sequence if one of the operands is empty, the expression can be pre-evaluated and substituted by an empty sequence.
- If values contain more than one item, the common super type is stored as a sequence type.
- As a result, the exact types of single items need to be inferred at runtime.

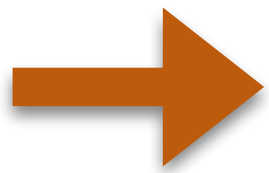
1+"5" → Number expected, xs:string found: "5".

1+() → ()

Static Typing [2]

- The existence of static types is beneficial in a number of other cases: path expressions that are composed of certain types will never yield results and need not be evaluated at all.
- As an example, the query below will always return an empty sequence, as attributes cannot have child nodes.
- Also, the root node of a path expression must always yield nodes to be correctly evaluated.

```
//@*/node()
```



```
()
```

Location Path Rewriting [1]

- As a supplement to static typing, and a surrogate for schema information, a path summary facilitates the validation of location paths.

- If the root of a path expression points to an existing database, or if the root context can be evaluated at compile time, the path summary of the referenced document can be used to match the document paths against the query.

- If a query path is not found in the summary, it can be eliminated and replaced by an empty result.

from Storing and Querying Large XML Instances. Christian Grün

Location Path Rewriting [2]

- Path summaries can be applied to optimize a path expression.
- In most implementations, the descendant step is expensive, as all descendants of the root node need to be touched, either recursively or in a linear run.
- If the expression is replaced by multiple child steps, the number of touched nodes is reduced to the distinct node paths.

Location Path Rewriting [3]

- The different cases in which a path expression of the form //ELEM can be optimized are:

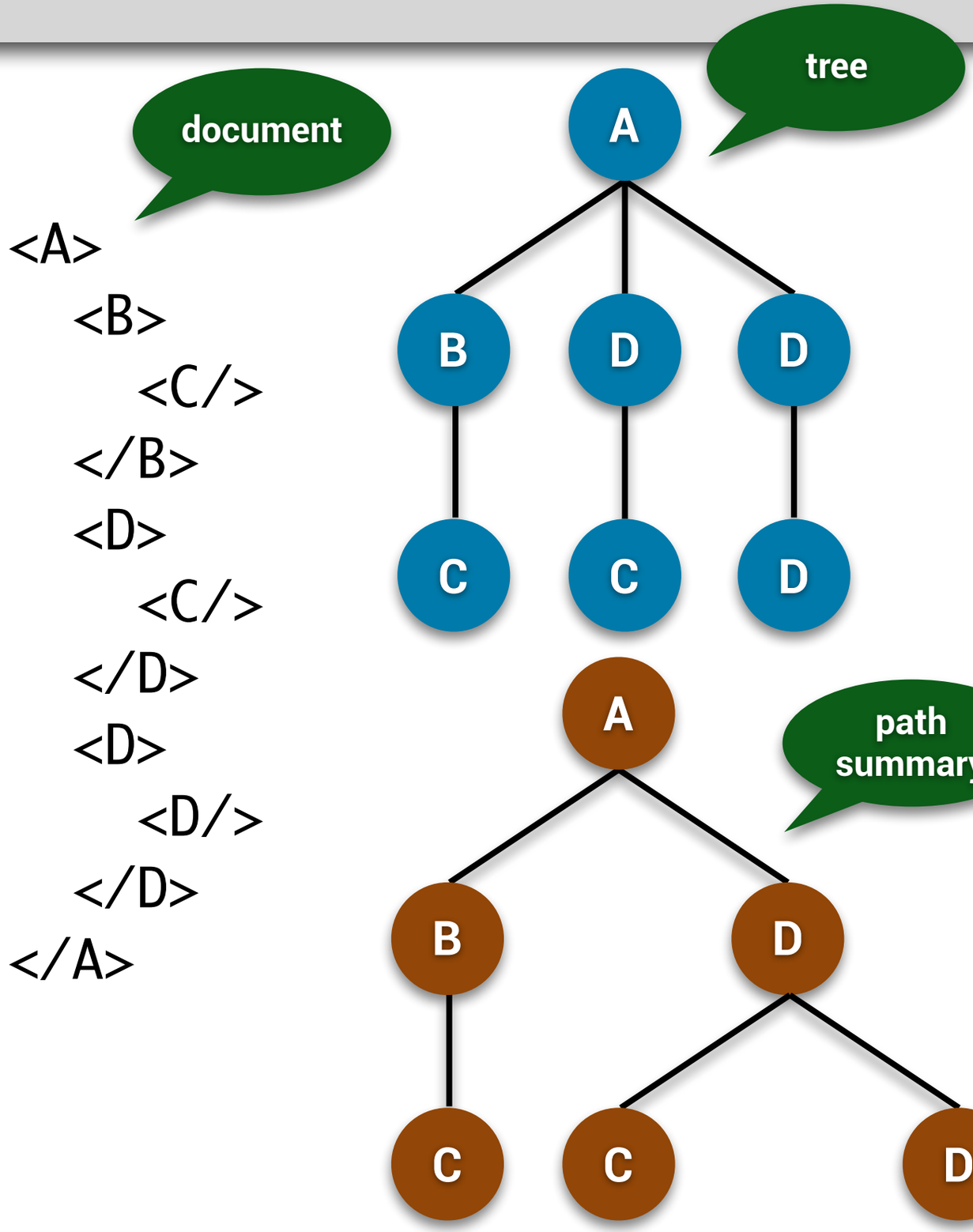
1. If the summary contains one single node with the element name ELEM, the path to this node is transformed into a location path with multiple child steps.
2. If several nodes exist with ELEM as element name, and if all nodes are located on the same level (i.e., if all node paths have the same length), the paths are used as input for a single location path in a level-by-level analysis, as follows.

   If the element names of all paths on a level are equal, a child step with this element name is generated.

   Otherwise, a wildcard * is used as node test to match all elements.
3. If occurrences of ELEM are found on different levels, all distinct paths will be returned and combined with a union expression.
4. If the element name ELEM is unknown, an empty sequence is returned.

- Paths that are refined by a predicate can also be rewritten.

An Example Location Path Rewriting



path	optimized
//B	/A/B
//C	/A/*/C
//D	/A/D | /A/D/D
//E	()
//A//C	/A/*/C

Location Path Rewriting [4]

- The two slashes (`//`) notation are an abbreviation of `/descendant-or-self::node()/`.
- This construct is very expensive, as all nodes of a document are selected and, in most cases, reduced to a much smaller number of results in the subsequent location step.
- A simple, but very efficient optimization is to rewrite a descendant-or-self and child step to a single descendant step.

