Module 3

XML Processing
(XPath, XQuery, XUpdate)
Part 3: XQuery

Roadmap for XQuery

- Introduction/ Examples
- Use Case Scenarios
- XQuery Environment+Concepts
- XQuery Expressions
- Evaluation

Why XQuery ?

- Why a "*query" language* for XML ?
  - Need to process XML data
  - Preserve logical/physical data independence
    - The semantics is described in terms of an *abstract data model*, independent of the physical data storage
  - Declarative programming
    - Such programs should describe the "*what*", not the "*how"*

Why XQuery ?

- Why a *native* query language ? Why not SQL ?
  - We need to deal with the *specificities* of XML (hierarchical, ordered, textual, potentially schema-less structure)
- Why another XML processing language ? Why not XSLT?
  - The template nature of XSLT was not appealing to the database people. Not declarative enough.

What is XQuery ?

- A programming language that can express arbitrary XML to XML data transformations
  - Logical/physical data independence
  - "Declarative"
  - "High level"
  - "Side-effect free"
  - "Strongly typed" language
- "An expression language for XML."
- Commonalities with *functional* programming, *imperative* programming and *query* languages
- The "*query*" part might be a misnomer (***)

XQuery Use Case Scenarios

- XML transformation language in Web Services
  - Large and very complex queries
  - Input message + external data sources
  - Small and medium size data sets (*x*K -> *x*M)
  - Transient and streaming data (no indexes)
  - With or without schema validation
- XML message brokers
  - Simple path expressions, single input message
  - Small data sets
  - Transient and streaming data (no indexes)
  - Mostly non schema validated data

XQuery Use Case Scenarios

- Semantic data verification
  - Mostly messages
  - Potentially complex (but small) queries
  - Streaming and multiquery optimization required
- Data Integration
  - Complex but smaller queries (FLOWRs, aggregates, constructors)
  - Large, persistent, external data repositories
  - Dynamic data (via Web Services invocations)

XQuery Use Case Scenarios

- Large volumes of centralized XML data
  - Logs and archives
  - Complex queries (statistics, analytics)
  - Mostly read only
- Large content repositories
  - Large volume of data (books, manuals, etc)
  - With or without schema validation
  - Full text essential, update required

XQuery Usage Scenarios (ctd.)

- Large volumes of distributed textual data
  - XML search engines
  - High volume of data sources
  - Full text, semantic search crucial
- RSS data (Blogs, but also other sources)
  - High number of input data channels
  - Data is pushed, not pulled
  - Structure of the data very simple, each item bounded size
  - Aggregators using mostly full-text search

XQuery usage scenarios…

- Content re-purposing
  - E.g. customized books and articles
  - E.g. enterprise customized engineering documentation (product requirements, specs, etc)
- Streamline automatic processing
  - E.g. the creation of the W3C specifications
    - From the *same* XML document we generate automatically the XQuery, Xpath 2.0, Function Libraries specifications, plus the Javacc code that implements the XQuery parser, plus the tests that correctly test the grammar. All those are XQuery views of the same XML document !
- (Ajax-style) dynamic Web pages
  - Xquery is a better way to manipulate the XML of the Web pages then Javascript
- Re-programming the Web /scripting the Web /mashups

Examples of XQuery – Ich bin auch ein XQuery

- `1`
- `1+2`
- `"Hello World"`
- `1,2,3`
-
```
<book year="1967" >
      <title>The politics of experience</title>
      <author>R.D. Laing</author>
</book>
```

Examples of XQuery (ctd.)

- `/bib/book`
- `//book[@year > 1990]/author[2]`
-
```
for $b in //book
where $b/@year
return $b/author[2]
```
-
```
let $x := ( 1, 2, 3 )
   return count($x)
```

Some more examples of XQuery

- ```
  for $b in //book,
              $p in //publisher
      where $b/publisher = $p/name
      return ( $b/title , $p/address)
  ```
- ```
  if ( $book/@year <1980 )
  then <old>{$x/title}</old>
  else <new>{$x/title}</new>
  ```

XQuery Implementations

- Relational databases
  - Oracle 11g, SQLServer 2008, DB2 Viper
- Middleware
  - Oracle, DataDirect, BEA WebLogic
- DataIntegration
  - BEA AquaLogic
- Commercial XML database
  - MarkLogic
- Open source XML databases
  - BerkeleyDB, eXist, Sedna, BaseX
- Open source XQuery processor (no persistent store)
  - Saxon, MXQuery, Zorba
- XQuery editors, debuggers
  - StylusStudio, oXygen

Overall more than 50 – see W3C XQuery pages

Recommended for Project

- Zorba: www.zorba-xquery.com
  - Open source XQuery engine in C++
  - Great Web interface to try out queries
  - Not enough to build an app
- MXQuery: www.mxquery.org
  - Open source XQuery engine in Java
  - Additional packages (Xpages) to build apps
  - Support for different platforms: mobile, browser, …
- Sausalito: www.28msec.com
  - All you need: XQuery apps in the cloud + tools
- XQDT: www.xqdt.org
  - Eclipse Plug-in; works with all the above

Concepts of XQuery

- Declarative/Functional: No execution order!
- Document Order: all nodes are in "textual order"
- Node Identity: all nodes can be uniquely identified
- Atomization
- Effective Boolean Value
- Type system

