



This work is licensed under a Creative Commons Attribution 4.0 International License

CMPUT391

Text and Documents

Instructor: Denilson Barbosa

University of Alberta

October 10, 2023

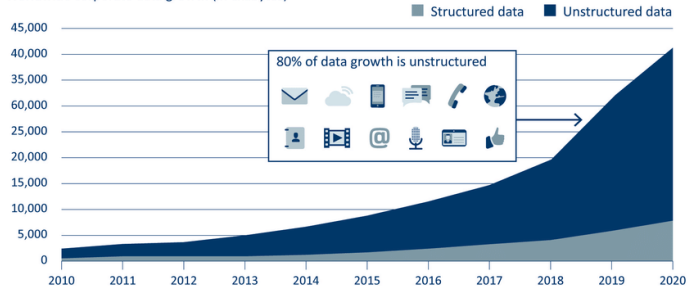
Slides by D. Barbosa, with suggested corrections and improvements by (in alphabetical order) C. Bins, D. Caminhas, K. Guhzva, Q. Lautischer, E. Macdonald, M. A. Nascimento, K. Newbury, M. Strobl, D. Sunderman, K. Wang, and K. Wong.

In a world of Big Data...

Despite all tremendous recent improvements in computing power and information technology, most information used by individuals and corporations is **not relational**.

Massive growth in unstructured content

Worldwide corporate data growth (in exabytes)



Source: The Digital Universe

XML — The Extensible Markup Language

XML Query Languages

XML Support in DBMSs

API Support for Application Development

XML — The Extensible Markup Language

The Web As Of Not Too Long Ago

HTML: HyperText Markup Language

- W3C Standard implemented by most browsers
- Still not fully supported across all platforms

HTML enabled mass publishing of human-generated content (e.g., news, blogs, etc.) and server-generated content (by pulling data out of a relational database)

- PHP, JavaScript, ASP...

HTML was designed to present content for **human consumption**. That is, it specifies how to make content look nice on the browser, not what that content **means**.

Markup Languages

Markup: annotations (tags) in a plain text document

HTML's markup vocabulary is **fixed** and tags have **pre-specified semantics**:

- `<head>` tag is for metadata while the "data" goes into the `<body>` tag.
- HTML has tags for choosing fonts and colors, drawing tables, adding line breaks, paragraphs, hierarchical headers, etc.

Tags need not be closed in HTML

```
<HTML>
  <head>
    <title>Movie List</title>
    <meta name="author"
          content="John Smith">
  </head>
  <body>
    <h1> Movie List </h1>
    <p> <i> Ghostbusters </i> (1984)
      <br> <b> Ivan Reitman </b>
      <br> With Bill Murray,
          Sigourney Weaver

    <p> <i> Groundhog Day </i> (1993)
      <br> <b> Harold Ramis </b>
      <br> With Bill Murray,
          Andie MacDowell

  </body>
</HTML>
```

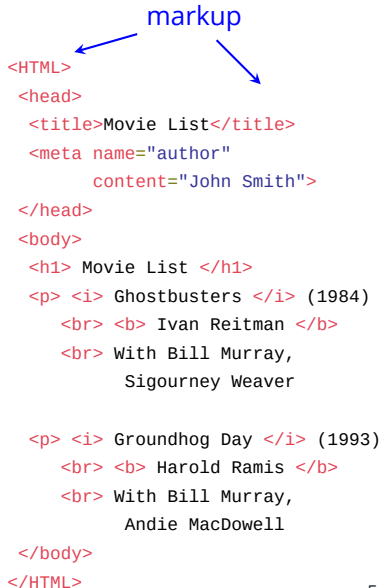
Markup Languages

Markup: annotations (tags) in a plain text document

HTML's markup vocabulary is **fixed** and tags have **pre-specified semantics**:

- `<head>` tag is for metadata while the "data" goes into the `<body>` tag.
- HTML has tags for choosing fonts and colors, drawing tables, adding line breaks, paragraphs, hierarchical headers, etc.

Tags need not be closed in HTML



```
graph TD; markup --> HTML["<HTML>"]; markup --> code["<head>\n  <title>Movie List</title>\n  <meta name='author' content='John Smith'>\n</head>\n<body>\n  <h1> Movie List </h1>\n  <p> <i> Ghostbusters </i> (1984)\n    <br> <b> Ivan Reitman </b>\n    <br> With Bill Murray,\n    Sigourney Weaver\n\n  <p> <i> Groundhog Day </i> (1993)\n    <br> <b> Harold Ramis </b>\n    <br> With Bill Murray,\n    Andie MacDowell\n\n</body>\n</HTML>"]
```

markup

```
<HTML>
<head>
  <title>Movie List</title>
  <meta name="author"
        content="John Smith">
</head>
<body>
  <h1> Movie List </h1>
  <p> <i> Ghostbusters </i> (1984)
    <br> <b> Ivan Reitman </b>
    <br> With Bill Murray,
    Sigourney Weaver

  <p> <i> Groundhog Day </i> (1993)
    <br> <b> Harold Ramis </b>
    <br> With Bill Murray,
    Andie MacDowell

</body>
</HTML>
```

Markup Languages

Markup: annotations (tags) in a plain text document

HTML's markup vocabulary is **fixed** and tags have **pre-specified semantics**:

- `<head>` tag is for metadata while the "data" goes into the `<body>` tag.
- HTML has tags for choosing fonts and colors, drawing tables, adding line breaks, paragraphs, hierarchical headers, etc.

Tags need not be closed in HTML

```
<HTML>
<head>
  <title>Movie List</title>
  <meta name="author"
        content="John Smith">
</head>
<body>
  <h1> Movie List </h1>
  <p> <i> Ghostbusters </i> (1984)
    <br> <b> Ivan Reitman </b>
    <br> With Bill Murray,
          Sigourney Weaver

  <p> <i> Groundhog Day </i> (1993)
    <br> <b> Harold Ramis </b>
    <br> With Bill Murray,
          Andie MacDowell
</body>
</HTML>
```


The Extensible Markup Language

W3C recommendation markup language that allows user-defined vocabulary.

- Programmers choose their own tags.
- Applications can interpret the tags any way they want.

XML tags must always be closed in the reverse order in which they are opened.

- Ensures the document can be parsed correctly.

```
<movies>
  <meta name="author"
        content="John Smith"/>
  <movie>
    <title>Ghostbusters</title>
    <year>1984</year>
    <director>Ivan Reitman</director>
    <cast>
      <actor>Bill Murray</actor>
      <actor>Sigourney Weaver</actor>
    </cast>
  </movie>
  <movie>
    <title>Groundhog Day</title>
    <year>1993</year>
    <director>Harold Ramis</director>
    <cast>
      <actor>Bill Murray</actor>
      <actor>Andie MacDowell</actor>
    </cast>
  </movie>
</movies>
```

XML vs HTML

In summary:

- Both are markup languages that apply to plain text documents
- HTML has a finite and pre-defined set of tags; with XML, you can choose your own set of tags
- HTML and XML documents can be represented as trees (through the Document Object Model)
- Closing tags in HTML is optional; the browser makes educated guesses about the shape of the tree
- Closing tags in XML is required; there is only one DOM tree that represents a well-formed XML document

XML Standards

XSL/XSLT: presentation and transformation

- ex: transforming XML into plain HTML content

RDF: resource description framework

- used for describing meta-data about XML documents
- foundation for the Semantic Web

Xpointer/Xlink: specifies links across documents and elements

DOM: Document Object Model for representing XML documents

- implementations in all popular languages

SAX: Simple API for XML parsing

- event-based framework
- implementations in all popular languages

What is XML good for?

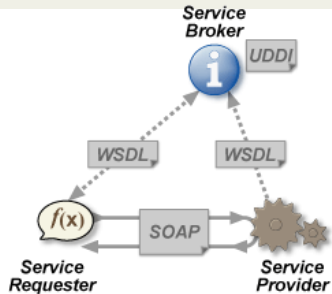
Web Services¹ are software systems designed to support interoperable machine-to-machine interaction over the Web (or some other network).

With Service-Oriented Architectures, an entire application can be implemented by calling services that perform specific tasks.

XML is used to encode messages between the application (the requester) and the service provider. The application can choose among many providers offering the same service.

See CMPUT401!

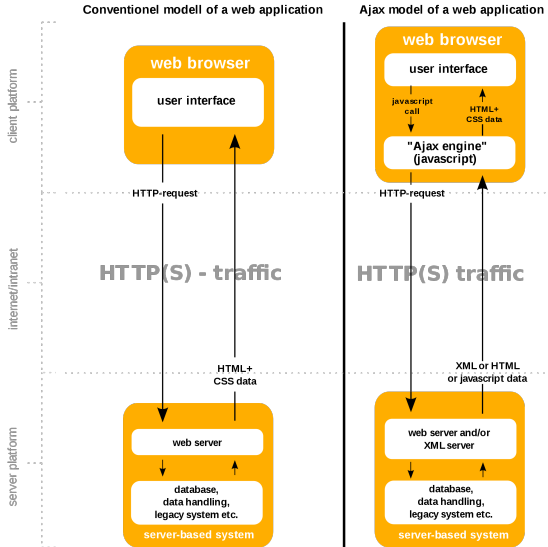
¹https://en.wikipedia.org/wiki/Web_service



What is XML good for?

Ajax², short for asynchronous JavaScript and XML, is a set of web development techniques using many web technologies on the client side to create asynchronous web applications.

See CMPUT404!



