The XQuery language

• XQuery is a query language developed by W3C.
• It is derived from several previous proposals:
  – XML-QL
  – YATL
  – Lorel
  – Quilt
  which all agree on the fundamental principles.
• XQuery relies on XPath and XML Schema data types.
Query language requirements

• The W3C Query Working Group has identified many technical requirements:
  – must be **declarative**
  – must respect XML data model
  – must be **namespace** aware
  – must coordinate with **XML Schema**
  – must work even if schemas are unavailable
  – must support simple and **complex data types**
  – must support **universal** and **existential quantifiers**
  – must support operations on **hierarchy** and **sequence** of document structures
  – must combine information from **multiple documents**
  – must support **aggregations**
  – must be able to **transform** and to **create XML structures**
  – must be able to traverse **ID references**

• In short, it must be **SQL generalized to XML**!
XQuery concepts

• A query in XQuery is an expression that:
  – reads a sequence of XML fragments or atomic values
  – returns a sequence of XML fragments or atomic values

• The **principal forms** of XQuery expressions are:
  – path expressions
  – element constructors
  – **FLWOR** (pronounced as "flower") expressions
  – list expressions
  – conditional expressions
  – quantified expressions
  – XQuery built-in functions
  – User-defined functions
  – datatype expressions
Path expressions

• The simplest kind of query is just an XPath expression. As usual, some specific extensions are allowed...

• A simple path expression example:

```xml
document("zoo.xml")//chapter[2]/figure[caption = "Tree Frogs"]
```

  – the result is all `figure` elements with caption “Tree Frogs” in the second chapter of the document `zoo.xml`

  – the result is given as a list of XML fragments, each rooted with a `figure` element

  – the order of the fragments respects the document order (order matters! - as opposed to SQL)
Path expressions (Cont.)

• An XQuery specific extension of XPath allows location steps to follow a new IDREF axis:

\[
\text{document("zoo.xml")//chapter[title="Frogs"]//figref/@refid => figure/caption}
\]

– the result is all captions in figures referenced in the chapter with title “Frogs”
– the => operator follows an IDREF attribute to its unique destination

• As a further generalization, XQuery allows an arbitrary XQuery expression to be used as a location step!
Element constructors

- An XQuery expression may **construct** new XML elements
- More interestingly, an expression may use values bound to **variables**:
  
  ```xml
  <employee empid={ $id }>
    { $name }
    { $job }
  </employee>
  ```

  Here the **variables** $id, $name, and $job must be bound to appropriate fragments.

- In a direct element constructor, **curly braces** `{ }` delimit enclosed expressions, distinguishing them from literal text.
- Enclosed expressions are **evaluated** and replaced by their value. *Without* curly braces `{ }`, e.g. $name will be simply treated as text string in the employee element.

- The output will be like:
  
  ```xml
  <employee empid = "e12">
    <name> Tan AK </name>
    <job> manager </job>
  </employee>
  ```
FLWOR expressions

- The main engine of XQuery is the **FLWOR** expression:
  - **FOR-LET-WHERE-ORDERBY-RETURN**
  - pronounced as "flower"
  - **FOR** iterates on a sequence, binds a variable to each element.
  - **LET** binds a variable to a sequence of elements as a whole
  - generalizes **SELECT-FROM-HAVING-WHERE** from SQL
Example:

```xml
FOR $p$ IN document("bib.xml")//publisher
LET $b$ := document("bib.xml")//book[publisher = $p$]
WHERE count($b) > 100
RETURN $p$
```

- **FOR** generates an ordered list of bindings of **publisher** to $p$
- **LET** associates to each binding a further binding of the list of **book** elements with that publisher (i.e. $p$) to $b$
- **WHERE** filters that list to retain only the desired tuples
- **RETURN** constructs for each tuple a resulting value

- The output of this example will have many publisher elements including the start and end tags, e.g.
  ```xml
  <publisher> Springer </publisher>
  ```
- The combined result is in this case an ordered list of publishers (may contain duplicates) that publish more than 100 books.
**FLWOR expressions (cont.)**

- We probably only want each publisher appears once, so the `distinct-values` function eliminates duplicates in a list:

  ```xml
  FOR $p$ IN `distinct-values(document("bib.xml")//publisher)`
  LET $b$ := document("bib.xml")//book[publisher = $p]
  WHERE count($b) > 100
  RETURN $p
  ```

- Note the difference between **FOR** and **LET**:

  ```xml
  FOR $x$ in /library/book
  ```
  - generates a list of bindings of $x$ to each book element in the library, but:

  ```xml
  LET $x$ := /library/book
  ```
  - generates a single binding of $x$ to the list of all the book elements in the library.
FOR vs. LET

Another example:

FOR $book IN document(“bib.xml”)//book
LET $a := $book/author
WHERE contains($book/publisher, “Addison-Wesley”)
RETURN

    <book>
        { $book/title }
        <count>
            Number of authors: { count($a) }
        </count>
    </book>
Inner Joins

FOR $book IN document("www.bib.com/bib.xml")//book,
    $quote IN
document("www.bookstore.com/quotes.xml")//listing
WHERE $book/isbn = $quote/isbn
RETURN
    <book>
        {$book/title}
        {$quote/price}
    </book>

Note: Inner join only output information which satisfy the join condition. In this example, only those books appeared in both documents will appear in the output.
Outer Joins

FOR $book IN document(“bib.xml”)//book
RETURN
  <book>
    { $book/title }
    {
      FOR $review IN document(“reviews.xml”)//review
        WHERE $book/isbn = $review/isbn
        RETURN $review/rating
    }
  </book>

Note: An outer join is a join that preserves information from one or more of the participating documents, including elements that have no matching element in the other documents.