Location Path Rewriting [5]

- In the box to the right, for each query on the left-hand side (LHS) we indicate whether it can (or cannot) be rewritten into the one on the right-hand side (RHS), where the latter is much cheaper to evaluate, as the intermediate materialization of all nodes is avoided.
- The first example shows that an expression with predicates can be rewritten as well, as long as they are not positional.
- Positions always refer to the last location step, so in the second example the LHS cannot be rewritten into the RHS shown as the former will select each first child element of all addressed nodes, whereas the latter will only return the first descendant element of the document.
- However, if the location path is enclosed by parentheses, the query can safely be rewritten as shown in the third example.
- The fourth example shows that for a descendant-or-self plus attribute step, since only element nodes can have attributes, the node() test of the first step is rewritten to an element test (*).

```
  //x
≡
  /descendant-or-self::node()/child::x
```

Example 1

```
  /descendant::x

   //x[1]
≠
  /descendant::x[1]
```

Example 2

```
  (//x)[1]
≡
  (/descendant::x)[1]
```

Example 3

```
  //@x
≡
  /descendant-or-self::*/attribute::x
```

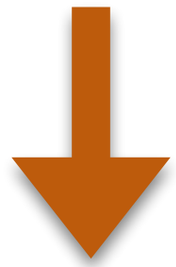
Example 4

FLWOR Expressions [1]

- In XQuery, many queries can be equivalently written as XPath location paths or as XQuery iterative FLWOR expressions.
- XQuery Core prefers the latter and, earlier in this course, we have shown how to map location paths into FLWOR expressions.
- BaseX prefers the opposite, i.e., it aims to map FLWOR expressions into location paths when possible.

FLWOR Expressions [2]

```
for    $item in doc("books.xml")/descendant::BOOK
where  $item/AUTHOR = 'Stevens'
return $item
```



```
doc("books.xml")/descendant::BOOK[AUTHOR = 'Stevens']
```

FLWOR Expressions [3]

- However, some FLWOR queries cannot be expressed in XPath.
- For example, no XPath equivalent exists for the ORDER clause, which sorts iterated values.
- Also, XQuery is needed if after being retrieved items are processed one by one before being placed in the result, e.g.,

```
for $n in 1 to 10
return $n * 2
```

FLWOR Expressions [4]

- To enforce uniformity, one would need to choose a common representation and normalize both variants to it.
- XPath is the most appropriate representation in BaseX because predicate tests are already the focus for index rewritings.
- In BaseX, if possible, FLWOR queries are normalized by rewriting the optional WHERE clause to one or more predicates that are added to the expressions defined by the variable declarations.
- When combined with variable inlining, it may lead to the removal of the FLWOR expression altogether.

FLWOR Expressions [5]

- Before the clause can be rewritten, two preconditions must be met:
  1. The FOR clauses must not specify a positional or a full-text scoring variable.
  2. A recursive algorithm checks if all occurrences of the variables that are introduced in the FOR clause can be removed from the WHERE expression and replaced with a context-item expression (.).
- Substitution is invalid whenever the new context item reference conflicts with a change of the context item at runtime.
- This is the case, e.g., if the variable in question is enclosed in a deeper predicate, or occurs in the middle of a path expression.
- The mapping does not attach predicates to inner LET clauses as the equivalence may not hold, e.g.,

```
for $a in 1
let $b := 2
where $b = 3
return $a
```

≠

```
for $a in 1
let $b := 2[. = 3]
return $a
```

Index-Related Optimizations [1]

- Many location paths contain predicates with comparison operators to match XML elements or attributes with specific contents.
- The expression `doc("books.xml")//BOOK[@YEAR = 2000]` is an example of a query with a comparison operator.
- It returns all `BOOK` elements in `books.xml` whose `YEAR` attribute has value `2000`.
- Value indexes can be used to speed up queries of this kind, if meta data and the statistics of a database indicate that an index access is possible and expected to be cheaper than scan-based query evaluation.
- There is evidence that, in BaseX, such optimizations contribute the most to good performance.

Index-Related Optimizations [2]

- The tasks the optimizer must perform are:

  1. It must verify that the path expression refers to exactly one database, or documents in a database.
  2. It must consider every predicate in every step of every path expression as candidates for index-based access and choose one or more for index-rewriting.
  3. It must index-rewrite path expressions so that the index access is performed first, followed by the evaluation of the inverted location path.

Index-Related Optimizations [3]

- In comparison to classical query optimization, the second step in the previous slide is similar in intention to the aim of rewriting selections.
- The first step does not normally apply in the classical case as tables are collected in a single database of which indices become part.
- The third step also does not apply in the classical case, since data is limited to flat tables and indices uniquely refer to specific columns of these tables.
- So, while XQuery is more versatile than SQL in that queries are not bound to a single database context, there are drawbacks, as a query that uses various documents and paths as input does not benefit as much from index-rewriting.

Index-Related Optimizations [4]

- The verification of unique reference requires determining the reference of every step and establishing that only one database is referenced.

unique reference: valid rewriting

```
for $x in (doc("amazon.xml")/BOOKS/BOOK/AUTHOR
          )
where $x = "Stevens"
return $x/text()
```

- For example, the example at the top to the right refers to a single document and can be index-rewritten, whereas the one at the bottom does not, and cannot.

non-unique reference: invalid rewriting

```
for $x in (doc("amazon.xml")/BOOKS/BOOK/AUTHOR,
           doc("fatbrain.xml")/BOOKS/BOOK/AUTHOR
          )
where $x = "Stevens"
return $x/text()
```

Index-Related Optimizations [5]

- Expressions that are potential candidates for index-based access (e.g., both equalities, inequalities as well as Boolean expressions that combine them) can occur in different steps of a path.
- This means that there are multiple opportunities for index-rewriting.
- In practice, most of the time most of the benefits are reaped with one single index-rewritten predicate.

Index-Related Optimizations [6]

- In BaseX, costs are defined to be the exact or estimated number of returned index results.
- If the cost is zero, the index is assumed to return no results at all, and the whole path expression can be replaced with an empty sequence at compile time.
- If the cost exceeds a certain maximum value (e.g., the number of text nodes in the database) index-rewriting is skipped in favour of standard query execution.

Index-Related Optimizations [7]

- The rewriting is carried out from right to left in the path, and thus has the effect of inverting it.
- The axis of each step is inverted and combined with the node test of its left-hand step.
- In the examples that follow in the next slides, let ***TI(e)*** (resp., ***AI(e)***) denote a function call that returns all text nodes from the index that match the string (resp., attribute) value of ***e***.

Index-Related Optimizations [8]

- In BaseX, a bottom-up approach is used, so that the index is accessed first, followed by the evaluation of all other predicate tests and axis steps.
- All location steps in the selected predicate and the main path have to be inverted to ensure that the expression yields the correct results.
- This inversion is possible for many paths, as numerous symmetries exist between location paths.
- For example, the following location paths are equivalent:

1. `descendant-or-self::m[child::n] ≡ descendant::n/parent::m`
2. `p[self::n]/parent::m ≡ p/self::n/parent::m`
3. `self::m[child::n] ≡ child::n/parent::m`

- As the three equivalences imply, it is not sufficient to just replace all steps with their inverse counterparts.
- Instead, some steps will be enclosed by a new predicate, and others will be moved out of an existing predicate.

Index-Related Optimizations [9]

```
  /descendant::m[child::text() = e]
≡
  TI(e)/parent::m

  /descendant::m[descendant::text() = e]
≡
  TI(e)/ancestor::m

  /descendant::m[child::n/child::text() = e]
≡
  TI(e)/parent::n/parent::m

  /descendant::m[descendant::n/child::text() = e]
≡
  TI(e)/parent::n/ancestor::m

  /descendant::m[child::n/descendant::text() = e]
≡
  TI(e)/ancestor::n/parent::m
```

example index-rewritings of descendant + predicate queries

Index-Related Optimizations [10]

example index-rewritings of descendant + predicate queries on attributes

```
/descendant::m[attribute::* = e]
≡
AI(e)/parent::m

/descendant::m[attribute::n = e]
≡
AI(e,n)/parent::m
```

See (Grün, 2010) for more equivalences.