Atomization

- Motivation: how to handle *<a>1+1*?
- fn:data(item*) -> xs:anyAtomicType*
- Extracting the "value" of a node, or returning the atomic value
- Implicitly applied:
  - Arithmetic expressions
  - Comparison expressions
  - Function calls and returns
  - Cast expressions
  - Constructor expressions for various kinds of nodes
  - `order by` clauses in FLWOR expressions
- Examples:
  - `fn:data(1) = 1`
  - `fn:data(<a>2</a>) ="2"`
  - `fn:data(<a><b>1</b><b>2</b></a>) = "12"`

Effective Boolean Value

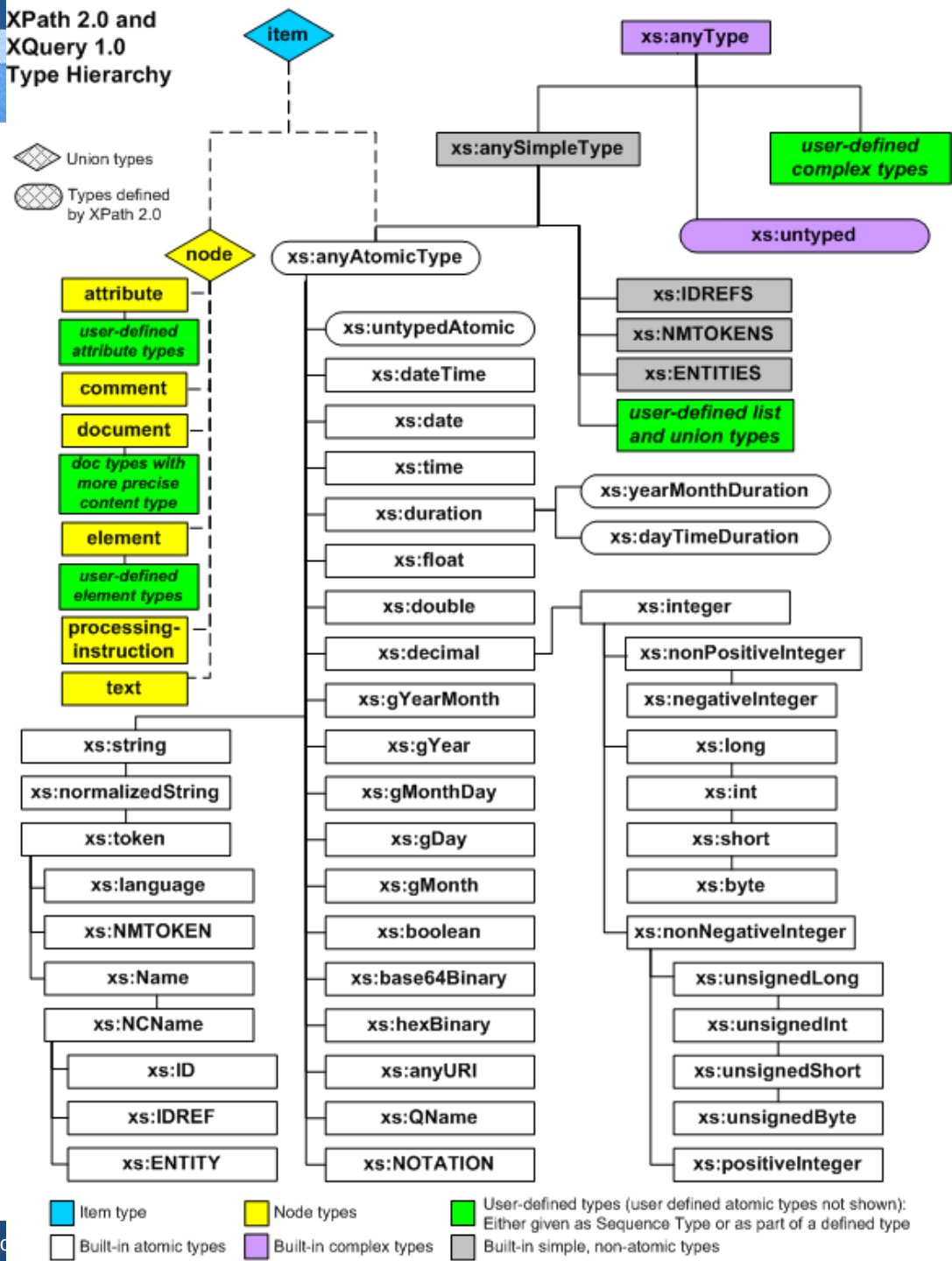
- What is the boolean interpretation of *"" or (<a/>, 1)* ?
- Needed to integrate XPath 1.0 semantics/existential qualification
- Implicit application of `fn:boolean()` to data
- Rules to compute:
  - if (), "", NaN, 0 => `false`
  - if the operand is of type xs:boolean, return it;
  - If Sequence with first item a node, => `true`
  - Non-Empty-String, Number <> 0 => `true`
  - else raise an error