²[https://en.wikipedia.org/wiki/Ajax_\(programming\)](https://en.wikipedia.org/wiki/Ajax_(programming))

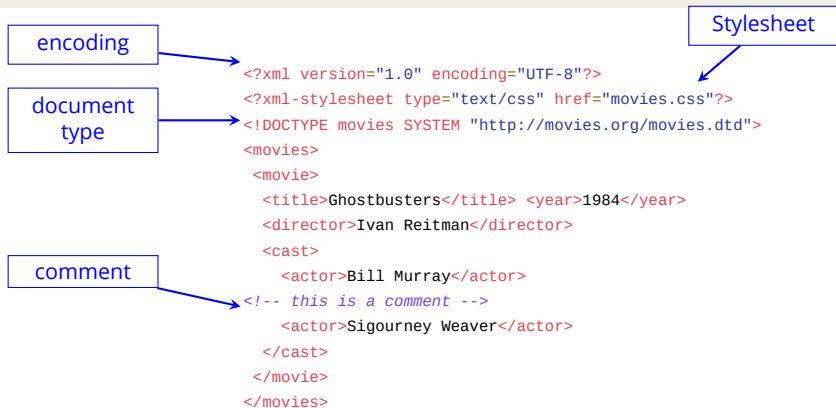
What is XML good for?

XML is great for **encoding and exchanging data between** existing (database-backed) **systems**.

Why?

- XML is **text-based**: it is a lot easier to read or write text files than binary files (e.g., the programmer can inspect the file while debugging).
- It enables incremental data sharing formats; once two systems can talk, it is easy to extend the markup to support more systems.
- XML is an open standard, supported by many tools (and developers).

What Does XML Look Like?



An XML element consists of an opening tag, the corresponding closing tag, and everything that goes in between them

The **content of the element** is what goes between the tags

— may consist of: text, other elements, or a mix of both

Representing relational data as XML

XML is a quite popular format for exchanging relational data, because most applications use RDBMSs. In fact, there is even a standard called SQL/XML that helps with that. (More on this later).

An obvious solution would be to map each row of each relation to an element:

Movie

title	year	imdb	director
Ghostbusters	1984	7.8	Ivan Reitman
Big	1988	7.3	Penny Marshall
Lost in Translation	2003	7.8	Sofia Coppola
Wadjda	2012	8.1	Haifaa al-Mansour
Ghostbusters	2016	5.3	Paul Feig

```
...
<Movie>
  <row> <title>Ghostbusters</title>
    <year>1984</year>
    <imdb>7.8</imdb>
    <director>Ivan Reitman</director>
  </row>
  <row> <title>Big</title>
    ...
  </row>
  ...
</Movie>
...
```


XML does not suffer from the 1NF limitation: elements can be multi-valued. Thus we can “nest” a table inside another (e.g., if there is a PK/FK relationship between them):

```
...
<Movie>
  <row> <title>Ghostbusters</title>
    <year>1984</year>
    <imdb>7.8</imdb>
    <director>Ivan Reitman</director>
    <cast>
      <actor>Bill Murray</actor> <role>Dr. Venkman</role>
      <actor>Sigourney Weaver</actor> <role>Dana Barret</role>
    </cast>
  </row>
  ...
</Movie>
...
```

Attributes, data and meta-data

```
<bibliography>
  <book isbn="046509760X">
    <title>The Book of Why</title>
    <author>Pearl</author>
    <author>Mackenzie</author>
    <publisher>Basic Books</publisher>
    <price currency="CDN">45.00</price>
  </book>
</bibliography>
```

Attributes may be added to describe, identify or **link** elements

- **isbn** identifies the book
- **currency** explains that the price is in Canadian dollars

In principle, attributes should be *metadata*; but it is hard to distinguish data from metadata sometimes.

Document Order

XML is meant to encode documents, thus, unlike with tuples in a table, the ordering of the elements in a document is important.

- Think about reading your favorite book in a random order of chapters!
- XML Storage systems must preserve element ordering.

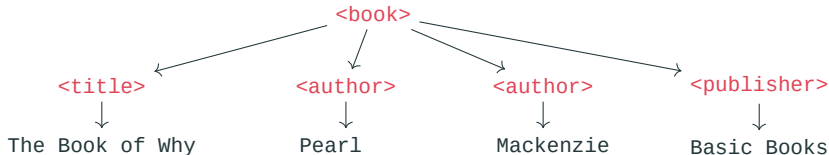
As we will see later, we can ask queries involving element order.

```
<book>
  <title>Pride And Prejudice</title>
  <chapter>
    <title>Chapter 1</title>
    <p>It is a truth universally acknowledged,
      that a single man in possession of a good
      fortune must be in want of a wife. </p>
    <p>...</p>
    ...
  </chapter>
  <chapter>
    <title>Chapter 2</title>
    <p>Mr. Bennet was among the earliest of
      those who waited on Mr. Bingley...</p>
    <p>...</p>
    ...
  </chapter>
  ...
</book>
```

XML Elements are Trees

The nesting of elements naturally induce a tree, often called the Document Object Model (DOM) tree:

```
<book>  
  <title>The Book of Why</title>  
  <author>Pearl</author>  
  <author>Mackenzie</author>  
  <publisher>Basic Books</publisher>  
</book>
```



Edges in the tree represent the **parent-child** relationship, from which we can derive other relationships: ancestor, descendant and sibling.

Constraints? Schemas?

Document Type Definitions (DTDs)

Originated from SGML³; uses **regular expressions** to specify which elements can be nested within others and simple rules to specify attributes.

XML Schema

Borrow best practices from databases and programming languages; adds support for specifying data types for the various elements.

Strictly more powerful than DTDs: you can always rewrite a DTD as a schema, but not the other way around.

³Standard Generalized Markup Language—https://en.wikipedia.org/wiki/Standard_Generalized_Markup_Language

ELEMENT rules specify what can go in an element via **regular expressions**.

```
<!DOCTYPE bibliography [  
  <!ELEMENT bibliography (book*)>  
  <!ELEMENT book (title, author+, publisher, price?)>  
  <!ATTLIST book isbn ID #REQUIRED>  
  <!ELEMENT title (#PCDATA)>  
  <!ELEMENT author (#PCDATA)>  
  <!ELEMENT publisher (#PCDATA)>  
  <!ELEMENT price (#PCDATA)>  
  <!ATTLIST price currency CDATA #IMPLIED>  

```

Each bibliography element can contain zero or more book elements and nothing else.

A book element **must** have a title, at least one author, and a publisher; it may also have a price element.

Elements title, author, publisher, and price can contain only “parsed character data” (**#PCDATA**), which is XML for “text”.

Validation of **ELEMENT** Rules

Defn: an element is valid if its *content* matches the regular expression associated with its name in the DTD.

Defn: a document is valid if all of its elements are valid.

For validation purposes, the *content* of element x is one of⁴:

- a string formed by the labels of the children of x
- **#PCDATA** if the element contains only text

Example: `<cast><actor>Bill Murray</actor></cast>`

The content of the `<cast>` element is actor, and the content of `<actor>` is **#PCDATA**.

⁴XML allows elements to have *mixed* content as well: that is, elements interspersed with text. We will ignore such content in CMPUT391 for simplicity.

```
<bibliography>
  <book>
    <title>The Book of Why</title>
    <author>Pearl</author>
    <author>Mackenzie</author>
    <publisher>Basic Books</publisher>
  </book>
</bibliography>
```

Is the element above valid?

```
<![ELEMENT bibliography (book*)>
<![ELEMENT book (title, author+, publisher, price?)>
<![ELEMENT title (#PCDATA)>
<![ELEMENT author (#PCDATA)>
<![ELEMENT publisher (#PCDATA)>
<![ELEMENT price (#PCDATA)>
```


ATTLIST rules specify whether attributes are required, and whether they can be used as identifiers.

```
<!DOCTYPE bibliography [  
  <!ELEMENT bibliography (book*)>  
  <!ELEMENT book (title, author+, publisher, price?)>  
  <!ATTLIST book isbn ID #REQUIRED>  
  <!ELEMENT title (#PCDATA)>  
  <!ELEMENT author (#PCDATA)>  
  <!ELEMENT publisher (#PCDATA)>  
  <!ELEMENT price (#PCDATA)>  
  <!ATTLIST price currency CDATA #IMPLIED>  

```

CDATA means “character data”. Attributes cannot have elements or other attributes as their value.

#REQUIRED indicates the attribute must be present; while **#IMPLIED** means the attribute is optional.

Finally, **#ID** indicates that the value of the attribute must be unique within the document.

ID and IDREF attributes

#ID and **#IDREF** attributes are “like” keys and foreign keys, but not quite, because their scope is the entire document (instead of individual tables).

```
<!DOCTYPE bibliography [  
  <!ELEMENT bibliography (book*)>  
  <!ELEMENT book (title, author+, publisher, price?)>  
  <!ATTLIST book isbn ID #REQUIRED  
                sequel IDREF #IMPLIED>  
  <!ELEMENT title (#PCDATA)>  
  <!ELEMENT author (#PCDATA)>  
  <!ELEMENT publisher (#PCDATA)>  
  <!ATTLIST publisher website ID #IMPLIED>  
  <!ELEMENT price (#PCDATA)>  
  <!ATTLIST price currency CDATA #IMPLIED>  

```

The (optional) **sequel** must match the value of some **#ID** attribute.

Validation of **ATTLIST** Rules

A **#REQUIRED** attribute must be present, otherwise the element (and the document) is invalid. An **#IMPLIED** attribute is optional.

