XPath and JSON

Week 8

COS60009: Data Management for the Big Data Age

SWINBURNE UNIVERSITY OF TECHNOLOGY

1

Learning Objectives

- XPath
- Other XML Query Languages (Brief Introduction)
  - XQuery
  - XSLT
- JSON

2

2

Query Languages for XML

- Data extraction, transformation, and integration are well-understood database issues that rely on a query language
- SQL and OQL do not apply directly to XML because of the irregularity of XML data
- XML Query Languages
  - XPath - formulate path expressions for navigating nested structure of XML
  - XQuery
  - XSLT

3

3

XPath

- a technology used for locating parts of an XML document for addressing or navigating to those parts
- be used with XSLT, XQuery
- essentially about moving through the *XPath tree* of a document, testing nodes to determine if they are relevant
- use a *location path* to locate a node or set of nodes in movement
- use *absolute location path* starting at root node or *relative location path* from context node
- W3C standard recommendation - http://www.w3.org/TR/xpath

4

4

XPath Data Model

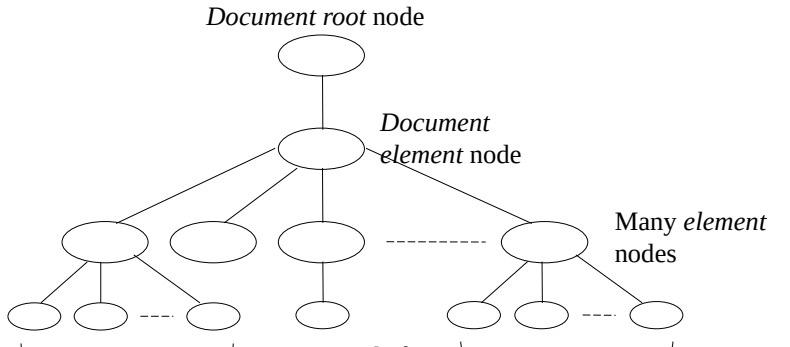
- A hierarchy, the *XPath tree*, is constructed by the parser from the contents of a document
- Tree is made from various nodes representing parts of the document
- There are 7 node types

1. Document root
2. Element
3. Attribute
4. Text
5. Namespace
6. Processing instruction
7. Comment

5

5

XPath Data Model

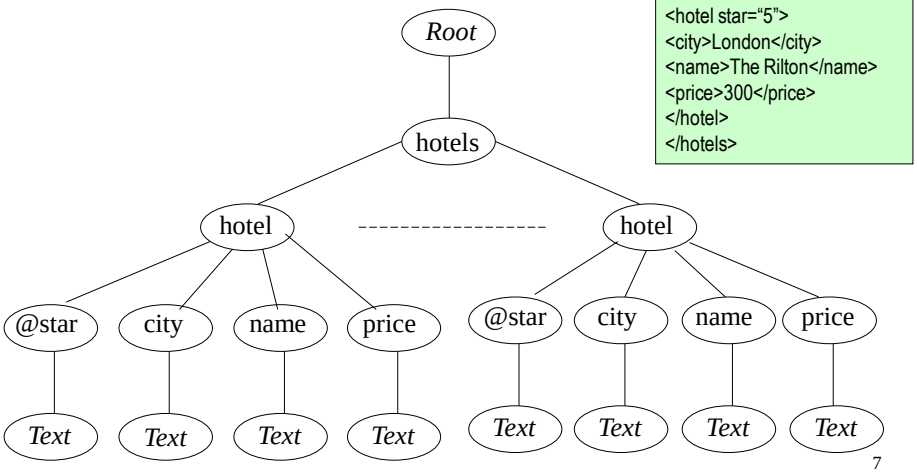


6

6

XPath Tree for the Example

```
<hotels>
<hotel star="3">
<city>Paris</city>
<name>La Splendide</name>
<price>100</price>
</hotel>
… …
<hotel star="5">
<city>London</city>
<name>The Rilton</name>
<price>300</price>
</hotel>
</hotels>
```



7

7

How Does XPath Operate?