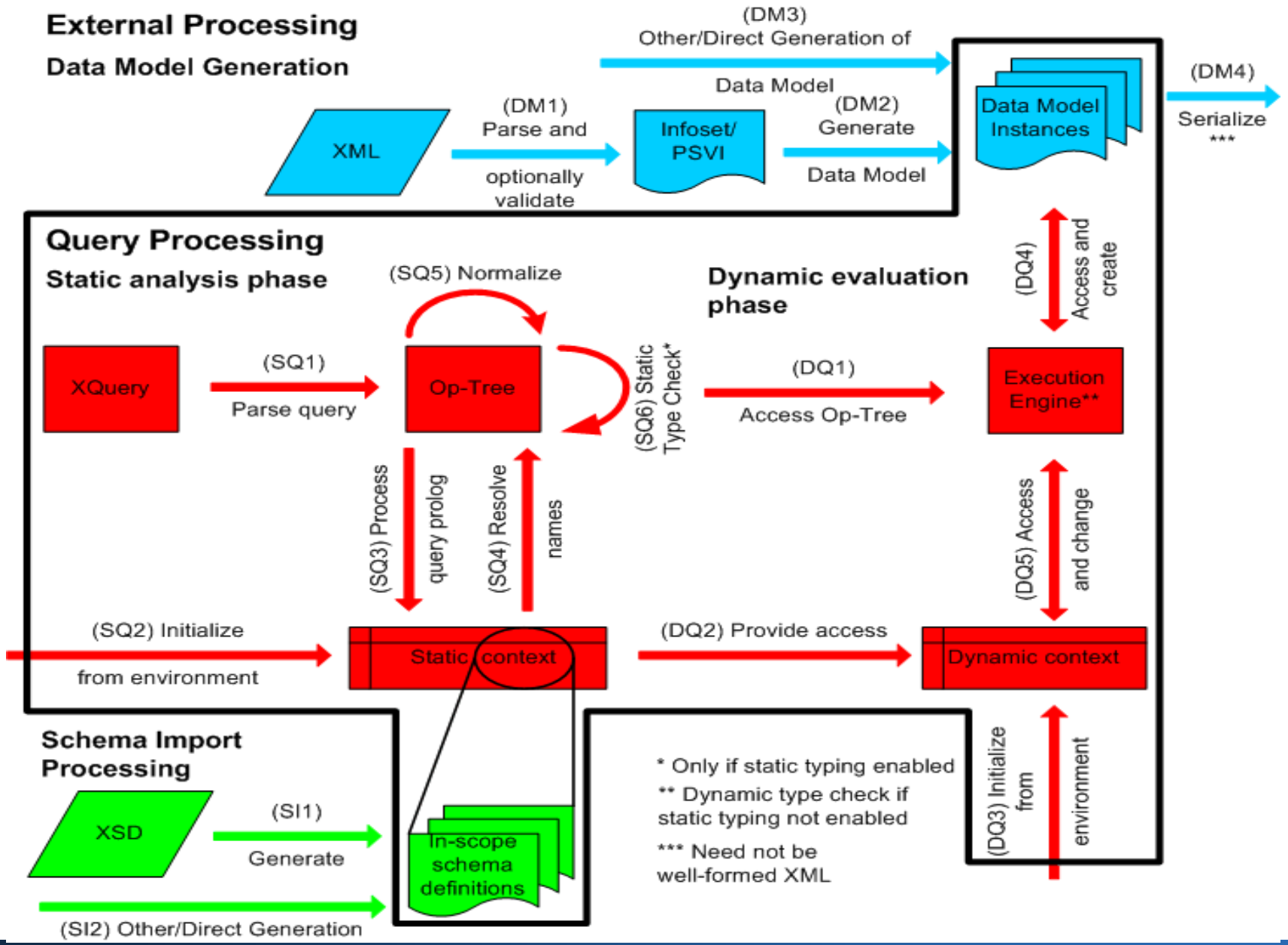
XQuery Type System

- XQuery has a powerful (and complex!) type system
- XQuery types are imported from XML Schemas
- Types are SequenceTypes: Base Type + Occurence Indicator, e.g. *element(), xs:integer+*
- Every XML data model instance has a dynamic type
- Every XQuery expression has a static type
- Pessimistic static type inference (optional)
- The goal of the type system is:
  1. detect statically errors in the queries
  2. infer the type of the result of valid queries
  3. ensure statically that the result of a query is of a given type if the input dataset is guaranteed to be of a given type

XQuery Types Overview

- Derived from XML Schema types
- Atomic Types
- List Types
- Nodes Types
- Special types:
  - Item
  - anyType
  - untyped




External Processing
Data Model Generation
(DM3)
Other/Direct Generation of
Data Model
(DM1)
Parse and
optionally
validate
(DM2)
Generate
Data Model
XML
Infoset/
PSVI
Data Model
Instances
(DM4)
Serialize
***
Query Processing
Static analysis phase
(SQ5) Normalize
Dynamic evaluation
phase
(DQ4)
Access and
create
XQuery
(SQ1)
Parse query
Op-Tree
(SQ6) Static
Type Check*
(DQ1)
Access Op-Tree
Execution
Engine**
(SQ3) Process
query prolog
(SQ4) Resolve
names
(DQ5) Access
and change
(SQ2) Initialize
from environment
Static context
(DQ2) Provide access
Dynamic context
Schema Import
Processing
XSD
(SI1)
Generate
In-scope
schema
definitions
(SI2) Other/Direct Generation
* Only if static typing enabled
** Dynamic type check if
static typing not enabled
*** Need not be
well-formed XML
(DQ3) Initialize
from
environment

Static context

- XPath 1.0 compatibility mode
- Statically known namespaces
- Default element/type namespace
- Default function namespace
- In-scope schema definitions
- In-scope variables
- In-scope function signatures
- Statically known collations
- Default collation
- Construction mode
- Ordering mode
- Boundary space policy
- Copy namespace mode
- Base URI
- Statically known documents and collections

➢ change XQuery expression semantics

➢ impact compilation

➢ can be set by application or by prolog declarations

Dynamic context

- Values for external variables
- Values for the current item, current position and size
- Current date and time
  (stable during the execution of a query!)
- Implementation for external functions
- Implicit timezone
- Available documents and collections

XML Query Structure

- An XQuery basic structure:
  - a *prolog* + an *expression*
- Role of the prolog:
  - Populate the context in which the expression is compiled and evaluated
- Prologue contains:
  - namespace definitions
  - schema imports
  - default element and function namespace
  - function definitions
  - function library (=module) imports
  - global and external variables definitions
  - …

XQuery Grammar

XQuery Expr :=Literal| Variable | FunctionCalls |
PathExpr | ComparisonExpr | ArithmeticExpr|
LogicExpr | FLWRExpr | ConditionalExpr |
QuantifiedExpr |TypeSwitchExpr |
InstanceofExpr | CastExpr |UnionExpr |
IntersectExceptExpr | ConstructorExpr |
ValidateExpr

Expressions can be nested with full generality !