The value of an **#ID** attribute be unique within the document, while the value of an **#IDREF** attribute must match some **#ID** attribute.

```
<!DOCTYPE bibliography [  
  ...  
  <!ATTLIST book isbn ID #REQUIRED  
                sequel IDREF #IMPLIED>  
  <!ATTLIST publisher website ID #IMPLIED>  
  <!ATTLIST price currency CDATA #IMPLIED>  

```

Note that, counter to what you would expect, we cannot specify that **sequel** must match the **isbn** of another book!

```
<bibliography>  
  <book isbn="1234">  
    ...  
    <publisher website="abc.ca">  
      ...  
    </publisher>  
  </book>  
  <book isbn="5678" sequel="1234">  
    ...  
    <publisher>...</publisher>  
  </book>  
  <book isbn="xyz" sequel="abc.ca">  
    ...  
  </book>  
</bibliography>
```

DTDs are enough for describing documents, but have many shortcomings if used to describe relational data.

- Can “simulate” unary keys (ID attributes), but there is no support for n-ary keys.
- No support for inclusion dependencies; IDREF attributes are not restricted to specific ID attributes.
- Only data type is “text”.
- Element name and “type” (regular expression on child labels) are associated globally (i.e., independently of context).

XML Schema

- In XML format.
- Element types are contextualized.
 - Ex: different book elements may have different types depending on the label of their parent.
- Defines several primitive data types (integers, strings, dates, etc.) and allows user-defined data types.
- Supports all common data types and the specification of domains (like SQL CHECK constraints)
- Supports proper keys (including n-ary) and foreign keys.
- Inheritance (extension or restriction).
- Element-type reference constraints.

(Simplified) Example XML Schema

```
<schema version="1.0" xmlns="http://www.w3.org/1999/XMLSchema">
  <element name="book">
    <complexType>
      <attribute name="isbn" type="string" use="required"/>
      <element name="title" type="string"/>

      <element name="author" minOccurs="1" maxOccurs="7" type="author_type"/>

      <element name="publisher" type="string" />

      <element ref="price" minOccurs="0" maxOccurs="1">
        <complexType>
          <simpleContent>
            <extension base="float"/>
            <attribute name="currency" type="string" use="optional"/>
          </simpleContent>
        </complexType>
      </element>
    </complexType>
  </element>
</schema>
```

XML Query Languages

Example XML document for the next few slides:

```
<bibliography>
  <book>
    <title>The Book of Why</title>
    <author>Perl</author>
    <author>Mackenzie</author>
    <publisher>Basic Books</publisher>
    <year>2018</year>
  </book>
  <book>
    <title>The Art of Computer Programming</title>
    <author>
      <first>Donald</first>
      <last>Knuth</last>
    </author>
    <published>Addison-Wesley, 1969</published>
  </book>
</bibliography>
```


Example XML document for the next few slides:



```
<bibliography>
  <book>
    <title>The Book of Why</title>
    <author>Perl</author>
    <author>Mackenzie</author>
    <publisher>Basic Books</publisher>
    <year>2018</year>
  </book>
  <book>
    <title>The Art of Computer Programming</title>
    <author>
      <first>Donald</first>
      <last>Knuth</last>
    </author>
    <published>Addison-Wesley, 1969</published>
  </book>
</bibliography>
```

(†) Elements need not have the same kind of content!

Example XML document for the next few slides:

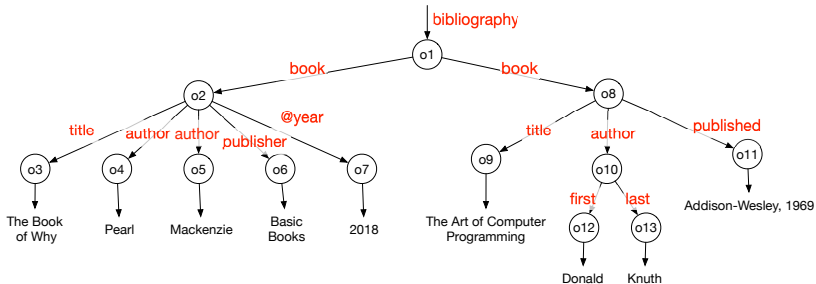
```
<bibliography>
  <book>
    <title>The Book of Why</title>
    <author>Perl</author>
    <author>Mackenzie</author>
    <publisher>Basic Books</publisher>
    <year>2018</year>
  </book>
  <book>
    <title>The Art of Computer Programming</title>
    <author>
      <first>Donald</first>
      <last>Knuth</last>
    </author>
    <published>Addison-Wesley, 1969</published>
  </book>
</bibliography>
```



- (†) Elements need not have the same kind of content!
- (†) Missing or new elements are also possible.

Data Model

XML elements (including the entire document) are modeled as **ordered labeled trees**.



XML is a **semi-structured** data format: different elements with the same label can have different types or different kinds of content.

XML Query Languages

The W3C defines (at least) two query languages for XML, which are used **together**.

XPath

- like regular expressions, but on paths over XML trees
- specifies and “returns” nodes in the tree

XQuery

- functional query language that computes an answer from input XML documents
- uses XPath to locate nodes in the input (like the `WHERE` clause in SQL) that are used to compute the answer

XPath

XPath is a language for **addressing** parts of an XML document, designed to be used by both XSLT and XPointer.

(<https://www.w3.org/TR/1999/REC-xpath-19991116/>)

- Contains approximately 200 built-in functions.
- A major element in the XML ecosystem, including XSLT, XLink, and XQuery.

Every XPath expression returns **an ordered sequence** of nodes in the XML tree.

Unless otherwise specified, the ordering of the nodes is the *document ordering*.

XPath

An XPath expression is called a **location path**:

Location Path

A location path is a **sequence** of **location steps**, separated by **/**, that identifies one or more nodes in a **document**.

Example: `doc("books.xml")/child::bibliography/child::book`

Each **Location Step**:

- Specifies a single navigational step in a path;
- Is evaluated against a single **context node**, and returns a **sequence of nodes**;
- Almost always consists of a **navigational axis** and a **node test**.

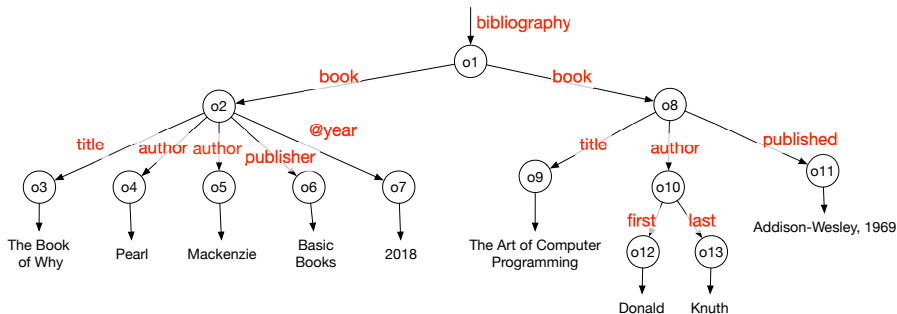
Location Steps

Formally: `axis::node test[predicate]`

- **Node tests** are optional. Often they are **name tests** (i.e., an element or attribute name).
- **Predicates** are also optional, and are used to filter nodes reachable using the axis and satisfying the name test.

Most **common axes** and their **abbreviated forms**:

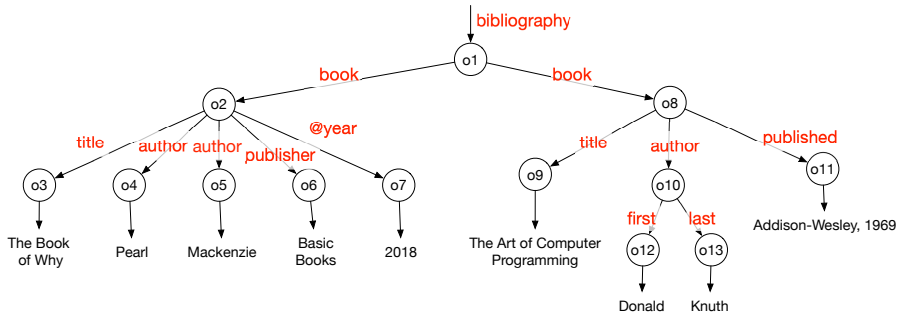
axis	abbreviation	example
child	/	<code>doc("books.xml")/bibliography</code>
descendant-or-self	//	<code>doc("books.xml")//book</code>
attribute	@	<code>//book/@year</code>
self	.	<code>//author[./last='Knuth']</code>
ancestor	..	<code>//first[..last='Knuth']</code>