Dynamic Optimizations in BaseX

Runtime Optimizations [1]

- All static optimizations, shown in the above slides, have constant or logarithmic costs.
- In most cases, the once-only overhead of performing them is more than compensated by efficiency gains in query evaluation.
- But some properties of the touched data are either not known before the query is evaluated or cannot be used for optimizations.
- Thus, some decisions must be taken at runtime, giving rise to runtime, or just-in-time, optimizations.
- As an example, a sequence might include items of different types, and an expression will need to choose between different evaluation plans at runtime.
- This is one aspect of a broader field of dynamic (or adaptive) query processing.

Runtime Optimizations [2]

- In dynamic query processing, a major challenge is to efficiently and effectively decide which alternative is more likely to result in the lowest evaluation cost.
- It becomes crucial to ensure that the cost of finding the best alternative does not outweigh the cost of evaluating the original, unoptimized query expression.
- In BaseX, two practical runtime optimizations are used that demonstrate how simple runtime tests, backed by appropriate extensions of the evaluation framework, can have a significant positive impact on the execution time:
  - efficient sequence access
  - efficient sequence comparison

Runtime Optimizations [3]

- Consider the XQuery expressions below:

```
declare variable
  $data := doc("books.xml")//BOOK;
for     $i in 1 to count($data)
return  $data[$i]
```

- The bracket notation in the return clause may look like an array operation but it is in fact a positional test on a dynamically computed sequence.

- Hence, if data has *n* elements, naïve evaluation would match the predicate against all items of the sequence *n* times, i.e., quadratic complexity.

- Since only one result is ever returned, one optimization would be to skip the rest after the first match, potentially reducing the complexity to half on the length of the initial segment whose last element is the match.

- If the complete sequence is available at runtime, direct access is possible.

Runtime Optimizations [4]

- Sequence comparisons are among the most frequently used expressions in XQuery.
- Thus, even small optimizations can lead to significant performance gains.
- Recall that they are existentially quantified comparisons, which atomize the items of both its operands and return true if at least one pairwise comparison is successful.
- Sometimes it is possible to statically optimize the comparison.
- In a comparison, one of the sequences must be iterated several times, better performance can be expected if the intermediate results are cached.
- Caching may be suboptimal, however, if only single items are to be compared, which is the most common case.
- Because of this, before caching, BaseX checks whether one or both of the sequences is either empty or a singleton, dealing with such cases more effectively than by caching.

Lecture 15

MANCHESTER 1824

The University of Manchester

BaseX: A Native XML DBMS

Query Evaluation

Query Processing Approaches

- After all normalizations and optimizations have been performed, the resulting expression tree contains all information needed to evaluate the query.
- BaseX uses interpretation, i.e., the expression tree that results from compilation and optimization is evaluated directly.
- In the evaluation step, all data will be handled that cannot be statically made use of.
- This step can often be vacuous (i.e., it simply returns the outcome of the compilation step) if the query is supposed to return statistical information on a database, or if it does not refer to a database at all.

Iterative Processing [1]

- BaseX adopts an iterator-based approach to evaluation.
- The XQuery literature tends to use indistinctly *iterative*, *streaming*, and *pipelined* to qualify one and the same style of query processing.
- An iterator-based approach is an inherently on-demand, as-needed model.
- In the ideal case, iterative processing results in regular CPU and I/O costs, as no intermediate results are generated that end up in much smaller final results.

Iterative Processing [2]

- As an example, the query `(1 to 100*1000*1000)[1]` consists of a filter expression with one predicate.
- It returns the first of 100 million items.
- If the query were executed in what is referred as an intermediate materialization approach, the range expression will first generate a sequence of 100 million integers.
- Then, and only then, all integers will be filtered by the predicate, and only the first one will be accepted as result.
- This makes it clear that queries on large documents could cause out-of-memory errors, even if the final results are very small.
- Iterative evaluation makes lazy evaluation possible, i.e., the computation of values can be delayed, or completely avoided, if they will not contribute to the final result.
- In the example above, only the first item needs to be generated and tested: all the remaining values can be ignored.

Intermediate Materialization

- Intermediate materialization (or caching) cannot always be avoided.
- It is needed for blocking operations, which repeatedly access the same items returned by an iterator.
- Examples of blocking operations include:
  - ORDER and GROUP clauses in FLWOR expressions
  - sequence (i.e., existentially quantified) comparison
  - filter expressions with multiple predicates or variables that occur multiple times in a query

An Extended Iterator API

- Relatively to the classical case, BaseX uses an extended API for iterators including the following additional operations:
  - *Size()* returns the number of iterable items if known, or *-1* otherwise.
  - *Get(int)* retrieves the item in position *int* if positional access is possible (i.e., the number of items is known), or null otherwise.
  - *Reset()* points the iterator to the first item and returns *true* if reset is supported, or *false* otherwise.
  - *Reverse()* reverses the order of the iterable items and returns *true* if reverse traversal is supported, or *false* otherwise.
  - *Finish()* returns a sequence with the remaining (or possibly all the) iterable items.

SIDOAED, Again

- Note that the nodes that result from the evaluation of location steps and node combination operators (i.e., `union`, `intersect`, `except`) must be SIDOAED, i.e., they must be in document order and free of duplicates.

- This requirement may cause additional costs and should thus be skipped if nodes are known to be SIDOAED.

- If it is known that nodes are SIDOEAED at compile time, the nodes need not be materizalized and the expression can be iteratively evaluated.

- Otherwise, extra care must be taken to ensure that the resulting nodes are SIDOAED.

Overhead Avoidance

- For non-blocking operators, it appears reasonable at first glance to apply iterative query processing whenever possible.
- There are some cases, however, in which this is suboptimal because of overheads.
- Before using an iterator, the query processor checks whether it is possible to determine that the sequence is empty or a singleton, in which case an iterator is not used as the overhead of doing so cannot be recouped.