Functional programming heritage.

Literal

XQuery grammar has built-in support for:

- Strings: "125.0" or ‘125.0’
- Integers: 150
- Decimal: 125.0
- Double: 125.e2

- 19 other *atomic types* available via XML Schema
- Values can be constructed
  - with constructors in F&O doc: `fn:true()`, `fn:date("2002-5-20")`
  - by casting (only atomic/simple types)
  - by schema validation (node/complex types)

Variables

- $ + QName
- bound, not assigned
- *XQuery does not allow variable assignment*
- created by `let`, `for`, `some/every`, `typeswitch` expressions, function parameters, prolog
- example:

```
declare variable $x := ( 1, 2, 3 );
$x
```

- $x defined in prolog, scope entire query

Constructing sequences

```
(1, 2, 2, 3, 3, <a/>, <b/>)
```

- "," is the sequence concatenation operator
- Nested sequences are flattened:

```
(1, 2, 2, (3, 3)) => (1, 2, 2, 3,3)
```

- range expressions: `(1 to 3) =>`
  `(1,2,3)`

Combining Sequences

- Union, Intersect, Except
- Work only for sequences of nodes, *not* atomic values
- Eliminate duplicates and reorder to document order

```
$x := <a/>, $y := <b/>, $z := <c/>

($x, $y) union ($y, $z) =>
                    (<a/>, <b/>, <c/>)
```

- F&O specification provides other functions & operators;
  eg. `fn:distinct-values()` and
  `fn:deep-equal()` particularly useful

Conditional expressions

```
if ( $book/@year <1980 )
then ns:WS(<old>{$x/title}</old>)
else ns:WS(<new>{$x/title}</new>)
```

- Only one branch allowed to raise execution errors
- Impacts scheduling and parallelization

Simple Iteration expression

- Syntax :

```
for variable in expression1
return expression2
```

- Example

```
for $x in document("bib.xml")/bib/book
return  $x/title
```

- Semantics :
  - bind the variable to each item returned by *expression1*
  - for each such binding evaluate *expression2*
  - concatenate the resulting sequences
  - nested sequences are automatically flattened

Local variable declaration

- Syntax :

```
let variable := expression1
return expression2
```

- Example :

```
let $x :=document("bib.xml")/bib/book
return count($x)
```

- Semantics :
  - bind the *variable* to the result of the *expression1*
  - add this binding to the current environment
  - evaluate and return *expression2*

FLW(O)R expressions

- Syntactic sugar that combines FOR, LET, IF

- Example

```
for $x in //bib/book                    /* similar to FROM in SQL */
 let $y := $x/author                    /* no analogy in SQL */
 where $x/title="The politics of experience"
                                        /* similar to WHERE in SQL */
 return count($y)                       /* similar to SELECT in SQL */
```

FLWR expression semantics

- FLWR expression:

```
for $x in //bib/book
let $y := $x/author
where $x/title="Ulysses"
return count($y)
```

- Equivalent to:

```
for $x in //bib/book
return  (let $y := $x/author
         return
                 if ($x/title="Ulysses" )
                 then count($y)
                 else  ()
         )
```

More FLWR expression examples

- Selections

```
for $b in document("bib.xml")//book
where $b/publisher = "Springer Verlag"
          and $b/@year = "1998"
return $b/title
```

- Joins

```
for $b in document("bib.xml")//book,
          $p in //publisher
where $b/publisher = $p/name
return ( $b/title , $p/address)
```

The "O" in FLW(O)R expressions

- Syntactic sugar that combines FOR, LET, IF

- Syntax

```
for $x in //bib/book                          /* similar to FROM in SQL */
 let $y := $x/author                          /* no analogy in SQL */
 [stable] order by  ( [expr] [empty-handling ? Asc-vs-desc?
  Collation?] )+                              /* similar to ORDER-BY in SQL */
 return count($y)                             /* similar to SELECT in SQL */
```

Path Expressions

- XQuery includes XPath (not just embedded)
- Second order expression

  expr1 / expr2

- Semantics:
  1. Evaluate *expr1* => sequence of nodes
  2. Bind . to each node in this sequence
  3. Evaluate *expr2* with this binding => sequence of nodes
  4. Concatenate the partial sequences
  5. Eliminate duplicates
  6. Sort by document order
- Implicit iteration
- A standalone step is an expression
  1. step = (axis, nodeTest) where
  2. nodeTest = (node kind, node name, node type)