/bibliography/book/@year =

/bibliography//author =

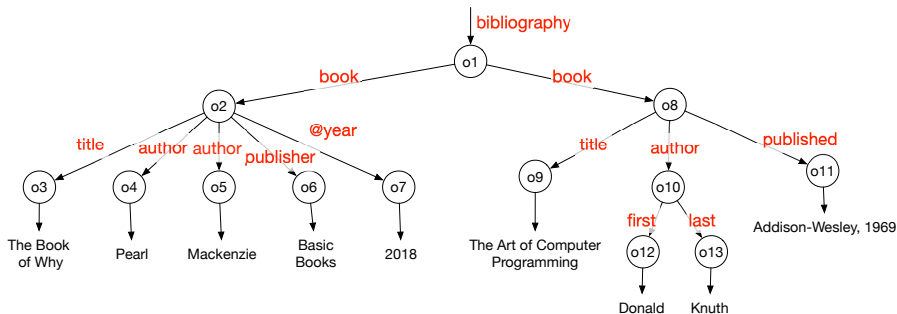
//first/..author =



`/bibliography/book/@year = (o7)`

`/bibliography//author =`

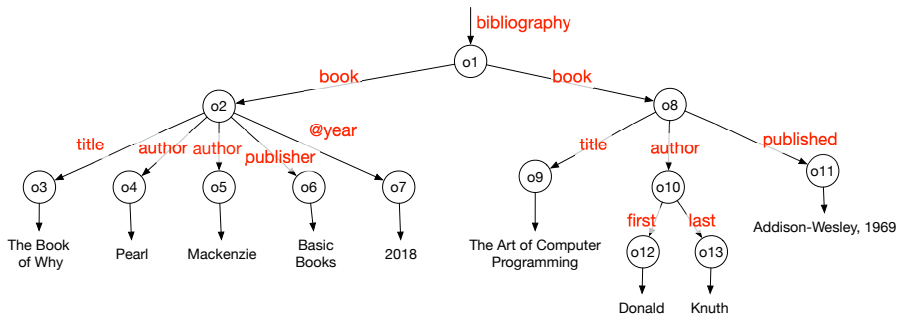
`//first/../author =`



`/bibliography/book/@year = (o7)`

`/bibliography//author = (o4, o5, o10)`

`//first/..author =`



`/bibliography/book/@year = (o7)`

`/bibliography//author = (o4, o5, o10)`

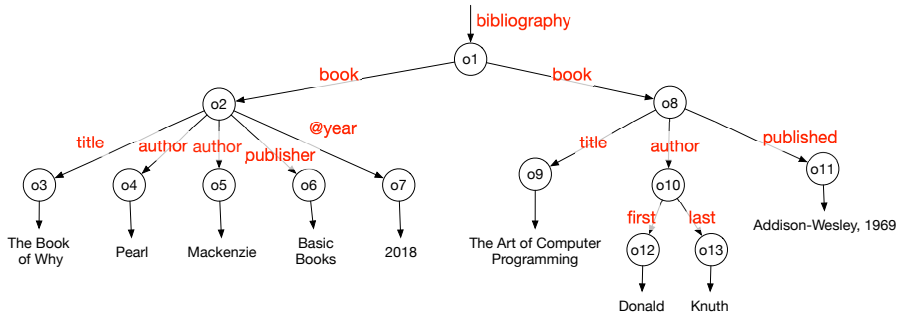
`//first/..author = (o10)`

Comparison Expressions

XPath and XQuery model XML have multiple comparators.

General Comparator	Value Comparator	Description
=	eq	Equals
!=	ne	Not equals
>	gt	Greater than
>=	ge	Greater than or equals to
<	lt	Less than
<=	le	Less than or equals to

General comparators can be used to compare values and sequences, while value comparators can only be used to compare two values.



Example: selecting books written by `Pearl`... because there can be many (i.e., a sequence) of authors in each book, we need to use the general comparator:

```
//book[./author=Pearl]
```

The comparison above evaluates to true if *at least one* item in the sequence is equal to the value `Pearl`.

Why do we need value comparators?

Why not just use general comparators?

Defensive programming

Comparisons involving sequences can lead to confusing results. If you *know* that the comparison should involve a single value (e.g., every book has only one isbn) it is best to use a value comparator.

With defensive programming, if you get an error, at least you know that the document is not structured as you expected.

More XPath

Wildcards: //person/@*

- Returns all attributes of any person.

Disjunctions: //(first|last)

- Returns all elements named first OR last.

Filters:

//person[./profession="physicist"]

- Returns person elements that contain a child called profession whose value is the string "physicist".

```
<?xml version="1.0"?>
<people>
  <person born="1912" died="1954">
    <name>
      <first>Alan</first>
      <last>Turing</last>
    </name>
    <profession>computer scientist</profession>
    <profession>mathematician</profession>
    <profession>cryptographer</profession>
  </person>
  <person born="1918" died="1988">
    <name>
      <first>Richard</first>
      <middle_initial>P</middle_initial>
      <last>Feynman</last>
    </name>
    <profession>physicist</profession>
    <hobby>Playing the bongoes</hobby>
  </person>
</people>
```

Variables in Predicates are Existentially Quantified

Logical test `//person[./profession="mathematician"]` returns person elements that contain **at least one child** called `profession` whose value is the string `"mathematician"`.

```
<person born="1912" died="1954">
  <name>
    <first>Alan</first>
    <last>Turing</last>
  </name>
  <profession>computer scientist</profession>
  <profession>mathematician</profession>
  <profession>cryptographer</profession>
</person>
```


Functions

XPath 2.0 and XQuery 1.0 share the same function library⁵

- Became a recommendation in Jan 2007.
- With approximately 200 functions.
- More were added over time.

Every function is evaluated with respect to a context node the higher-level specification in which XPath is used, such as XSLT or XPointer, decides how this context node is determined

XPath is weakly typed automatic type conversions (casting) are done whenever possible

⁵<https://www.w3.org/TR/xpath-functions-31/>

XPath defines a “Core Function Library” that every implementation **must** provide:

- **Node Set Functions** operate on properties of XML nodes
ex: `id("046509760X")/author[position()=1]` selects the first author child of the element whose **#ID** attribute equals 046509760X.
- **String Functions:** usual string manipulation found in most programming languages (sub-strings, concatenation, etc.)
- **Boolean Functions:** `and()`, `or()`, `not()` plus `boolean(x)` which converts object X into a boolean, returning:
 - **True** if X is one of: a positive number, a non-empty string, or a non-empty node set.
 - **False** otherwise.

XPath “Core Function Library” continued:

- **Number Functions** manipulate numbers and cast strings as numbers.
 - **number**(x) converts strings spelling numeric values (e.g., "123.45") into proper numbers
 - **number**(x) converts booleans into numbers (true:1, false:0)
 - **sum**() returns the sum of a sequence of numbers
 - **floor**(), **ceiling**() and **round**() work as defined in most programming languages

Programming Languages and XML

Unlike with SQL, the W3C approach with XPath/XQuery was to provide **many functions** programmers would be familiar with.

XQuery – FLOWR Expressions

Main clauses in XQuery since version 1.0:

```
[for variable bindings]  
[let variable bindings]  
[where condition(s)]  
[order by criteria]  
[return expression]
```

- **for** clause: create tuple streams
- **let** clause: bind variables to results of expressions
- **where** clause: filter tuples that don't satisfy a condition
- **order by** clause: sort the tuples in the tuple stream
- **return** clause: builds the result of the expression for each tuple in the stream

XQuery – analogy to SQL

Like with SQL and relational data, most XML queries perform three major tasks:

- (1) Specify a **scope** (i.e., identify data elements are relevant to the query)
- (2) Specify **filters** to remove undesirable elements
- (3) **Compute** a result from the elements that remain

Equivalence between SQL
and XQuery clauses.

task	SQL	XQuery
(1)	FROM	for
(2)	WHERE	where
(3)	SELECT	return

Example document – books.xml

```
<books>
  <book year="2009">
    <title>Causality</title>
    <author><last>Pearl</last><first>Judea</first></author>
    <publisher>Cambridge</publisher>
  </book>
  <book year="1999">
    <title>Foundations of Statistical Natural Language Processing</title>
    <author><last>Manning</last><first>Christopher</first></author>
    <author><last>Schütze</last><first>Hinrich</first></author>
    <publisher>MIT Press</publisher>
  </book>
  <book year="2005">
    <title>Introduction to Information Retrieval</title>
    <author><last>Manning</last><first>Christopher</first></author>
    <author><last>Raghavan</last><first>Prabhakar</first></author>
    <author><last>Schütze</last><first>Hinrich</first></author>
    <publisher>Cambridge</publisher>
  </book>
</books>
```

XQuery expressions

Prototypical simple XQuery query:

```
for $b in doc("books.xml")//book
where $b/@year eq "2009"
return $b/title
```

Evaluation algorithm

- (a) Compute path expression `doc("books.xml")//book`
- (b) For each element `$b` in the result, if `$b` satisfies the `where` clause, evaluate the `return` clause, *appending* to the final result.

XQuery expressions

Prototypical simple XQuery query:

```
for $b in doc("books.xml")//book
where $b/@year eq "2009"           <title>Causality</title>
return $b/title
```

Evaluation algorithm

- (a) Compute path expression `doc("books.xml")//book`
- (b) For each element `$b` in the result, if `$b` satisfies the `where` clause, evaluate the `return` clause, *appending* to the final result.

The **return** Clause

Like with SQL, an XQuery query can compute **new** values.

For example, we can create new XML elements that were not in the document

```
for $b in doc("books.xml")//book
where $b/@year eq "2009"
return <newElement>
    { $b/title/text() }
</newElement>
```

In XQuery, we compute the values that go inside elements (or attributes) with expressions within brackets: **{ ... }**

The **return** Clause

Like with SQL, an XQuery query can compute **new** values.

For example, we can create new XML elements that were not in the document

```
for $b in doc("books.xml")//book
where $b/@year eq "2009"
return <newElement>                                <newElement>Causality</newElement>
    { $b/title/text() }
</newElement>
```

In XQuery, we compute the values that go inside elements (or attributes) with expressions within brackets: **{ ... }**

Using the `let` Clause as “CTEs”

```
let $CUPauthors := for $b in doc("books.xml")//book
                    where $b/publisher eq "Cambridge"
                    return $b/author
for $a in $CUPauthors
return <author>{ $a/first/text() , " " , $a/last/text()}</author>
```

in XQuery, commas are used for concatenating the values of expressions:

Using the `let` Clause as “CTEs”

```
let $CUPauthors := for $b in doc("books.xml")//book
                    where $b/publisher eq "Cambridge"
                    return $b/author
for $a in $CUPauthors
return <author>{ $a/first/text() , " " , $a/last/text()}</author>
```

in XQuery, commas are used for concatenating the values of expressions:

```
<author>Judea Pearl</author>
<author>Christopher Manning</author>
<author>Prabhakar Raghavan</author>
<author>Hinrich Schütze</author>
```

Sequences

XQuery is a functional language; its basic abstraction is **iterating over ordered sequences** of data elements.

- The **let** clause **binds** sequences to variables.

```
let $books := doc("books.xml")//book
```

Think of `$books` as an array or linked list with the book elements.