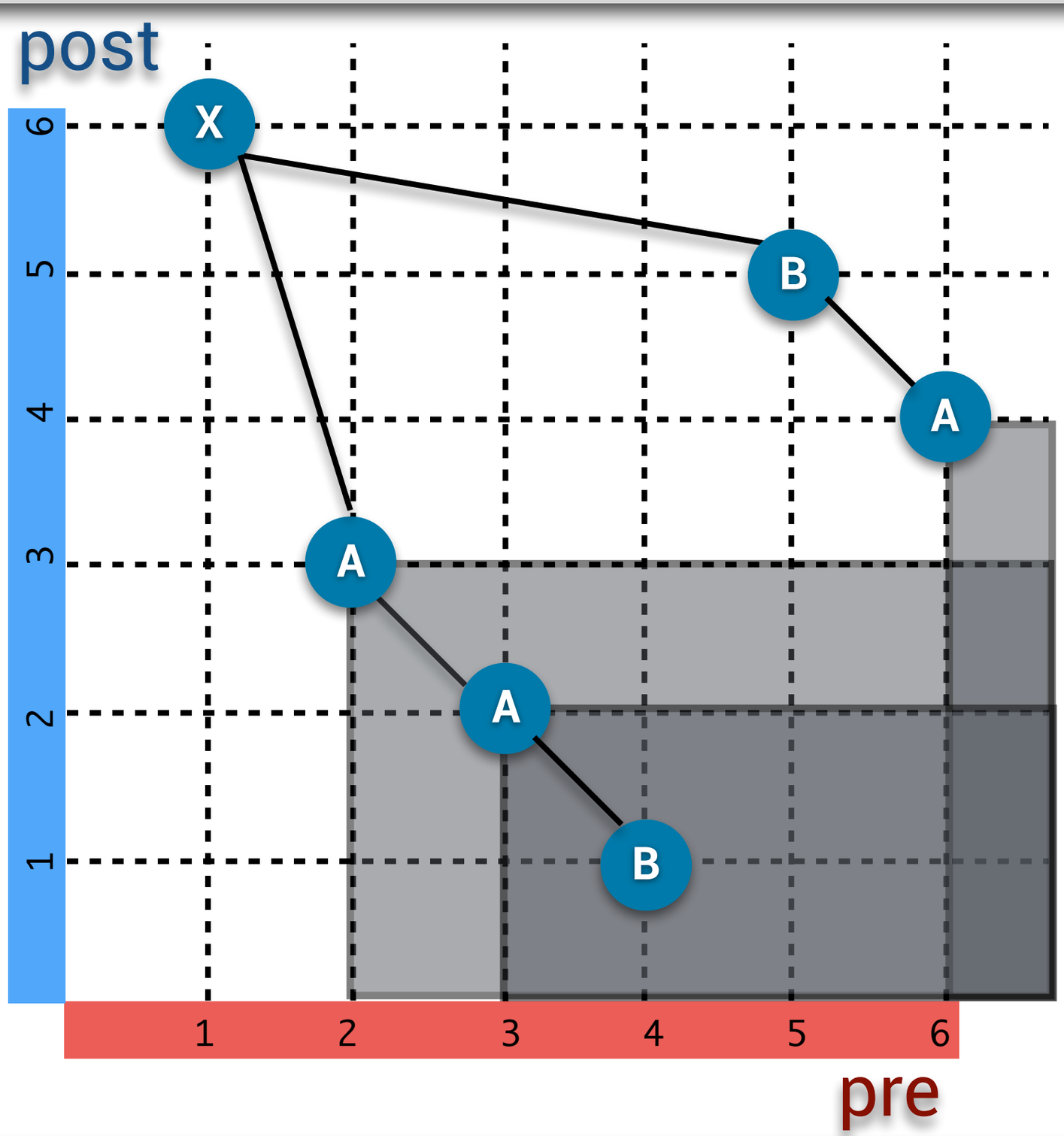
Evaluation Expressions and Functions

- Some operations (e.g., **`union`**, **`intersect`**, and **`except`**) can be more efficiently evaluated if it is known that their inputs are ordered.
- Complex Boolean expressions can also short-circuit.
- For example, they can be applied in linear fashion, and, if possible, curtail the sequence of predicate applications as soon as it is detected that continuation will not change the final outcome.
- Finally, many XQuery functions (e.g., **`index-of`**, **`reverse`**, etc.) consume and produce sequences, and are, therefore, implementable by iterators.

Evaluating Location Paths [1]

- BaseX evaluates location paths using an approach inspired by techniques developed for the *staircase join algorithm* (Grust-van Keulen, 2003).
- The staircase join builds upon the pre/post plane table-encoding of XML discussed in relation to storage.
- Three optimization stategies were proposed that are complementary to one another and relevant here.
- They are referred to as pruning, partitioning and skipping.
- The next slides provide an illustration in the case of descendant steps.

Evaluating Location Paths [2]

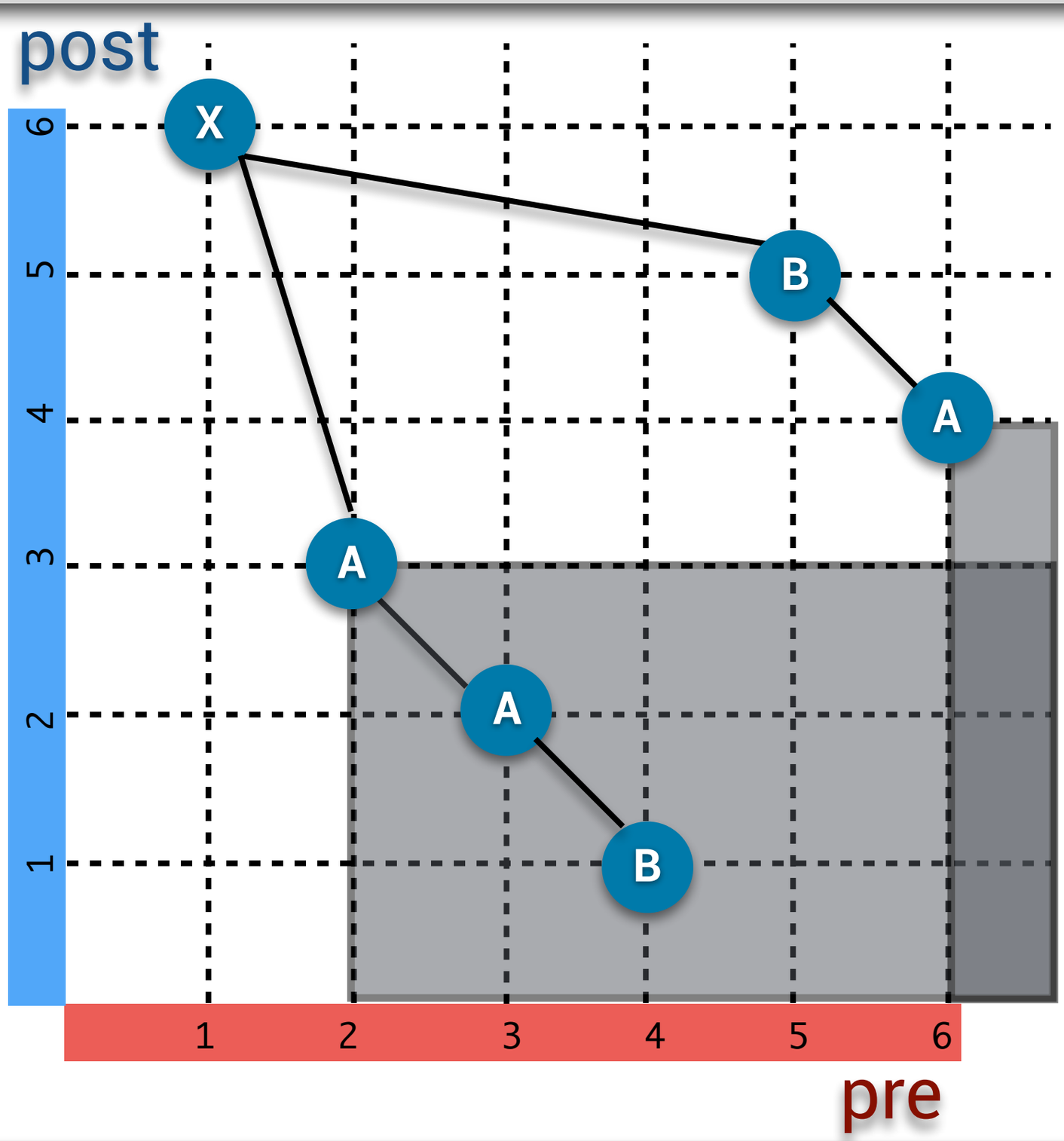


With no optimization

Each region with an A node in the top-left corner needs to be scanned left to right to a total of 20-1 + 12-1 + 5-1 coordinates.