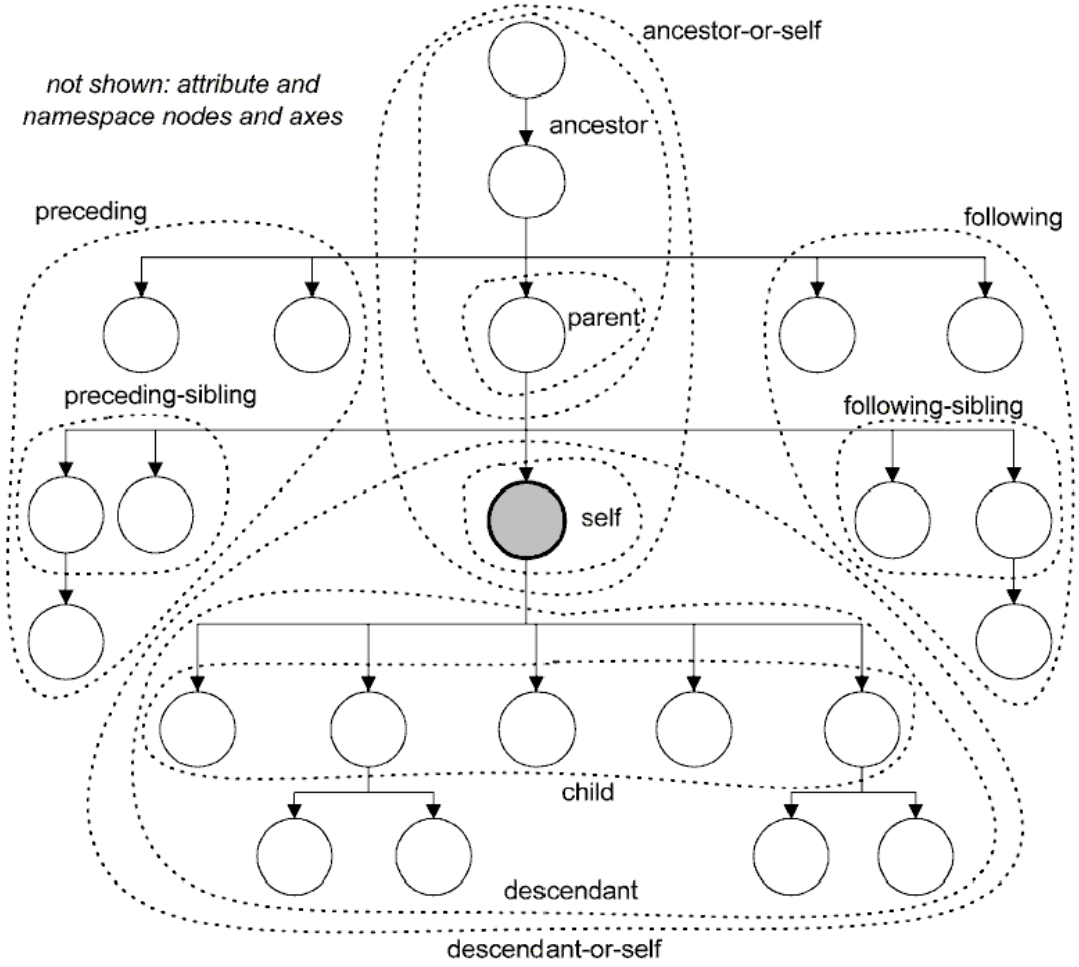
Path Expressions by Example

- Names of all family members (Navigation)
  /family/member/name (~ Projection)
- Names of four year olds.
  /family/member[@age = 4]/name (~Selection)
- Name of the second eldest.
  /family/member[2]/name (~Selection + Ranking)
- Names of members who have a hobby.
  /family/member[hobby]/name(~Selection by Type)
- All names (of anything).
  //name (~Transitive Closure, Recursion)

More on XPath expressions

- A stand-alone step is an expression
- Any kind of expression can be a step !
- Two syntaxes for steps: abbreviated or not
- Step in the non-abbreviated syntax:
  axis '::' *nodeTest*
- Axis control the navigation direction in the tree
  - attribute, child, descendant, descendant-or-self, parent, self
  - The other Xpath 1.0 axes are optional
- Node test by:
  - Name (publisher, myNS:publisher, *: publisher, myNS:* , *)
  - Kind of item (e.g. node(), comment(), text())
  - Type test (e.g. element(ns:PO, ns:PoType), attribute(*,xs:integer)

XPath Axes


ancestor-or-self
not shown: attribute and namespace nodes and axes
ancestor
preceding
following
parent
preceding-sibling
following-sibling
self
child
descendant
descendant-or-self

Long syntax of XPath

- `document("bibliography.xml")/child::bib`
- `$x/child::bib/child::book/attribute::year`
- `$x/parent::*`
- `$x/child::*/descendent::comment()`
- `$x/child::element(*, ns:PoType)`
- `$x/attribute::attribute(*, xs:integer)`
- `$x/(child::element(*, xs:date) | attribute::attribute(*, xs:date)`
- `$x/f(.)`

XPath abbreviated syntax

- Axis can be missing
  - By default the child axis

```
$x/child::person   ->     $x/person
```

- Short-hands for common axes
  - Descendent-or-self

```
$x/descendant-or-self::*/child::comment()->
  $x//comment()
```

  - Parent

```
$x/parent::*    ->     $x/..
```

  - Attribute

```
$x/attribute::year    ->    $x/@year
```

  - Self

```
$x/self::*     ->      $x/.
```

XPath filter predicates

- Syntax:

  expression1 [*expression2*]

- [ ] is an overloaded operator
- Filtering by position (if numeric value) :

```
/book[3]
/book[3]/author[1]
```

- Filtering by predicate :
  - `//book [author/firstname = "ronald"]`
  - `//book [@price <25]`
  - `//book [count(author [@gender="female"])>0]`
- Classical XPath mistakes
  - $x/a/b[1] means $x/a/(b[1]) and not ($x/a/b)[1]
  - `//book [count(author [@gender="female"])]`

Logical expressions

```
expr1 and expr2
expr1 or expr2
```

- return `true, false`
- Different from SQL
  - *two value logic,* not three value logic
- Different from imperative languages
  - *and*, *or* are commutative
- false and error => both false *or* error possible! (non-deterministically)
- For each expression, compute EBV
  - then use standard two value Boolean logic on the two EBV's as appropriate

Arithmetic expressions

```
1 + 4                          $a div 5
5 div 6                        $b mod 10
1 - (4 * 8.5)                  -55.5
<a>42</a> + 1                  <a>baz</a> + 1
validate {<a xsi:type="xs:integer">
          42</a> }+ 1
validate {<a xsi:type="xs:string">
          42</a> }+ 1
```

What is `1 / 2`?