- The **for** clause **binds** items in a sequence to a variable.

```
let $books := doc("books.xml")//book  
for $b in $books
```

Document Ordering!

Unless otherwise specified, the order of the items in the sequence is **the same** as in the document.

Positional Predicates

Because sequences are ordered, we can use positional predicates to filter out items.

The following query returns the title of the second book in the document:

```
for $b in doc("books.xml")//book[position() = 2]
return $b/title
```

The test above can be abbreviated as `doc(...)//book[2]`.

The following returns the last author of every book.

```
for $b in doc("books.xml")//book
return $b/author[position() = count($b/author)]
```

The test above can be abbreviated as `$b/author[last()]`.

Operations on Sequences

See: <https://www.w3.org/TR/xpath-functions-31/#sequence-functions>

Function	Meaning
<u>fn:empty</u>	Returns true if the argument is the empty sequence.
<u>fn:exists</u>	Returns true if the argument is a non-empty sequence.
<u>fn:head</u>	Returns the first item in a sequence.
<u>fn:tail</u>	Returns all but the first item in a sequence.
<u>fn:insert-before</u>	Returns a sequence constructed by inserting an item or a sequence of items at a given position within an existing sequence.
<u>fn:remove</u>	Returns a new sequence containing all the items of \$target except the item at position \$position.
<u>fn:reverse</u>	Reverses the order of items in a sequence.
<u>fn:subsequence</u>	Returns the contiguous sequence of items in the value of \$sourceSeq beginning at the position indicated by the value of \$startingLoc and continuing for the number of items indicated by the value of \$length.
<u>fn:unordered</u>	Returns the items of \$sourceSeq in an <u>implementation-dependent</u> order.

Function	Meaning
<u>fn:distinct-values</u>	Returns the values that appear in a sequence, with duplicates eliminated.
<u>fn:index-of</u>	Returns a sequence of positive integers giving the positions within the sequence \$seq of items that are equal to \$search.
<u>fn:deep-equal</u>	This function assesses whether two sequences are deep-equal to each other. To be deep-equal, they must contain items that are pairwise deep-equal; and for two items to be deep-equal, they must either be atomic values that compare equal, or nodes of the same kind, with the same name, whose children are deep-equal, or maps with matching entries, or arrays with matching members.

Example:

```
for $n in distinct-values(doc("books.xml")//publisher/text())
return <publisher>{$n}</publisher>
```


Function	Meaning
<u>fn:distinct-values</u>	Returns the values that appear in a sequence, with duplicates eliminated.
<u>fn:index-of</u>	Returns a sequence of positive integers giving the positions within the sequence \$seq of items that are equal to \$search.
<u>fn:deep-equal</u>	This function assesses whether two sequences are deep-equal to each other. To be deep-equal, they must contain items that are pairwise deep-equal; and for two items to be deep-equal, they must either be atomic values that compare equal, or nodes of the same kind, with the same name, whose children are deep-equal, or maps with matching entries, or arrays with matching members.

Example:

```
for $n in distinct-values(doc("books.xml")//publisher/text())
return <publisher>{$n}</publisher>

<publisher>Cambridge</publisher><publisher>MIT Press</publisher>
```

Why do we need `text()` in the path expression above?

Function	Meaning
----------	---------

<u>fn:count</u>	Returns the number of items in a sequence.
---------------------------------	--

<u>fn:avg</u>	Returns the average of the values in the input sequence \$arg, that is, the sum of the values divided by the number of values.
-------------------------------	--

<u>fn:max</u>	Returns a value that is equal to the highest value appearing in the input sequence.
-------------------------------	---

<u>fn:min</u>	Returns a value that is equal to the lowest value appearing in the input sequence.
-------------------------------	--

<u>fn:sum</u>	Returns a value obtained by adding together the values in \$arg.
-------------------------------	--

Example:

```
let $cnts := for $b in doc("books.xml")//book
              return count($b/author)
return round(avg($cnts), 2)
```

Function	Meaning
----------	---------

<code>fn:count</code>	Returns the number of items in a sequence.
-----------------------	--

<code>fn:avg</code>	Returns the average of the values in the input sequence <code>\$arg</code> , that is, the sum of the values divided by the number of values.
---------------------	--

<code>fn:max</code>	Returns a value that is equal to the highest value appearing in the input sequence.
---------------------	---

<code>fn:min</code>	Returns a value that is equal to the lowest value appearing in the input sequence.
---------------------	--

<code>fn:sum</code>	Returns a value obtained by adding together the values in <code>\$arg</code> .
---------------------	--

Example:

```
let $cnts := for $b in doc("books.xml")//book
              return count($b/author)
return round(avg($cnts), 2)
```

2

Comparisons Involving Sequences

Recall the **general comparators** that apply to values and sequences:

= (equals to)	!= (different than)
< (less than)	<= (less than or equal to)
> (greater than)	>= (greater than or equal to)

Existential Semantics

`$s1 OP $s2` evaluates to **true** if there exist items `x` in `$s1` and `y` in `$s2` for which `x OP y` is true.

Examples: all programs below return **true**...

<code>let \$s1 := (1,2,3)</code>	<code>let \$s1 := (1,2,3)</code>	<code>let \$s1 := (1,2,3)</code>
<code>let \$s2 := (1,2)</code>	<code>let \$s2 := (1,2)</code>	<code>let \$s2 := (1,2)</code>
<code>return \$s1 = \$s2</code>	<code>return \$s1 < \$s2</code>	<code>return \$s1 > \$s2</code>

Comparisons with sequences can be convenient...

Ex: find the titles of books that have at least one author whose first name is Prabhakar

```
for $b in doc("books.xml")//book
where $b/author/first/text() = "Prabhakar"
return $b/title
```

Returns

```
<title>Introduction to Information Retrieval</title>
```

Why? For that book, we have that

```
$b/author/first/text() ← ("Christopher", "Prabhakar", "Hinrich").
```

XQuery uses **casting** aggressively, in an attempt to simplify the queries. The following expression from the previous query compares two **sequences**:

```
$b/author/first/text()="Prabhakar"
```

XQuery casts the RHS to ("Prabhakar")

The following query (note it does not have `text()`) computes the same answer:

```
for $b in doc("books.xml")//book
where $b/author/first = "Prabhakar"
return $b/title
```

XQuery understands the RHS is a sequence of values, and casts each **<first>** element into the corresponding value.

But comparisons with sequences can also be confusing...

Ex: find the titles of books that have at least one author whose first name is Prabhakar and last name is Manning.

```
for $b in doc("books.xml")//book
where $b/author/first = "Prabhakar" and $b/author/last = "Manning"
return $b/title
```

The query “should return nothing” (as there is no author satisfying the conditions). Yet, it returns:

```
<title>Introduction to Information Retrieval</title>
```

Why? For that book, both

```
("Christopher", "Prabhakar", "Hinrich")= "Prabhakar" and  
("Manning", "Raghavan", "Schütze")= "Manning" evaluate to true.
```

Fixing the query so that it finds correctly the titles of books that have at least *one author* whose first name is Prabhakar and last name is Manning.

```
for $b in doc("books.xml")//book
where $b/author/first = "Prabhakar" and $b/author/last = "Manning"
return $b/title
```


Fixing the query so that it finds correctly the titles of books that have at least *one author* whose first name is Prabhakar and last name is Manning.

```
for $b in doc("books.xml")//book
where $b/author/first = "Prabhakar" and $b/author/last = "Manning"
return $b/title
```

```
for $b in doc("books.xml")//book
  for $a in $b/author
  where $a/first = "Prabhakar" and $a/last = "Manning"
  return $b/title
```

Nested Loops

The previous query is an example of a *correlated* nested loop:

```
for $b in doc("books.xml")//book
  for $a in $b/author
    return $a/last
```

```
<books>
  <book year="2009">
    <title>Causality</title>
    <author><last>Pearl</last><first>Judea</first></author>
    <publisher>Cambridge</publisher>
  </book>
  <book year="1999">
    <title>Foundations of Statistical Natural Language Processing
    <author><last>Manning</last><first>Christopher</first></author>
    <author><last>Schütze</last><first>Hinrich</first></author>
    <publisher>MIT Press</publisher>
  </book>
  <book year="2005">
    <title>Introduction to Information Retrieval</title>
    <author><last>Manning</last><first>Christopher</first></author>
    <author><last>Raghavan</last><first>Prabhakar</first></author>
    <author><last>Schütze</last><first>Hinrich</first></author>
    <publisher>Cambridge</publisher>
  </book>
</books>
```

Nested Loops

The previous query is an example of a *correlated* nested loop:

```
for $b in doc("books.xml")//book
  for $a in $b/author
    return $a/last
```

```
<last>Pearl</last>
<last>Manning</last>
<last>Schütze</last>
<last>Manning</last>
<last>Raghavan</last>
<last>Schütze</last>
```

```
<books>
  <book year="2009">
    <title>Causality</title>
    <author><last>Pearl</last><first>Judea</first></author>
    <publisher>Cambridge</publisher>
  </book>
  <book year="1999">
    <title>Foundations of Statistical Natural Language Processing</title>
    <author><last>Manning</last><first>Christopher</first></author>
    <author><last>Schütze</last><first>Hinrich</first></author>
    <publisher>MIT Press</publisher>
  </book>
  <book year="2005">
    <title>Introduction to Information Retrieval</title>
    <author><last>Manning</last><first>Christopher</first></author>
    <author><last>Raghavan</last><first>Prabhakar</first></author>
    <author><last>Schütze</last><first>Hinrich</first></author>
    <publisher>Cambridge</publisher>
  </book>
</books>
```

Products and Joins

We write cross products explicitly as nested loops:

```
let $s1 := (1,2,3)
let $s2 := (1,2)
for $a in $s1
  for $b in $s2
    return <pair>{ $a , ":" , $b }
```

<pair>1 : 1</pair>
<pair>1 : 2</pair>
<pair>2 : 1</pair>
<pair>2 : 2</pair>
<pair>3 : 1</pair>
<pair>3 : 2</pair>

Recall a join is a cross product followed by a selection...

```
let $s1 := (1,2,3)
let $s2 := (1,2)
for $a in $s1
  for $b in $s2
    where $a eq $b
    return <pair>{ $a , ":" , $b }
```

<pair>1 : 1</pair>
<pair>2 : 2</pair>

More Joins

Suppose we have two XML files: `books.xml` as before and `loans.xml` to keep track of books loaned out to library users.