Some coordinates are scanned more than once.

Evaluating Location Paths [3]

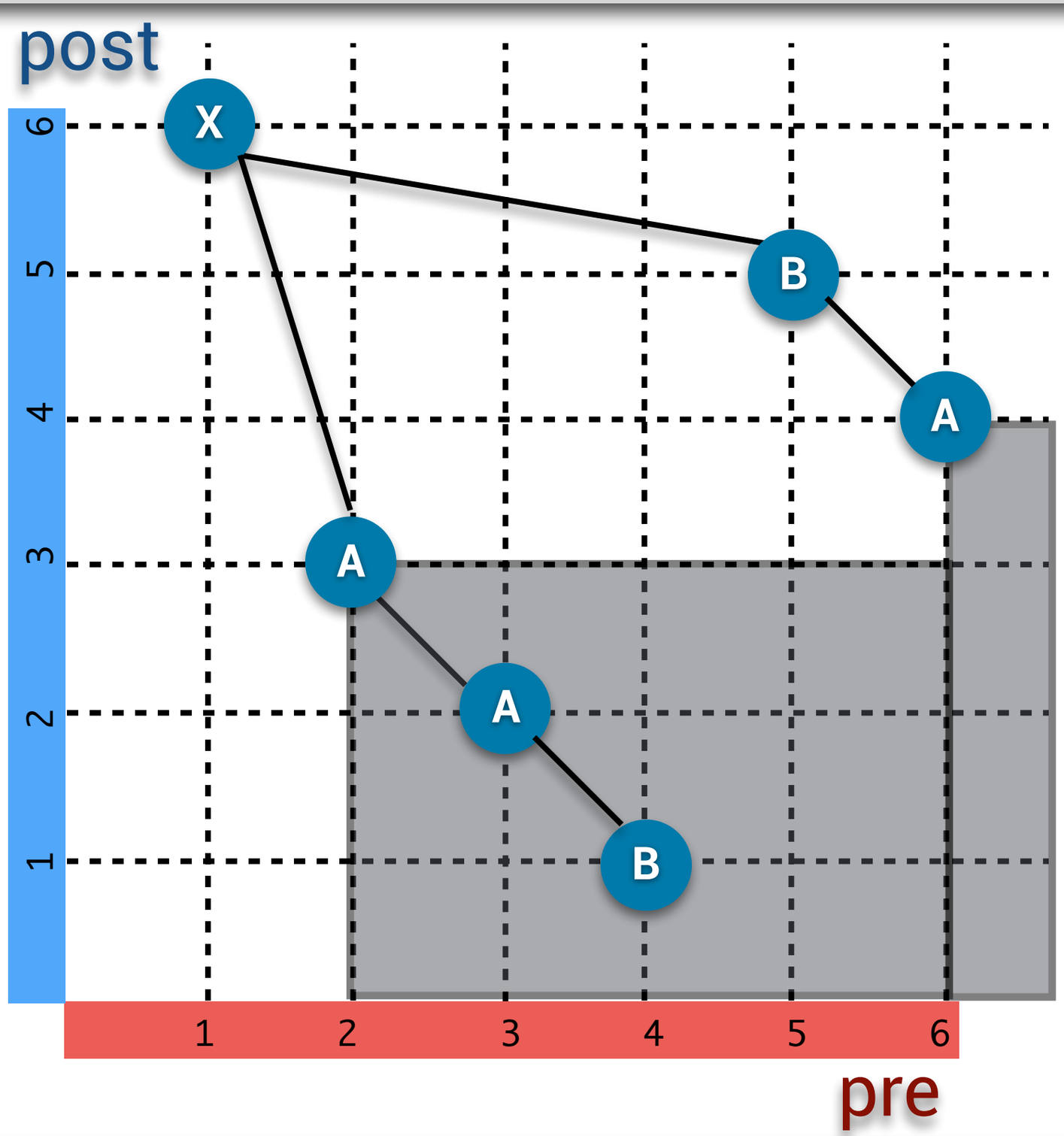


After pruning

Since the region where the descendants of A at (3,2) lie is contained in that where the descendants of A at (2,3) lie, it needn't be scanned

Fewer coordinates are scanned more than once.

Evaluating Location Paths [4]

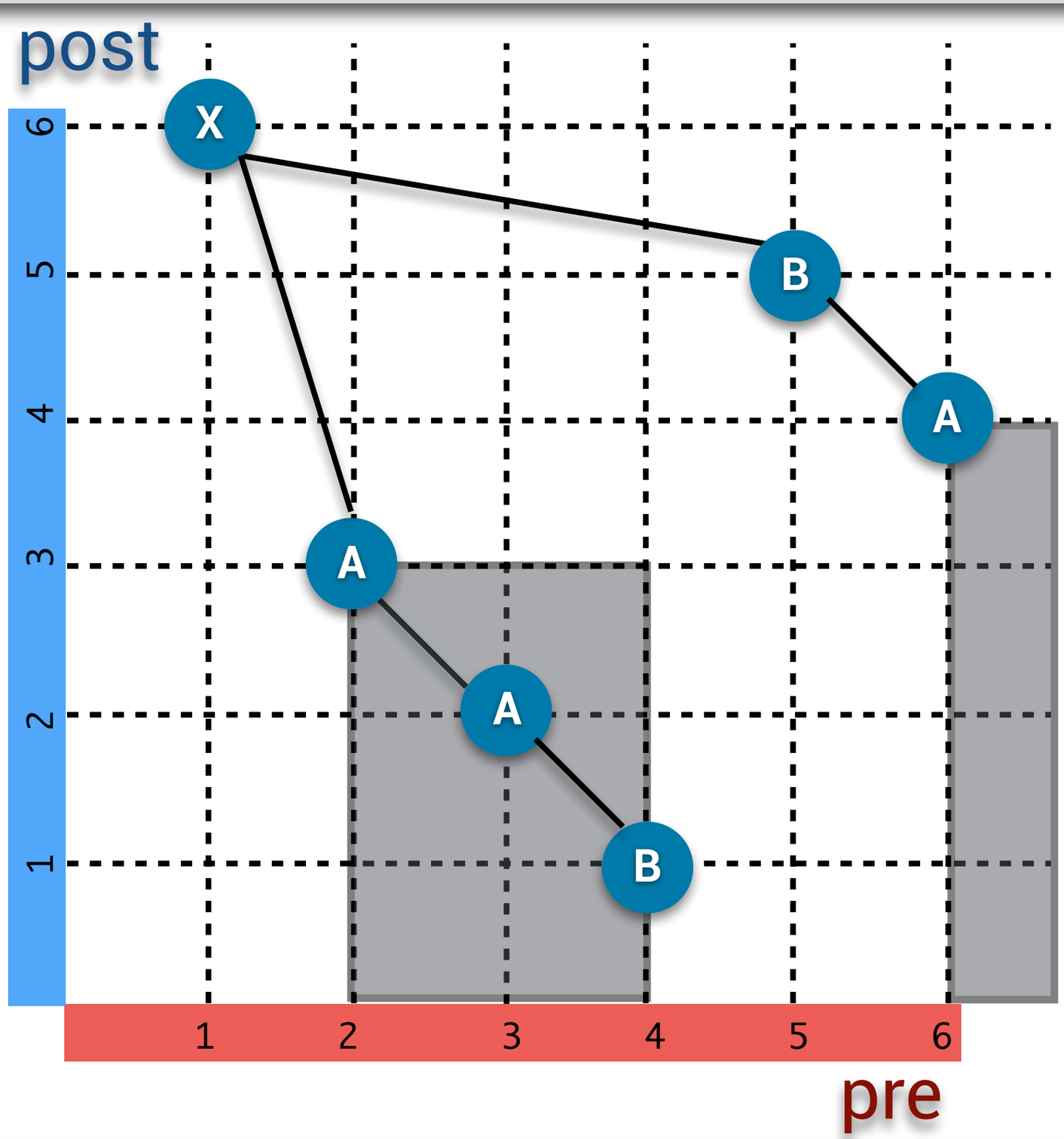


After partitioning

The overlap between regions is eliminated.