In this example, the query returned titles of all books in document bib.xml regardless whether or not they have a review in document reviews.xml
Example:

FOR $p$ IN document("www.irs.gov/taxpayers.xml")//person
    $n$ IN document("neighbors.xml")//neighbor[ssn = $p/ssn]
ORDER BY $p/income
RETURN

    <person>
        { $p/ssn }
        { $n/name }
        { $p/income }
    </person>

Note: Order the output by person’s income in ascending order.
Example:

For each “item_tuple” element return the description and reserve_price if the reserve_price is below 50 dollars, and return them in alphabetically ascending order of the item description.

```
FOR $item IN
document("data/R-items.xml")/items/item_tuple
WHERE $item/reserve_price < 50
ORDER BY $item/description
RETURN

    <item>
        {$item/description}
        {$item/reserve_price}
    </item>
```
List expressions

• XQuery expressions manipulate lists of values, for which many built-in functions are supported. For example, the $avg(...)$ function computes the average of a list of integers.

• The following query lists each publisher and the average price of their books:

```
FOR $p IN distinct-values(document("bib.xml")//publisher)
LET $a := document("bib.xml")//book[publisher = $p]/price
RETURN
  <publisher>
    <name> { $p/text() } </name>
    <avgprice> { avg($a) } </avgprice>
  </publisher>
```

Note: $text()$ matches any text node. $p/text()$ returns only the text value of the publisher without the start and end tags of publisher.\textsuperscript{15}
List expressions (cont.)

- Lists can be sorted, as in the following where books costing more than $100 are listed in sorted order:
  - first by the first author
  - second by the title

```xml
document("bib.xml")//book[price > 100]
  SORTBY (author[1],title)
```

- Other list operators compute unions, intersections, differences, and subranges of lists.
Conditional expressions

- XQuery supports a general **IF-THEN-ELSE** construction. The example query:

```xml
FOR $h IN document("library.xml")//holding
RETURN
  <holding>
    { $h/title,
      IF ($h/@type = "Journal")
      THEN $h/editor
      ELSE $h/author
    }
  </holding>
```

This query extracts from the holdings of a library the titles and either editors or authors.
Quantified expressions

- XQuery allows quantified expressions, which decide properties for all elements in a list:

  SOME-IN-SATISFIES
  EVERY-IN-SATISFIES

Similar to existential quantifier and universal qualifier.
Quantified expressions  (cont.)

The following example finds the titles of all books which mention both sailing and windsurfing in some paragraph:

FOR $b$ IN document("bib.xml")//book
WHERE SOME $p$ IN $b$//paragraph
    SATISFIES (contains($p,"sailing")
            AND contains($p,"windsurfing")))
RETURN $b/title
Quantified expressions (cont.)

• The next example finds the titles of all books which mention sailing in every paragraph:

\[
\text{FOR } b \text{ IN document("bib.xml")//book}
\text{WHERE EVERY } p \text{ IN } b//\text{paragraph}
\text{ SATISFIES contains}(p, \text{"sailing"})
\text{RETURN } b/\text{title}
\]
Some More Expressions

- **SOME** $emp$ IN //employees SATISFIES
  ($emp/bonus > 0.25 * $emp/salary)$

- **EVERY** $emp$ IN //employes SATISFIES
  ($emp/bonus > 0.05 * $emp/salary)$
Other issues

Things not covered here:

• hundreds of **built-in operators** and **functions** - contains anything you might think of

• **computed element** and attribute names - allow more flexible queries

• **user-defined functions** - allow general-purpose computations

• the XQuery language definition has many outstanding issues - stay tuned for changes
XQuery 3.0 is an extended version of the XQuery 1.0 Recommendation published on 23 January 2007. A list of changes made since XQuery 1.0 can be found in J Change Log. Here are some of the new features in XQuery 3.0:

1. group by clause in FLWOR Expressions.
2. tumbling window and sliding window in FLWOR Expressions.
3. count clause in FLWOR Expressions
4. allowing empty in for clause, for functionality similar to outer joins in SQL.
5. try/catch expression, for exception handling
6. dynamic function invocation
7. Inline functions
8. Private functions
9. switch expressions
10. Computed namespace constructors
11. Output declarations
12. Annotations
13. Annotation assertions in function tests.