The following query joins the two files, returning the titles of the books which are loaned and due (at the date the query executes).

```
let $due := for $h in doc("holdings.xml")//loan
  where $h/due eq current-date()
  return $h
for $b in doc("books.xml")//book
  for $h in $due
  where $h/@isbn eq $b/@isbn
  return $b/title
```

Aggregation in XQuery 1.0

XQuery V1.0 does not have an explicit clause for aggregation, unlike SQL. Aggregation can still be accomplished through nested loops.

Example: counting the books per publisher.

```
let $publishers := distinct-values(doc("books.xml")//publisher/text())
for $p in $publishers
  let $books := doc("books.xml")//book[./publisher/text() eq $p ]
  return <book_counts><publisher>{$p}</publisher>
        <count>{count($books)}</count>
        </book_counts>
```

However, developers missed the familiar construct from SQL.

From an implementation point of view, having an explicit clause for aggregation would allow XQuery implementers to optimize that operation as well.

Aggregation in XQuery 1.1

XQuery 1.1 brought new features, many of which were familiar to SQL developers, including an **group by** clause.

Example: counting the books per publisher."

```
for $b in doc("books.xml")//book
let $p := $b/publisher
group by $p
return <book_counts>
    <publisher>{$p}</publisher>
    <count>{count($b)}</count>
</book_counts>
```

```
<book_counts>
  <publisher>Cambridge</publisher>
  <count>2</count>
</book_counts>
<book_counts>
  <publisher>MIT Press</publisher>
  <count>1</count>
</book_counts>
```

Multiple attributes and multiple *sequence* functions can be used in the aggregation.

Warning: there must be a 1-1 relationship between the elements being aggregated and the elements defining the groups!

This works because each book has a single publisher (i.e., a single group).

If there were multiple `$p` for a single `$b` you would have to “un-nest” those pairs first:

```
for $b in doc("books.xml")//book
let $p := $b/publisher
group by $p
return <book_counts>
    <publisher>{$p}</publisher>
    <count>
        {count($b)}
    </count>
</book_counts>
```

```
let $pairs := for $b in doc("books.xml")//book
               for $p in $b/publisher
               return <pair>{$b,$p}</pair>
for $pair in $pairs
let $p := $pair/publisher
group by $p
...
```


Order By

The **order by** clause sorts elements in a sequence.

Example

```
let $lnames := distinct-values(doc("books.xml")//author/last/text())
for $n in $lnames
order by $n ascending
return <name>{ $n }</name>
```

Returns

Order By

The **order by** clause sorts elements in a sequence.

Example

```
let $lnames := distinct-values(doc("books.xml")//author/last/text())
for $n in $lnames
order by $n ascending
return <name>{ $n }</name>
```

Returns

```
<name>Manning</name>
<name>Pearl</name>
<name>Raghavan</name>
<name>Schütze</name>
```

Type Conversion

Unless otherwise specified by an XML Schema, the content of every element and/or attribute in a document is of type **string**.

Like with SQL, we can cast string types to numbers⁶:

```
for $b in doc("books.xml")//book
where xs:integer($b/@year) gt 2000
return $b
```

In fact, XQuery has a complex and rich type system.

⁶<https://www.w3.org/TR/xpath-functions-31/#casting-to-numeric>

Quantification in XQuery

XQuery designers made explicit existential and universal quantification part of the language.

Existential quantification is the **default**, so the two queries below are equivalent.

```
for $b in doc("books.xml")//book
where $b/author/first = "Prabhakar"
return $b/title
```

```
for $b in doc("books.xml")//book
where some $f in $b/author/first satisfies ($f eq "Prabhakar")
return $b/title
```

Quantification in XQuery

Universal quantification is also written explicitly:

Ex: which publishers have published all books by Christopher Manning?

```
let $books := doc("books.xml")//book
let $CMbooks := for $b in $books
    where some $a in $b/author satisfies
        ($a/first/text() eq "Christopher" and
         $a/last/text() eq "Manning")
    return $b

for $p in $books/publisher/text()
where every $CMPublisher in $CMbooks/publisher satisfies
    $p = $CMPublisher
return $p
```

Ex: Which authors co-wrote all books with Christopher Manning?

```
let $books := doc("books.xml")//book
let $CMbooks := for $b in $books
    where some $a in $b/author satisfies
        ($a/first/text() eq "Christopher" and
         $a/last/text() eq "Manning")
    return $b

for $a in $CMbooks/author
where every $b in $CMbooks satisfies
    some $coauthor in $b/author satisfies $a = $coauthor
return $a
```

What else?

A lot more...

- **User-defined functions**. XQuery is a fully functional language, and allows users to define their own functions in XQuery itself.
- **Recursion**. XQuery allows recursive function calls, in a way that is familiar to all programmers.
- Familiar **programming constructs** (e.g., if-then-else).
- A very rich **standard library**, sufficient for a wide range of applications.
- Support for **updates** to documents.

XQuery is powerful enough to implement virtually anything that a Web service provider might need.

XML Support in DBMSs

Approaches for storing XML

- File System: keep documents as text files.
- Native XML Store.
 - Implement persistent data structures to represent the XML (DOM) tree.
- Inside a Relational DBMS: re-use existing storage, indexing, (and querying) systems.
 - String or BLOB attribute.
 - *XML Mappings*.
 - ★ Schema-based.
 - ★ Structure-based.

XML documents in the File System

- Advantages:
 - Simplicity.
 - Re-using existing I/O layer (and file-level locking) provided by the Operating System.
- Disadvantages:
 - Document (and data therein) is exposed to **any user** and/or **any tool** (e.g., a text editor) that has access to the directory where the documents are stored.
 - Duplicated efforts: logging, concurrency control, back-ups, security, etc. are done outside the DBMSs.
 - Hard to coordinate queries that involve relational (in-DBMS) and XML (out-of-DBMS) data.

Native XML Stores

These systems are tailored for working with XML documents of any size (typically, they support up to a few GB per document), supporting queries and update natively.

In a sense, these are highly I/O optimized implementations of tree data-structures.

Some systems listed on Wikipedia (circa 2019):

Name	License	Native Language	XQuery 3.1	XQuery 3.0	XQuery 1.0	XQuery Update	XQuery Full Text	EXPath Extensions	EXQuery Extensions	XSLT 2.0
BaseX	BSD	Java	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes
eXist	GNU LGPL	Java	Partial	Partial	Yes	Proprietary	Proprietary	Yes	Yes	Yes
MarkLogic Server	Commercial	C++	No	Partial	Yes	Proprietary	Proprietary	No	No	Yes
OpenText xDB	Commercial	Java	Partial	Partial	Yes	Yes	Yes	No	No	No
Oracle Berkeley DB XML	Commercial									
Qizx	Commercial	Java	No	No	Yes	Yes	Yes	No	No	Yes
Sedna	Apache License 2.0									

Native XML Stores

Full fledged native stores combine the functionality of a file server (e.g., WebDAV support) and a Web server (HTTP support) with that of a database management system (e.g., command-line SQL and API support for queries and updates, indexing, transaction management, etc.).

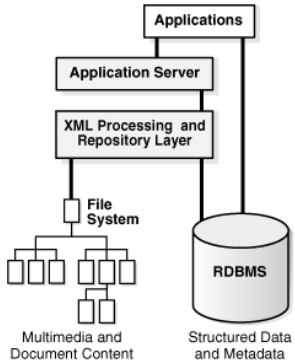
For commercial reasons, XML Stores must also support SQL systems to be successful.

Some strategies:

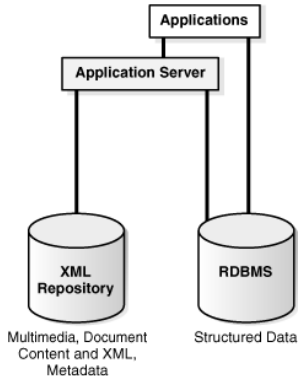
- Exporting XML data as relational views that can be read and modified by an external application.
- Offering native SQL support.

Oracle XML DB

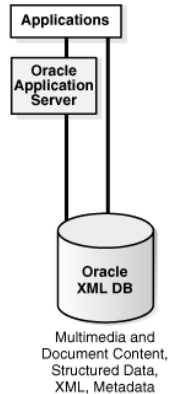
Non-Native XML Processing



Separate Data and Content Servers

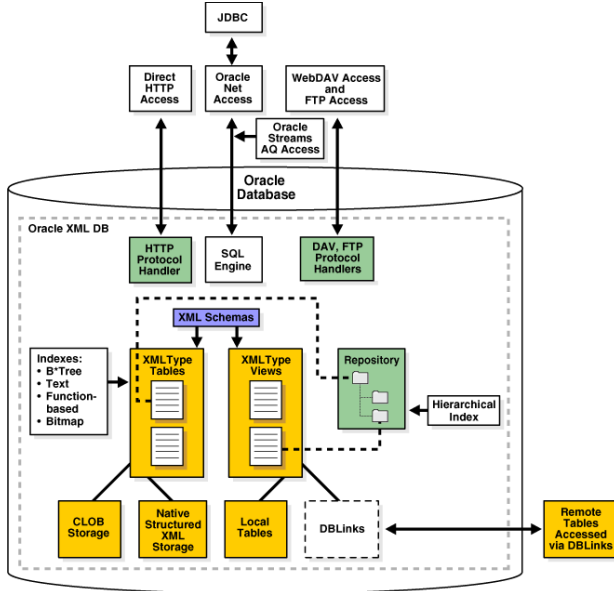


Oracle XML DB



Source [https:](https://docs.oracle.com/cd/B19306_01/appdev.102/b14259/xdb01int.htm)

[//docs.oracle.com/cd/B19306_01/appdev.102/b14259/xdb01int.htm](https://docs.oracle.com/cd/B19306_01/appdev.102/b14259/xdb01int.htm)



Source https://docs.oracle.com/cd/B19306_01/appdev.102/b14259/xdm01int.htm

https://docs.oracle.com/cd/B19306_01/appdev.102/b14259/xdm01int.htm

BaseX was started by Christian Grün at the University of Konstanz in 2005. In 2007, BaseX went open source and has been BSD-licensed since then.