No coordinate is scanned more than once.

Evaluating Location Paths [5]



After partitioning

No scanning is performed after (from the post values) we know that all descendants have been scanned.

Fewer coordinates are scanned unnecessarily.

Evaluating Location Paths [6]

- The same approach can be used for other axes.
- The same approach can be deployed using other also encodings, such as pre/dist/size, as used by BaseX.
- In fact, as shown in (Grün, 2010), there are advantages stemming from the BaseX encoding as there is no need for a skipping step.
- Also, using the BaseX encoding pruning can be done on-the-fly.
- It has been observed in practice that many location paths return small result sets with few duplicates.
- This reduces the benefits from using the staircase join technique.
- BaseX makes the most of its encoding and implements the evaluation of location paths using simple iterators.

Query Evaluation in BaseX

Seven Example Queries: Compilation Steps, Compiled Queries, Evaluation Plans

Examples [1.a]

Query:

```
doc("auction.xml")/
  descendant::item[@id = 'item0']
```

Compiling:

- pre-evaluating doc("auction.xml")
- applying attribute index for "item0"

Optimized Query:

```
db:attribute("auction", "item0")/
  self::*:id/
  parent::*:item
```

In optimizing this query, the doc() function is pre-evaluated to get access to the database meta information.

Then, the comparison operator is rewritten for index access and the predicate is rewritten to an inverted path.

Examples [1.b]

Note the tree depth of 1: a root and three children

The leftmost child is a value index access followed by two axis step iterations.

Query plan:

```
<QueryPlan compiled="true">
 <CachedPath>
  <ValueAccess
    data="auction" type="ATTRIBUTE">
   <Str value="item0" type="xs:string"/>
  </ValueAccess>
  <IterStep axis="self" test="*:id"/>
  <IterStep axis="parent" test="*:item"/>
 </CachedPath>
</QueryPlan
```

Optimized Query:

```
db:attribute("auction", "item0")/
  self::*:id/
  parent::*:item
```

Examples [2.a]

Query:

```
doc("auction.xml")//
  item[payment = 'Creditcard']/
  ..
```

Compiling:

- pre-evaluating doc("auction.xml")
- rewriting descendant-or-self step(s)
- applying text index for "Creditcard"

Optimized Query:

```
db:text("auction", "Creditcard")/
  parent::*:payment/
  parent::*:item/
  ..
```

In optimizing this query, the doc() is pre-evaluated to get access to the database meta information.

Then, the descendant-or-self path is rewritten into child paths.

Then, the comparison operator is rewritten for index access and the predicate is rewritten to an inverted path.

Examples [2.b]

The query plan is structurally similar to that for Example 1.

Query plan:

In Ex. 1 the index was on attribute values, here it is on the textual content of an element

```
<QueryPlan compiled="true">
 <CachedPath>
  <ValueAccess
   data="auction" type="TEXT" name="*:payment">
   <Str value="Creditcard" type="xs:string"/>
  </ValueAccess>
  <IterStep axis="parent" test="*:item"/>
  <IterStep axis="parent" test="node()"/>
 </CachedPath>
</QueryPlan>
```

Optimized Query:

```
db:text("auction", "Creditcard")/
  parent::*:payment/
  parent::*:item/
  ..
```

Examples [3.a]

Query:

```
(for $i in doc('auction.xml')//item
 where $i/payment = 'Creditcard'
 return $i)/..
```

Compiling:

- pre-evaluating doc("auction.xml")
- rewriting descendant-or-self step(s)
- converting descendant::*:item to child steps
- applying text index for "Creditcard"
- rewriting where clause(s)
- simplifying flwor expression

Optimized Query:

```
db:text("auction", "Creditcard")/
  parent::*:payment/
  parent::*:item[parent::*/
  parent::*:regions/
  parent::*:site/
  parent::document-node()]/
  ..
```

The original query is written as a FLWOR expression.

So, the compiler first attaches the WHERE clause as a predicate to the location step of the FOR clause.

The doc() function is pre-evaluated to get access to the database meta information.

The descendant-or-self path is rewritten into child paths.

The comparison operator is rewritten for index access and the predicate is rewritten to an inverted path.

After all other optimizations have been performed, the FLWOR expression is eliminated, and the resulting index path is merged with the suffixed parent step.

Examples [3.b]

Note that this tree is deeper than in Ex. 1-2

Otherwise, the basic elements are similar: index accesses and iteration steps

Optimized Query:

```
db:text("auction", "Creditcard")/
  parent::*:payment/
  parent::*:item[parent::*/
  parent::*:regions/
  parent::*:site/
  parent::document-node()]/
  ..
```

Query plan:

```
<QueryPlan compiled="true">
 <CachedPath>
  <ValueAccess
   data="auction" type="TEXT" name="*:payment">
   <Str value="Creditcard" type="xs:string"/>
  </ValueAccess>
  <CachedStep axis="parent" test="*:item">
   <CachedPath>
    <IterStep axis="parent" test="*"/>
    <IterStep axis="parent" test="*:regions"/>
    <IterStep axis="parent" test="*:site"/>
    <IterStep axis="parent" test="document-node()"/>
   </CachedPath>
  </CachedStep>
  <IterStep axis="parent" test="node()"/>
 </CachedPath>
</QueryPlan>
```

Examples [4.a]

Query:

```
let $auction := doc("auction.xml")
return
  for $b in
    $auction/site/people/person[@id = "person0"]
  return $b/name/text()
```

Compiling:

- pre-evaluating doc("auction.xml")
- inlining $auction_0
- applying attribute index for "person0"
- simplifying flwor expression

Optimized Query:

```
for $b_1 in db:attribute("auction", "person0")/
            self::*:id/
            parent::*:person
return $b_1/*:name/text()
```

The original query is written as a FLWOR expression with only L and R components, where the latter is a nested FLWOR expression itself.

The doc() function is pre-evaluated to get access to the database meta information and the variable declared in the LET is inlined.

The comparison operator is rewritten for index access and the predicate is rewritten to an inverted path.

After all other optimizations have been performed, the FLWOR expression is eliminated.