Arithmetic operations - Evaluation

- Apply the following rules:
  - *atomize* all operands.
  - if either operand is (), => ()
  - if an operand is untyped, cast to `xs:double` (if unable, => `error`)
  - if the operand types differ but can be *promoted* to common type, do so (e.g.: `xs:integer` can be promoted to `xs:double`)
  - if operator is consistent w/ types, apply it; result is either atomic value or `error`
  - if type is not consistent, throw type exception

Comparisons

Value	for comparing single values	eq, ne, lt, le, gt, ge
General	Existential quantification + automatic type coercion (similar to arithmetic)	=, !=, <=, <, >, >=
Node	testing identity of single nodes	is
Order	testing relative position of one node vs. another (in document order)	<<, >>

Value and general comparisons

- `<a>42</a>  eq  "42"` → `true`
- `<a>42</a>  eq  42` → `error`
- `<a>42</a>  eq  "42.0"` → `false`
- `<a>42</a>  eq  42.0` → `error`
- `<a>42</a>  =  42` → `true`
- `<a>42</a>  =  42.0` → `true`
- `<a>42</a>  eq  <b>42</b>` → `true`
- `<a>42</a> eq <b>  42</b>` → `false`

Value and general comparisons

- `<a>baz</a> eq 42` — error
- `() eq 42` — ()
- `() = 42` — false
- `(<a>42</a>, <b>43</b>) = 42.0` — true
- `(<a>42</a>, <b>43</b>) = "42"` — true
- `ns:shoesize(5) eq ns:hatsize(5)` — true
  (shoesize, hatsize derived types of xs:integer)
- `(1,2) = 1` — true
- `(1,2) = (2,3)` — true

General Comparison Evalution

Example: $a = $b

- Atomize $a and $b => sequences of atomic values
- Find a pair of values in $a and $b with matching characteristics:
  - Adapt untyped to match type of other operand:
    - Numeric: cast to double
    - String or untyped: cast to string
    - Any other type: cast to other type
  - Perform value comparison on adapted value, e.g. `eq`
- Not deterministic regarding error generation,

  e.g. failed casts, evalution order in sequence

Algebraic properties of comparisons

- General comparisons not reflexive, transitive
  - (1,3) = (1,2) *(but also !=, <, >, <=, >=)*
  - Reasons
    - implicit existential quantification, dynamic casts
- Negation rule does not hold
  - fn:not($x = $y) is not equivalent to $x != $y
- General comparison not transitive, not reflexive
- Value comparisons are *almost* transitive
  - Exception:
    - xs:decimal due to the loss of precision

FunctionCall

- Calling:
  - `my:function(parameter, ...)`
- Signatures:
  - fn:function-name($parameter-name
    as parameter-type, ...) as return-type
- No overloading on type
- Careful with sequences:
  - my:function(1,2,3) ⇔ my:function((1,2,3))
- Library of built-in functions ("F&O")
  - Namespace http://www.w3.org/2006/xpath-functions, Prefix fn:
  - Shared with XSLT, XPath 2.0
  - Also type constructor functions xs:atomicType(…)

Built-in Functions

- Xquery provides a core functions library, shared with XSLT 2.0 and XPath 2.0, in total around 220 functions
- Functions cover operations on built-in data types, node accessors, sequence functions, typecasting, aggregates, context access
- Examples:
  - `fn:string-length(xs:string?) => xs:integer?`
  - `fn:empty(item()*) => boolean`
  - `fn:doc(xs:anyURI)=> document?`
  - `fn:distinct-values(item()*) => item()*`
  - `fn:true() => xs:boolean`
  - `fn:year-from-date(xs:date) => xs:integer?`
  - `fn:max (xs:anyAtomicType*) => xs:anyAtomicType`
  - `fn:current-date() => xs:date`

User-Defined Functions in XQuery

- Function declaration in prolog (or library module)
- In-place/external XQuery functions:
  `"declare" "function" QName "(" ParamList? ")" ("as" SequenceType)? (EnclosedExpr | "external")`
- *declare function local:foo($x as xs:integer) as element()*
  { <a> {$x+1} }
  - Can be recursive and mutually recursive
  - For atomic types, atomization+cast for parameters and result(!)
  - For non-atomic types, only type check
- External functions
- XQuery functions can also serve as
  - database views
  - RPC stubs (e.g. for Web Services)

Node constructors

- Constructing new nodes:
  - elements
  - attributes
  - documents
  - processing instructions
  - comments
  - text
- Side-effect operation
  - Affects *optimization* and *expression rewriting*
- Element constructors create local scopes for namespaces
  - Affects *optimization* and *expression rewriting*