Fully developed in Java.

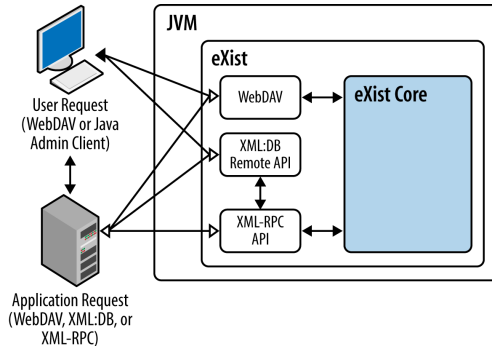
Supports XPath, XQuery 3.1, WebDAV, and various API/connector protocols.

Concurrency control:

- Queries run in parallel.
- Updates run with locking.

Sources <http://basex.org> and <https://en.wikipedia.org/wiki/BaseX>

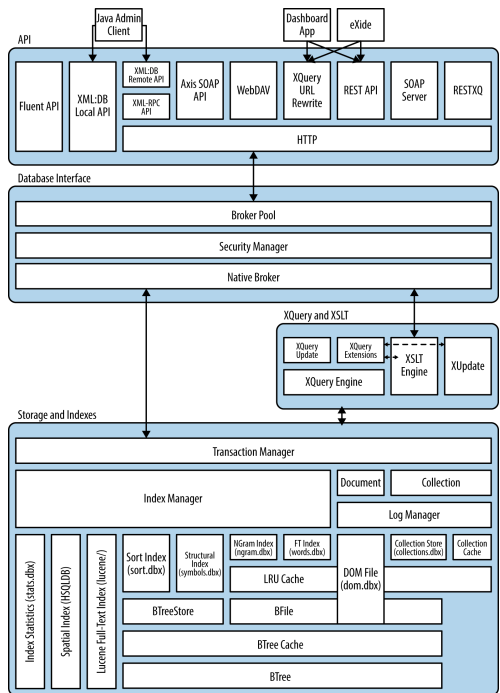
eXist is an open-source **client/server** native XML database manager supporting all popular XML protocols, XQuery, indexing, updates, proper concurrency control, **triggers**, and more!



Source: eXist, by Retter and Siegel, O'Reilly 2014

<https://learning.oreilly.com/library/view/exist/9781449337094/>

Source: eXist, by
Retter and Siegel,
O'Reilly 2014



Native XML Stores

Native XML stores are a form of NoSQL that has gained much popularity since the mid-2000s, and can be used in support of document stores and Web Services.

- Advantages:
 - Can implement optimizations specific for XML and XQuery.
 - Single application development platform.
 - “Single Source of Truth” for XML data: avoid fragmenting or mapping data into possibly inconsistent pieces.
- Disadvantages:
 - Might not integrate well with existing legacy applications.

RDBMS support for XML

With the rise of native XML database management systems, many applications were ported away from RDBMSs.

Relational vendors responded by offering support for XML within their relational products, which came in two forms:

- An **XML data type** which can be used to store (**natively**) XML documents as attributes of a regular tuple.
- **XML Mappings**, in which XML data and queries are **translated** into relational data and queries.

An industry standard, SQL/XML⁷ was defined with extensions to relational systems to handle XML content and queries.

⁷<https://en.wikipedia.org/wiki/SQL/XML>

XML Data Type

With this approach, XML content is stored as **an attribute of a tuple** with the XML data type.

Example from the IBM DB2 documentation:

```
CREATE TABLE Customer (Cid BIGINT NOT NULL PRIMARY KEY,  
                        Info XML,  
                        History XML)
```

SQL is still used as the main language for querying the database, and for inserting and modifying data.

To manipulate the XML content, we use either custom SQL functions that XPath/XQuery or SQL/XML extensions.

Inserting XML Content

Special API calls are used to insert XML content.

Example from the IBM DB2 documentation:

```
PreparedStatement insertStmt = null;

int cid = 12345; // some customer id
String sqls = "INSERT INTO Customer (Cid, Info) VALUES (?, ?)";

java.sql.Connection conn = ... //initialize connection

insertStmt = conn.prepareStatement(sqls);
insertStmt.setInt(1, cid);

File file = new File("c6.xml");

insertStmt.setBinaryStream(2, new FileInputStream(file), (int)file.length());
insertStmt.executeUpdate();
```

Querying XML Data in an XML Column

Example from the IBM DB2 documentation:

Suppose the XML document to the right is inserted into the **INFO** attribute of some tuple.

```
<customerinfo Cid="12345">
  <name>Kathy Smith</name>
  <addr country="Canada">
    <street>5 Rosewood</street>
    <city>Toronto</city>
    <prov-state>Ontario</prov-state>
    <pcode-zip>M6W 1E6</pcode-zip>
  </addr>
  <phone type="work">416-555-1358</phone>
</customerinfo>
```

We can query that document by invoking XQuery via an SQL extension:

```
SELECT XMLQUERY ("document{<phonelist>{$doc/customerinfo/phone}</phonelist>}")
  PASSING INFO AS "doc")
FROM CUSTOMER
WHERE Cid=12345
```

Result:

```
<phonelist><phone type="work">416-555-1358</phone></phonelist>
```

Exporting relational data as XML

PostgreSQL SQL/XML (on the movies database):

```
SELECT XMLElement(NAME "Movies",
    XMLAGG(
        XMLElement (NAME "movie",
            XMLElement (NAME "title", m.title),
            XMLElement (NAME "year", m.year),
            XMLElement (NAME "cast",
                (SELECT XMLAGG(
                    XMLElement(NAME "actor",
                        XMLAttributes(c.role AS "role"), c.actor))
                FROM movies.Cast c
                WHERE c.title=m.title and c.year=m.year)
            ) -- cast element
        ) -- movie element
    ) -- XMLAGG
) -- Movies element
FROM movies.Movie m
WHERE EXISTS(
    SELECT *
    FROM movies.Cast
    WHERE actor = 'Bill Murray' AND title=m.title AND year=m.year
);
```

Result:

```
<Movies>
  <movie>
    <title>Lost in Translation </title>
    <year>2003</year>
    <cast>
      <actor role="Bob Harris">Bill Murray </actor>
    </cast>
  </movie>
  <movie>
    <title>Ghostbusters </title>
    <year>1984</year>
    <cast>
      <actor role="Dr. Venkman">Bill Murray </actor>
      <actor role="Dana Barret">Sigourney Weaver </actor>
    </cast>
  </movie>
</Movies>
```


SQL/XML: summary

- Advantages:
 - Industry standard supported by many vendors and likely to remain relevant for industry.
 - Single interface for querying relational and XML data.
 - Very practical for **exporting relational data as XML**.
- Disadvantages:
 - Fairly complex standard.
 - XQuery and SQL are not fully compatible.
 - Very hard to optimize.

Storing XML via Mappings

The main motivation behind developing XML mappings was to leverage existing and highly optimized relational machinery instead of developing everything new again, for XML.

(Recall that XML documents are trees, and that we can represent arbitrary graphs (including trees) inside a relational database.)

mapping = translation

With this approach, we:

- **Translate** XML data into **pure relational data**.
- **Translate** XQuery queries into **pure SQL**.

There are two kinds of mappings: based on the tree structure of the data, and based on the tags.

Semantic Mappings

Semantic Mappings are intended for the cases when XML data comes with a schema (or DTD). The idea is to use “**mapping patterns**” to handle common constructs in a schema (or DTD).

Examples:

Pattern **a (... , b* , ...)**

- (1) Use separate tables for **a** and **b**.
- (2) Define a foreign key from **b.id** into **a.id**.

Pattern **a (... , b+ , ...)**

- (1) Do as in the Pattern **a (b*)**.
- (2) Define trigger to make sure a **b** is always present for every **a**.

Pattern **a (... b?, ...)**

- (1) Add an attribute **b** to the table for **a**.

Pattern **a (... b, ...)**

- (1) Add an attribute **b** to the table for **a**.
- (2) Define the attribute to be **required** (i.e., **NOT NULL**).

Dealing with XML attributes can be done in a similar way (i.e., they become “SQL attributes” of the table that stores the corresponding XML element).

Database Constraints

To enforce the restrictions on **#ID** and **#IDREF** attributes, we need to use **triggers**:

- **Before inserting** a tuple with an **#ID** attribute, check that no other such attribute (in any of the tables with such attributes) has the same value as the one in the *new* tuple.
- **Before inserting** a tuple with an **#IDREF** attribute, check that the value in the *new* tuple exists in the database (in any of the tables with **#ID** attributes).
- **Before deleting** a tuple with an **#ID** attribute, check that no tuple representing an **#IDREF** attribute (in any of the tables with such attributes) has the same value as the one in the *old* tuple.