- We know that XML content is in hierarchy
- Need to locate various parts of this hierarchy in order to use them
- Start at a *context node*, e.g., **hotels**
- From context node, proceed in one of several directions in the hierarchy – these directions are called *axes*
- Get to desired node through a number of *location steps* involving a *location path*
- Example of location path: **/hotels/hotel/name**
- Need to determine applicable type of node using a *node test*, e.g., Is the node a name element?

8

8

How Does XPath Operate?

- Location step has three aspects:
  - Axis
  - Node test
  - Zero or more predicates
- Syntax of *unabbreviated* location step:

  <axis>::<nodetest><predicate>
- Example of unnabbreviated location step

  parent::hotel[attribute::star="5"]
- Selecting **hotel** elements from the **hotels** element:

  /child::hotels/child::hotel or */hotels/hotel* (*abbreviated* form)
- Location steps separated by "/"

9

9

Axes

- child – context node's children, excluding attribute and namespace nodes
- parent – context node's parent (if node is not document element)
- descendant – context node's descendants, excluding attribute and namespace nodes
- ancestor – context node's parent, its parent's parent, etc.
- self – context node!
- descendant-or-self – context node + descendants
- ancestor-or-self – context node + ancestors
- attribute – attribute nodes of context (element) node
- namespace – namespace nodes of context (element) node
- following-sibling
- preceding-sibling

10

10

Node Tests

- If interested in selecting a particular element or attribute, just name that element or attribute.
- Wildcard (*) can be used to select all nodes of context node (depending upon axis used)
  - child::* selects all child *element* nodes of context node
  - attribute::* selects all *attribute* nodes of context node
- Other tests:
  - text() selects text nodes
  - comment() selects comment nodes
  - processing-instruction() selects PI nodes
  - node() selects all nodes, of all types

11

11

Node tests

```
<Staff>
   <Staff_member>
      <Name>Alan Colman</Name>
      <Email uni="true">acolman@it….</Email>
      <Phone>8771</Phone>
   </Staff_member>
   … …
   <Staff_member>
      <Name>Yun Yang</Name>
      <Email uni="true">yyang@it….</Email>
      <Phone>8752</Phone>
   </Staff_member>
</Staff>
```

Context node	Expression	Returns …
Staff	child::Staff_member	**Staff_member** elements
Staff_member	child::Staff_member	Empty node set
Staff_member (of Alan)	child::*	**Name, Email, Phone** (of Alan)
Staff_member	child::text()	Empty node set
Phone (of Yun)	child::text()	8752
Staff_member (either)	parent::*	**Staff**
8752	parent::*	**Phone** (of Yun)
Phone (of Alan)	attribute::*	Empty node set
Email (of Alan)	attribute::uni	**uni** (of Alan)
Staff	descendant::Email	Both **Email** elements

12

12

Predicates

- Predicates filter the returned *nodeset*
- Predicates are presented in square brackets, e.g.,
  - [attribute::star="5"]
  - [child::price < 150]
- Predicates consist of expressions built from XPath functions
- There are four main categories of XPath functions for use in predicates:
  1. Nodeset functions
  2. Boolean functions
  3. Number functions
  4. String functions

13

13

Abbreviated Location Steps

Axis	Abbreviation	Unabbreviated equivalent
Child	hotel	child::hotel
	*	child::*
	*/price	child::*/child::price
	hotel[last()]	child::hotel[last()]
Self	.	self::node()
Parent	..	parent::node()
Attribute	@star	attribute::star
	@*	attribute::*
descendant-or-self		
	.//city	self::node()/descendant-or-self ::node()
		/child::city
	hotels//name	child::hotels/descendant::name