Examples [4.b]

Note that this is the first example of FLWOR-structured query plan

Query plan:

```
<QueryPlan compiled="true">
 <GFLWOR>
  <For>
   <Var name="$b" id="1"/>
   <IterPath>
    <ValueAccess data="auction" type="ATTRIBUTE">
     <Str value="person0" type="xs:string"/>
    </ValueAccess>
    <IterStep axis="self" test="*:id"/>
    <IterStep axis="parent" test="*:person"/>
   </IterPath>
  </For>
  <IterPath>
   <VarRef>
    <Var name="$b" id="1"/>
   </VarRef>
   <IterStep axis="child" test="*:name"/>
   <IterStep axis="child" test="text()"/>
  </IterPath>
 </GFLWOR>
</QueryPlan>
```

Note that the VarRef takes values from Var in the For clause

Optimized Query:

```
for $b_1 in
  db:attribute("auction", "person0")/
    self::*:id/
    parent::*:person
return $b_1/*:name/text()
```

Examples [5.a]

Query:

```
let $auction := doc("auction.xml")
return
  for $p in $auction/site
  return count($p//description) +
         count($p//annotation) +
         count($p//emailaddress)
```

Compiling:

- pre-evaluating doc("auction.xml")
- rewriting descendant-or-self step(s)
- rewriting descendant-or-self step(s)
- rewriting descendant-or-self step(s)
- inlining $auction_0
- rewriting singleton for to let
- inlining $p_1
- pre-evaluating count(document-node {"auction.xml"}/*:site/descendant::*:description)
- pre-evaluating count(document-node {"auction.xml"}/*:site/descendant::*:annotation)
- pre-evaluating (44500 + 21750)
- pre-evaluating count(document-node {"auction.xml"}/*:site/descendant::*:emailaddress)
- pre-evaluating (66250 + 25500)
- simplifying flwor expression
- simplifying flwor expression

Optimized Query:

```
91750
```

The original query is written as a FLWOR expression with only L and R components, where the latter is a nested FLWOR expression itself.

The doc() function is pre-evaluated to get access to the database meta information and the variable declared in the LET is inlined.

The power of path summaries is illustrated here: (a) $p can be inlined because the path summary indicates it returns a singleton, (b) after the descendant-or-self and child steps are rewritten, the path summaries directly provide information on the counts.

After all other optimizations have been performed, the FLWOR expressions are eliminated.

The query can be evaluted at compile time, with zero accesses to document content.

Examples [5.b]

This is the general form of a query whose compilation alone yields the desired value.

Query plan:

```
<QueryPlan compiled="true">
  <Int value="91750" type="xs:integer"/>
</QueryPlan>
```

Optimized Query:

```
91750
```

Examples [6.a]

Query:

```
let $auction := doc("auction.xml")
return
  let $ca := $auction/site/closed_auctions/closed_auction
  return
      let $ei :=  $auction/site/regions/europe/item
      for $p in $auction/site/people/person
      let $a := for $t in $ca
                where $p/@id = $t/buyer/@person
                return
                  let $n := for $t2 in $ei
                            where $t/itemref/@item = $t2/@id
                            return $t2
                  return
                    <item>{$n/name/text()}</item>
return
  <person name="{$p/name/text()}">{ $a }</person>
```

Examples [6.b]

Compiling
:
- pre-evaluating doc("auction.xml")
- swapping operands: (@*:id = $t_4/*:itemref/@*:item)
- rewriting where clause(s)
- simplifying flwor expression
- inlining $n_6
- simplifying flwor expression
- swapping operands: (*:buyer/@*:person = $p_3/@*:id)
- rewriting where clause(s)
  - flattening
  - let $ca_1 := $auction_0/*:site/*:closed_auctions/*:closed_auction return
  - let $ei_2 := $auction_0/*:site/*:regions/*:europe/*:item
  - for $p_3 in $auction_0/*:site/*:people/*:person
  - let $a_7 := for $t_4 in ($ca_1)[(*:buyer/@*:person = $p_3/@*:id)]
  - return element item { (($ei_2)[(@*:id = $t_4/*:itemref/@*:item)]/name/text()) }
  - return element person { (attribute name { ($p_3/*:name/text()) }, $a_7) }
- inlining $auction_0
- inlining $ca_1
- applying attribute index for $p_3/@*:id
- inlining $ei_2
- applying attribute index for $t_4/*:itemref/@*:item
- simplifying flwor expression

Examples [6.c]

Optimized Query:

```
for $p_3 in document-node {"auction.xml"}/*:site/*:people/*:person
  let $a_7 :=
    for $t_4 in db:attribute("auction", $p_3/@*:id)/
                self::*:person/
                parent::*:buyer/
                parent::*:closed_auction
    return element item { (db:attribute("auction", $t_4/*:itemref/@*:item)/
                           self::*:id/
                           parent::*:item[parent::*:europe/
                                          parent::*:regions/
                                          parent::*:site/
                                          parent::document-node()]/
                                          name/text()) }
return element person { (attribute name { ($p_3/*:name/text()) }, $a_7) }
```

Examples [6.d]

Query plan:

```
<QueryPlan compiled="true">
 <GFLWOR>
  <For>
   <Var name="$p" id="3"/>
   <IterPath>
    <DBNode name="auction" pre="0"/>
    <IterStep axis="child" test="*:site"/>
    <IterStep axis="child" test="*:people"/>
    <IterStep axis="child" test="*:person"/>
   </IterPath>
  </For>
  <Let>
   <Var name="$a" id="7"/>
   <GFLWOR>
    <For>
     <Var name="$t" id="4"/>
     <CachedPath>
      <ValueAccess data="auction" type="ATTRIBUTE">
       <IterPath>
        <VarRef>
         <Var name="$p" id="3"/>
        </VarRef>
        <IterStep axis="attribute" test="*:id"/>
       </IterPath>
      </ValueAccess>
      <IterStep axis="self" test="*:person"/>
      <IterStep axis="parent" test="*:buyer"/>
      <IterStep axis="parent" test="*:closed_auction"/>
     </CachedPath>
    </For>
    <CElem>
     <QNm value="item" type="xs:QName"/>
     <CachedPath>
      <ValueAccess data="auction" type="ATTRIBUTE">
       <IterPath>
        <VarRef>
         <Var name="$t" id="4"/>
        </VarRef>
        <IterStep axis="child" test="*:itemref"/>
        <IterStep axis="attribute" test="*:item"/>
       </IterPath>
      </ValueAccess>
      <IterStep axis="self" test="*:id"/>
      <IterStep axis="parent" test="*:item">
       <CachedPath>
        <IterStep axis="parent" test="*:europe"/>
        <IterStep axis="parent" test="*:regions"/>
        <IterStep axis="parent" test="*:site"/>
        <IterStep axis="parent" test="document-node()"/>
       </CachedPath>
      </IterStep>
      <IterStep axis="child" test="name"/>
      <IterStep axis="child" test="text()"/>
     </CachedPath>
    </CElem>
   </GFLWOR>
  </Let>
  <CElem>
   <QNm value="person" type="xs:QName"/>
   <CAttr>
    <QNm value="name" type="xs:QName"/>
    <IterPath>
     <VarRef>
      <Var name="$p" id="3"/>
     </VarRef>
     <IterStep axis="child" test="*:name"/>
     <IterStep axis="child" test="text()"/>
    </IterPath>
   </CAttr>
   <VarRef>
    <Var name="$a" id="7"/>
   </VarRef>
  </CElem>
 </GFLWOR>
</QueryPlan>
```

Examples [7.a]

Query:

```
let $auction := doc("auction.xml")
return
  for $a in $auction/descendant::closed_auction[price >= 500 and price <= 1000]
    for $i in $auction/descendant::item
        for $c in $auction/descendant::category
        where  $a/itemref/@item = $i/@id
        and    $c/@id = $i/incategory/@category
    return $c/name
```

Examples [7.b]

Compiling:

- pre-evaluating doc("auction.xml")
- rewriting (*:price >= 500)
- rewriting (*:price <= 1000)
- rewriting (500.0 <= *:price and *:price <= 1000.0)
- swapping operands: (@*:id = $a_1/*:itemref/@*:item)
- rewriting where clause(s)
- inlining $auction_0
- converting descendant::*:closed_auction[500.0 <= *:price <= 1000.0] to child steps
- applying attribute index for $a_1/*:itemref/@*:item
- applying attribute index for $i_2/*:incategory/@*:category
- simplifying flwor expression

Examples [7.c]

Optimized Query:

```
for $a_1 in document-node {"auction.xml"}/
            *:site/*:closed_auctions/
            *:closed_auction[500.0 <= *:price <= 1000.0]
  for $i_2 in db:attribute("auction", $a_1/*:itemref/@*:item)/
              self::*:id/
              parent::*:item
    for $c_3 in db:attribute("auction", $i_2/*:incategory/@*:category)/
                self::*:id/
                parent::*:category
return $c_3/*:name
```

Examples [7.d]

Query plan:

```
<QueryPlan compiled="true">
 <GFLWOR>
  <For>
   <Var name="$a" id="1"/>
   <IterPath>
    <DBNode name="auction" pre="0"/>
    <IterStep axis="child" test="*:site"/>
    <IterStep axis="child" test="*:closed_auctions"/>
    <IterStep axis="child" test="*:closed_auction">
     <CmpR min="500.0" max="1000.0">
      <CachedPath>
       <IterStep axis="child" test="*:price"/>
      </CachedPath>
     </CmpR>
    </IterStep>
   </IterPath>
  </For>
```

```
  <For>
   <Var name="$i" id="2"/>
   <CachedPath>
    <ValueAccess data="auction" type="ATTRIBUTE">
     <IterPath>
      <VarRef>
       <Var name="$a" id="1"/>
      </VarRef>
      <IterStep axis="child" test="*:itemref"/>
      <IterStep axis="attribute" test="*:item"/>
     </IterPath>
    </ValueAccess>
    <IterStep axis="self" test="*:id"/>
    <IterStep axis="parent" test="*:item"/>
   </CachedPath>
  </For>
```

```
  <For>
   <Var name="$c" id="3"/>
   <CachedPath>
    <ValueAccess data="auction" type="ATTRIBUTE">
     <IterPath>
      <VarRef>
       <Var name="$i" id="2"/>
      </VarRef>
      <IterStep axis="child" test="*:incategory"/>
      <IterStep axis="attribute" test="*:category"/>
     </IterPath>
    </ValueAccess>
    <IterStep axis="self" test="*:id"/>
    <IterStep axis="parent" test="*:category"/>
   </CachedPath>
  </For>
  <IterPath>
   <VarRef>
    <Var name="$c" id="3"/>
   </VarRef>
   <IterStep axis="child" test="*:name"/>
  </IterPath>
 </GFLWOR>
</QueryPlan>
```

Conclusions

Topics This Week

- We have taken a quick, example-driven tour of XQuery, to remind ourselves of the basic characteristics of the language.
- We have introduced and briefly explored an algebraic view of XQuery, and some equivalence laws that arise, on the basis of which one can conceive of a rewriting approach to logical optimization for the language.
- We then studied storage, optimization and evaluation of XQuery as implemented in the BaseX XML native DBMS.

Learning Objectives This Week

- Having completed our exploration of classical query processing, we were ready to use what we have learnt and explore querying data on the Web, in the narrower sense, starting with XML/XQuery.
- We have aimed at acquiring a deeper understanding of some of the challenges arising in querying over large XML collections using XQuery.

Topics Next Week

- We will take a quick, example-driven tour of SPARQL, to remind ourselves of the basic characteristics of the language.
- We wil introduce and briefly explore an algebraic view of SPARQL, and some equivalence laws that arise, on the basis of which one can conceive of a rewriting approach to logical optimization for the language.
- We will then briefly study storage, optimization and evaluation of SPARQL.

Learning Objectives Next Week

- Having acquired a deeper understanding of some of the challenges arising in querying over large XML collections using XQuery, we will shift our attention to querying over large RDF triple sets using SPARQL.

Bibliography

References

1. **XQuery: An XML query language**. Donald D. Chamberlin. *IBM Systems Journal* 41(4): 597-615 (2002) http://dx.doi.org/10.1147/sj.414.0597

2. **XQuery from the Experts**. D. Chamberlin, D. Draper, M. Fernández et al. *Addison-Wesley, 2004*. ISBN 0-321-18060-7. © Pearson Education Limited 2004

3. **XQuery Tutorial**. Peter Fankhauser, Philip Wadler. *XML 2001, Orlando, 9-14 December 2001*. http://homepages.inf.ed.ac.uk/wadler/papers/xquery-tutorial/xquery-tutorial.pdf

4. **Storing and Querying Large XML Instances**. Christian Grün. *Doctoral dissertation, University of Konstanz, 2010*. http://kops.uni-konstanz.de/handle/123456789/6106

5. **Tree Awareness for Relational DBMS Kernels: Staircase Join**. Torsten Grust, Maurice van Keulen. Intelligent Search on XML Data 2003: 231-245. http://dx.doi.org/10.1007/978-3-540-45194-5_16

6. **An Introduction to XML and Web Technologies**. Anders Møller and Michael Schwartzbach. *Addison-Wesley, 2006*. ISBN 978-0-321-26966-9. http://www.brics.dk/ixwt/ © Pearson Education Limited 2006

7. **XQuery**. Priscilla Walmsley. *O'Reilly, 2007*. ISBN 978-0-596-00634-1. http://www.datypic.com/books/xquery/ © Priscilla Walmsley 2007

W3C Documents

I. **XML Path Language (XPath) 2.0 (Second Edition)**. Anders Berglund, Scott Boag, Don Chamberlin, Mary F. Fernández, Michael Kay, Jonathan Robie, Jérôme Siméon. http://www.w3.org/TR/xpath20/

II. **XQuery 1.0: An XML Query Language (Second Edition)**. Scott Boag, Don Chamberlin, Mary F. Fernández, Daniela Florescu, Jonathan Robie, Jérôme Siméon. http://www.w3.org/TR/xquery/

III. **XML Query Use Cases**. Don Chamberlin, Peter Fankhauser, Daniela Florescu, Massimo Marchiori, Jonathan Robie. http://www.w3.org/TR/xquery-use-cases/

IV. **XQuery 1.0 and XPath 2.0 Formal Semantics (Second Edition)**. Denise Draper, Michael Dyck, Peter Fankhauser, Mary Fernández, Ashok Malhotra, Kristoffer Rose, Michael Rys, Jérôme Siméon, Philip Wadler. http://www.w3.org/TR/query-semantics/

V. **The XML Query Algebra**. Peter Fankhauser, Mary Fernández, Ashok Malhotra, Michael Rys, Jérôme Siméon, Philip Wadler. http://www.w3.org/TR/2001/WD-query-algebra-20010215/

VI. **XQuery 1.0 and XPath 2.0 Functions and Operators (Second Edition)**. Ashok Malhotra, Jim Melton, Norman Walsh, Michael Kay. http://www.w3.org/TR/xpath-functions/

VII. **XQuery 3.0: An XML Query Language**. Jonathan Robie, Don Chamberlin, John Snelson. http://www.w3.org/TR/xquery-30/