Document Ordering Constraints

XML elements are **ordered**, so the XML storage system must preserve the ordering. Ordering is an issue for elements that can appear multiple times.

Essentially, we need to record the order of those elements appearing inside (b*) or (b+) rules.

More triggers needed!

The order of the elements **must** be updated as the document is modified!

- **Global ordering**: assign elements their actual document order.
- **Local ordering**: assign elements an order relative to their parents.

Global ordering

```
1<bibliography>
2  <book isbn="052189560X">
3    <title>Causality</title>
4    <author>Judea Pearl</author>
5    <publisher>Cambridge</publisher>
6  </book>
7  <book isbn="0262133601">
8    <title>Foundations of Statistical Inference</title>
9    <author>Christopher Manning</author>
10   <author>Hinrich Schütze</author>
11   <publisher>MIT Press</publisher>
12 </book>
13 <book isbn="0521865719">
14   <title>Introduction to Information Retrieval</title>
15   <author>Christopher Manning</author>
16   <author>Prabhakar Raghavan</author>
17   <author>Hinrich Schütze</author>
18   <publisher>Cambridge</publisher>
19 </book>
20 </bibliography>
```

Local ordering

```
1<bibliography>
1.1 <book isbn="052189560X">
1.1.1 <title>Causality</title>
1.1.2 <author>Judea Pearl</author>
1.1.3 <publisher>Cambridge</publisher>
1.2 <book isbn="0262133601">
1.2.1 <title>Foundations of Statistical Inference</title>
1.2.2 <author>Christopher Manning</author>
1.2.3 <author>Hinrich Schütze</author>
1.2.4 <publisher>MIT Press</publisher>
1.3 <book isbn="0521865719">
1.3.1 <title>Introduction to Information Retrieval</title>
1.3.2 <author>Christopher Manning</author>
1.3.3 <author>Prabhakar Raghavan</author>
1.3.4 <author>Hinrich Schütze</author>
1.3.5 <publisher>Cambridge</publisher>
20 </bibliography>
```

Exercise

Define a mapping for this DTD:

```
<!DOCTYPE bibliography [  
  <!ELEMENT bibliography (book*)>  
  <!ELEMENT book (title, author+, publisher, price?)>  
  <!ATTLIST book isbn ID #REQUIRED  
                sequel IDREF #IMPLIED>  
  <!ELEMENT title (#PCDATA)>  
  <!ELEMENT author (#PCDATA)>  
  <!ELEMENT publisher (#PCDATA)>  
  <!ATTLIST publisher website ID #IMPLIED>  
  <!ELEMENT price (#PCDATA)>  
  <!ATTLIST price currency CDATA #IMPLIED>  

```


Exercise

```
CREATE TABLE bibliography (elementID INTEGER, deweyOrdinal TEXT,  
    PRIMARY KEY (elementID));
```

```
CREATE TABLE book (elementID INTEGER, parentID INTEGER,  
    deweyOrdinal TEXT, title TEXT, publisher TEXT NOT NULL,  
    price TEXT, isbn TEXT NOT NULL, sequel TEXT,  
    publisher_website TEXT, price_currency TEXT,  
    PRIMARY KEY(elementID),  
    FOREIGN KEY (parentID) REFERENCES bibliography(elementID));
```

```
CREATE TABLE author (elementID INTEGER, parentID INTEGER,  
    PCDATA TEXT, deweyOrdinal TEXT,  
    PRIMARY KEY (elementID),  
    FOREIGN KEY parentID REFERENCES book(elementID));
```

Of course, we also need the **triggers** to ensure: (1) every book has at least one author, (2) the ordering of authors is consistent, and (3) **#ID** and **#IDREF** are consistent.

Mapping XPath to SQL

Mapping XPath expressions to SQL is done in two ways:
“navigating” the XML tree, and computing XML results.

Path Expressions become Joins

Ex: finding all element (ids) reachable from

`//book[title="XYZ"]/author:`

- (1) Find `elementIDs` of books with `title="XYZ"`.
- (2) Join that with **SELECT * FROM** `author`.

But How to Materialize the Results as XML Elements??

- Very hard: SQL computes “flat” tuples instead of hierarchically organized elements!
- **SQL/XML to the rescue**: use the functions to create elements and attributes.

Schema Mappings – Summary

In a sense, schema mappings work best (or only?) for data that is fairly structured to begin with. In other words, it works for relational data and not for documents.

- Advantages:

- Reuses relational engines and SQL.

- Disadvantages:

- Requires a fairly regular DTD or XML schema. Does not work for truly unstructured data.
 - Works for a subset of XQuery only (e.g., it can't handle positional predicates easily).

Structural Mappings

Structural mappings are not based on tag names tags (i.e., the semantics of the data). Instead, they use a generic relational schema that can represent any ordered labeled tree.

node (elementID, parentID, type, label, value, ordinal)

Triggers are used to
enforce all XML
constraints.

Example:

```
1    <bibliography>
1.1  <book isbn="052189560X">
1.1.2 <title>Causality</title>
1.1.3 <author>Judea Pearl</author>
1.1.4 <publisher>Cambridge</publisher>
      </book>
    </bibliography>
```

node

nodeID	parentID	type	label	value	ordinal
1	NULL	element	bibliography	NULL	1
2	1	element	book	NULL	1.1
3	2	attribute	isbn	052189560X	1.1.1
4	2	element	title	Causality	1.1.2
5	2	element	author	Judea Pearl	1.1.3
6	2	element	publisher	Cambridge	1.1.4

Query Processing

With structural mappings, query processing is straightforward (albeit slow):

Every step in a path expression becomes an “individual” SQL query over the node relation. Consecutive steps, effectively, become self-joins.

Ex: `//book[title="XYZ"]/author/text()`

```
SELECT n2.value
FROM node n1, node n2
WHERE n2.parentID = n1.nodeID AND n2.label="author" AND
    n1.label="book" AND EXISTS (SELECT *
                                FROM node n3
                                WHERE n3.parentID = n1.elementID AND
                                    n3.label="title" AND
                                    n3.value="XYZ")
```

Structural Mappings

- Advantages:
 - Clean relational schema that works for all XML documents, even those without DTDs or schemas.
 - Reuses existing relational technology; no need for new tools.
- Disadvantages:
 - SQL query results are “flat” tuples, so we still need SQL/XML to compute trees using the values in the tuples as PCDATA.
 - Translated XQuery/XPath queries boil down to many self-joins on the `node` table, which can be expensive.

API Support for Application Development

Support for Application Development

In the relational world, applications use a database either through **embedded SQL**, native use of proprietary libraries, or via an API such as ODBC⁸ (Open Database Connectivity).

XML systems are nearing maturity, and some of the APIs are becoming standard.

Name ↕	XQJ ↕	XML:DB ↕	RESTful ↕	RESTXQ ↕	WebDAV ↕
BaseX	Yes	Yes	Yes	Yes	Yes
eXist	Yes	Yes	Yes	Yes	Yes
MarkLogic Server	Yes	No	Yes	Yes	Yes
Qizx	No	No	Yes	No	No
Sedna	Yes	Yes	No	No	No

Unlike with relational data, however, XML systems must also expose them as documents which can be opened, edited, and saved by XML (file) editors.

⁸https://en.wikipedia.org/wiki/Open_Database_Connectivity

XQJ: XQuery API for Java

XQJ was developed by the Java community and backed by major vendors (including Oracle and IBM); it became a standard in 2009.

```
// Create a new connection to an XML database
XQConnection conn = vendorDataSource.getConnection("myUser", "myPassword");

XQExpression expr = conn.createExpression(); // reusable XQuery Expression

XQResultSequence result = expr.executeQuery(
    "for $n in fn:collection('catalog')//item " +
    "return fn:data($n/name)"); // execute an XQuery expression

// Process the result sequence iteratively
while (result.next()) {
    // Print the current item in the sequence
    System.out.println("Product name: " + result.getItemAsString(null));
}

// Free all resources created by the connection
conn.close();
```

Parameterized XQuery

As with SQL, programs can pass parameters to an XQuery processor at runtime, via **external variables** in the query.

```
XQExpression expr = conn.createExpression();

// The XQuery expression to be executed
String es = "declare variable $x as xs:integer external;" +
            " for $n in fn:collection('catalog')//item" +
            " where $n/price <= $x" +
            " return fn:data($n/name)";

// Bind a value (21) to an external variable with the QName x
expr.bindInt(new QName("x"), 21, null);

// Execute the XQuery expression
XQueryResultSequence result = expr.executeQuery(es);

while (result.next()) {
    // Process the result ...
}
```

Although not the only XQuery API out there, XQJ⁹ is well supported.

- **File-based XQuery Processors**: Saxon XSLT and XQuery processor, [Zorba](#), MXQuery, Oracle XQuery Processor.
- **Native Systems**: MarkLogic, eXist, BaseX, Sedna, Oracle XDB, Tamino, TigerLogic, ...
- **Relational Systems**: Oracle DB (Not XDB), IBM DB2, Microsoft SQL Server, Sybase ASE, Informix, MySQL, [PostgreSQL](#)

Supporting= relational systems is not easy, because each implements XML in a different way.

⁹https://en.wikipedia.org/wiki/XQuery_API_for_Java

ACID Transactions with XQJ

From their manual (coloring added):

*By default a connection operates in auto-commit mode, which means that **each XQuery is executed and committed in an individual transaction**. If auto-commit mode is disabled, a transaction must be ended explicitly by the application calling `commit()` or `rollback()`.*

An `XQConnection` object can be created on top of an existing `JDBC` connection. If an `XQConnection` is created on top of the `JDBC` connection, it inherits the transaction context from the `JDBC` connection.