Direct Element constructors

- A special kind of expression that creates (and outputs) new elements
  - Equivalent of a *new Object()* in Java
- Syntax that mimics exactly the XML syntax

  foo bar

  is a normal XQuery expression.
- Embed computed content into Fixed content using {}
  - <a>{*some-expression*}
  - <a> some fixed content {*some-expression*} some more fixed content
  - All Xquery expressions inside {} allowed

Computed (element) constructors

- If even the name of the element is unknown at query time, use the other syntax
  - Not XML, but more general

element {*name*-expression} {*content*-expression}

```
let $x := <a b="1">3</a>
return element {fn:node-name($e)}    {$e/@*, 2 *
  fn:data($e)}
```

⇒ 6

Similar for other node types (attribute, document, PI)

Quantified expressions

- Universal and existential quantifiers
- Second order expressions
  - some *variable* in *expression* satisfies *expression*
  - every *variable* in *expression* satisfies *expression*
- Examples:
  - `some $x in //book satisfies $x/price <100`
  - `every $y in //(author | editor) satisfies $y/address/city = "New York"`

Operators on datatypes

```
expression instanceof sequenceType
```

- returns true if its first operand is an instance of the type named in its second operand

```
expression castable as singleType
```

- returns true if first operand can be casted as the given sequence type

```
expression cast as singleType
```

- used to convert a value from one datatype to another

```
expression treat as sequenceType
```

- treats an expr as if its datatype is a subtype of its static type (down cast)

```
typeswitch
```

- case-like branching based on the type of an input expression

Schema validation

- *Explicit* syntax

  validate [validation mode] { expression }

- Validation mode: strict or lax
- Semantics:
  - Translate XML Data Model to Infoset
  - Apply XML Schema validation
  - Ignore identity constraints checks
  - Map resulting PSVI to a new XML Data Model instance
- It is not a side-effect operation

Ignoring order

- In the original application XML was totally ordered
  - XPath 1.0 preserves the document order through implicit expensive sorting operations
- In many cases the order is not semantically meaningful
  - The evaluation can be optimized if the order is not required
- Ordered { *expr* } and unordered { *expr* }
- Affect : path expressions, FLWR without order clause, union, intersect, except
- Leads to non-determinism
- Semantics of expressions is again context sensitive

```
let $x:= (//a)[1]                    unordered {(//a)[1]/b}
return unordered {$x/b}
```

How to pass "input" data to a query ?

- External variables (bound through an external API)
  declare variable $x as xs:integer external
- Current item (bound through an external API)
  .
- External functions (bound through an external API)
  declare function ora:sql($x as xs:string) as node() external*
- Specific built-in functions
  fn:doc(*uri*), fn:collection(*uri*)

XQuery optional features

- All XQuery up to this point are mandatory for a compliant XQuery implementation
- Schema import feature
- Static typing feature
- Full axis feature
- Module feature

Library modules (example)

Library module

```
module namespace
   mod="moduleURI";
define variable $mod:zero
   as xs:integer {0}
define function
   mod:add($x as
   xs:integer, $y as
   xs:integer)
          as xs:integer
{
   $x+$y
}
```

Importing module

```
import module namespace

   ns="moduleURI";

ns:add(2, ns:zero)
```

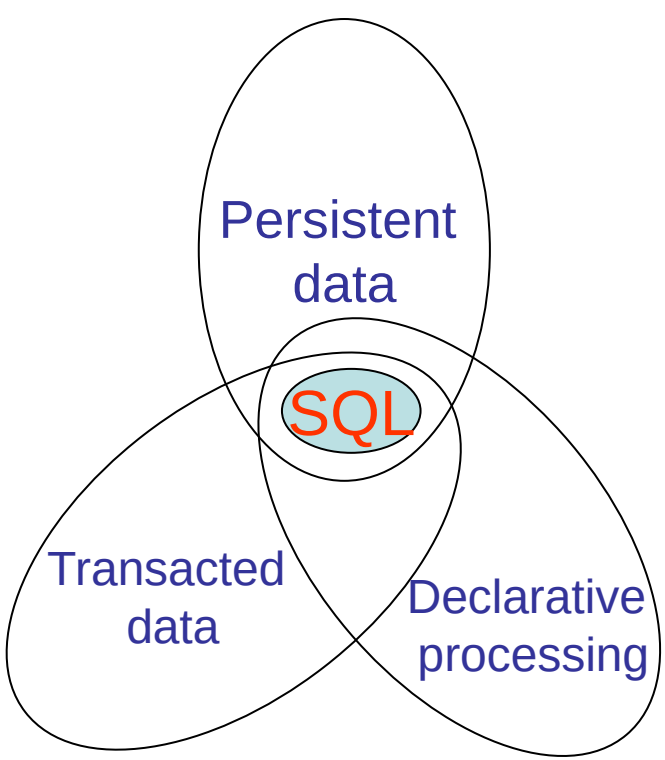
Caution: Import not transitive!

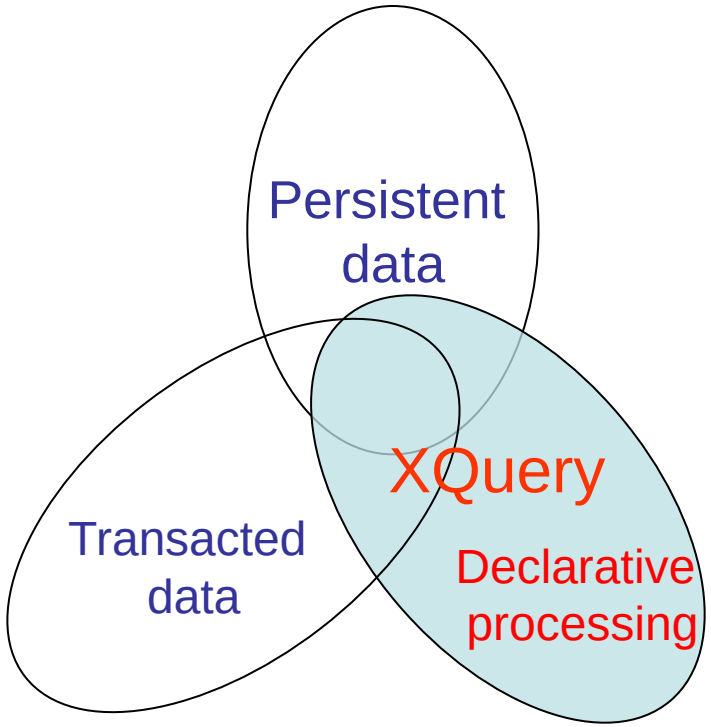
SQL vs. XQuery

- XQuery has implicit Operations
  - casts, exists, duplicate elimination, sorting, ...
  - Important for heterogeneous data

    Important for queries if the schema is unknown
- XQuery has Constructors
  - Important for Transformations of Messages (Info Hubs)
- XQuery can be used for Documents
  - Important for natural-language processing, CMS
- XQuery ist Turing-complete
  - Can be extended to be a full-fledge PL
- XQuery has formal semantics
  - Easier to implement, optimize, and teach (???)

XQuery vs. SQL: beyond the tree vs. table





"XQuery: the *XML* replacement for SQL ?"
No, it's more likely that in the *long term* will be the *declarative* replacement for imperative programming languages like Java or C#.

Some missing functionality

- Web services invocation
- Try-catch mechanism
- Window-based aggregates
- Group by
- Eval () function
- Updates
- Integrity constraints / assertions
- Metadata introspection
- Added as part of 3.0, scripting, libraries

A fraction (2%) of a real customer XQuery

```
let $wlc :=
document("tests/ebsample/data/ebSample.xml")
let $ctrlPackage := "foo.pkg"
let $wfPath := "test"

let $tp-list :=
for $tp in $wlc/wlc/trading-partner
return
<trading-partner
        name="{$tp/@name}"
        business-id="{$tp/party-
identifier/@business-id}"
        description="{$tp/@description}"
        notes="{$tp/@notes}"
        type="{$tp/@type}"
        email="{$tp/@email}"
        phone="{$tp/@phone}"
        fax="{$tp/@fax}"
        username="{$tp/@user-name}"
```

```
{
        for $tp-ad in $tp/address
        return
          $tp-ad
    }
    {
        for $eps in $wlc/extended-property-set
        where $tp/@extended-property-set-name eq $eps/@name
        return
            $eps
    }
    {
        for $client-cert in $tp/client-certificate
        return
        <client-certificate
                name="{$client-cert/@name}"
        >
        </client-certificate>
    }
```

```
{
        for $server-cert in $tp/server-certificate
        return
        <server-certificate
                name="{$server-cert/@name}"
        >
        </server-certificate>
    }
    {
        for $sig-cert in $tp/signature-certificate
        return
        <signature-certificate
                name="{$sig-cert/@name}"
        >
        </signature-certificate>
    }
    {
        for $enc-cert in $tp/encryption-certificate
        return
        <encryption-certificate
                name="{$enc-cert/@name}"
        >
        </encryption-certificate>
    }
```

A Real Query

- Customer use case of BEA Systems
  - WebLogic Integration Product
  - Web Services architecture
- Generated by a graphical tool
- Specifies a complex transformation of a purchase order (business application)
- The alternative is a Java program: appr. same amount of code, 20 x cost

Summary

- XQuery is a functional progr. language
  - strongly typed
  - structured programming with modules, services
  - powerful function library
  - works great with XML, JSON, CSV, ...
- A GREAT data model (totally underestimated)
  - sequences of items (i.e., lists and unord.collections)
  - mother of all: structured, unstructured, streaming, ...
- Family of standards
  - XPath 2.0, XSLT 2.0, XQuery 1.0, XQuery 3.0
  - Update, Scripting, Fulltext

Myths about XQuery

- XQuery is the SQL for XML
  - XQuery is not restricted to databases
  - XQuery works in all tiers
- XQuery is slow
  - again, languages are never slow only impl.
- XQuery is complicated
  - implicit operations (casts, duplicate elimination)
  - 1 line XQuery ~ 10 lines Java

Alternatives to XQuery

- General-purpose languages: Java, C#, ...
  - work well for what they were designed for
  - impedance mismatch to the DB, Web
- LINQ
  - getting better and better, addresses same scope
  - problem: proprietary (owned by Microsoft)
- Scripting languages: Ruby, Groovy, ...
  - I dunno...
- Open question: How many PLs do we need?
  - religious war + power games of vendors

Why are we interested in XQuery?

- Because we are interested in XML
  - only viable way to process XML
- Because it is a declarative language
  - automatic optimization and parallelization
- Because it is powerful: all you need for Web
  - enables single-tier application development
  - great for the cloud and „global optimization“
- Because it is open and we know it
  - we have an easy start and no dead-end worries

Problems of XQuery

- Name
  - both „X“ and „Query“ are misnomers
- Hard and boring if you build processor
  - there is no free lunch...
- Negative marketing – worse than ignore!
  - DeWitt, Stonebraker, IBM, Microsoft
  - all for different reasons
- Poor packaging and products
  - SQL/XML is a nightmare
  - first generation XDBs were terribly slow