14

14

Relative and Absolute XPath Expressions

- Relative XPath expressions start from the context node
  - hotel[@star="5"]/price
  - ./city
- Absolute XPath expressions start from the root node "/", i.e., using "/" at start of expression denotes root node as context node, e.g.:
  - /
  - /hotels//city[../hotel/@star="3"] or /hotels/hotel[@star="3"]/city
  - //hotel/name

15

15

Nodesets

- *In /hotels/hotel/city, hotel* elements form a nodeset, and each *hotel* is a context node for any further location step */city*
- If there was a predicate for *hotel* , such as:

  hotel[position() = last()]

  this would return a nodeset of one node
- Some Nodeset functions
  - count(node-set): Number of nodes in any nodeset
    - count(//hotel)
  - last(): Number of last node in context nodeset
  - position(): Number of context node in context nodeset

16

16

XPath Operators and Other Functions

- Arithmetic operators: + | - | * | div | mod
- Logical comparison operators: = | != | < | <= | > | >=
- Logical functions: not | true | false | and | or
- Some number functions
  - sum(node-set)
    - sum(/hotels/hotel/price) div count(/hotels/hotel)
  - ceiling(expr), floor(expr), round(expr)
- Some string functions
  - concat(string, string, …)
  - contains(string1, string2), substring(string, offset, range)
  - substring-before(string1, string2), substring-after(string1, string2)
  - starts-with(string1, string2), string-length(string)

17

17

XQuery

- http://www.w3.org/TR/xquery
- Based on XPath expressions
- FLWOR expressions
  - for-let-order by-where-return
  - similar to SQL: select-from-where-group by-having-order by
  - recursive
  - very powerful

18

18

FLWOR Expressions

- FLWR expression binds values to one or more variables, then uses these variables to construct a result (in general, ordered forest of nodes).
- FOR clauses and/or LET clauses serve to bind values to one or more variables using expressions (e.g., XPath expressions).
- FOR used for iteration, associating each specified variable with expression that returns list of nodes.
- Optional WHERE clause specifies one or more conditions to restrict the binding-tuples generated by FOR and LET.
- Variables bound by FOR, representing single node, are typically used in scalar predicates such as $S/salary > 10000.
- Variables bound by LET may represent lists of nodes, and can be used in list-oriented predicate such as AVG($S/salary) > 20000.

19

19

Example

```
<!DOCTYPE STAFFLIST[
  <!ELEMENT STAFFLIST (STAFF*)>
  <!ELEMENT STAFF (NAME, POSITION, DOB?, SALARY)>
  <!ELEMENT NAME (FNAME, LNAME)>
  <!ELEMENT FNAME (#PCDATA)>
  <!ELEMENT LNAME (#PCDATA)>
  <!ELEMENT POSITION (#PCDATA)>
  <!ELEMENT DOB (#PCDATA)>
  <!ELEMENT SALARY (#PCDATA)>
  <!ATTLIST STAFF branchNo CDATA #IMPLIED>
]>
```

20

20

FLWR Expression 1

List each branch office and average salary at branch.

```
FOR $B IN DISTINCT(document("staff_list.xml")//@branchNo)
    LET $avgSalary :=avg(document("staff_list.xml")/
                              STAFF[@branchNo = $B]/SALARY)
    RETURN
        <BRANCH>
                <BRANCHNO>$B/text()</BRANCHNO>,
                <AVGSALARY>$avgSalary</AVGSALARY>
        </BRANCH>
```

21

FLWR Expression 2

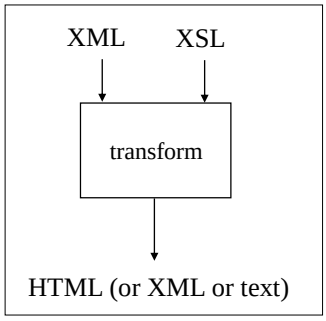
List the branches that have more than 20 staff.

```
<LARGEBRANCHES>
      FOR $B IN
        DISTINCT(document("staff_list.xml")//@branchNo)
        LET $S :=  document("staff_list.xml")/
                          STAFF[@branchNo = $B]
       WHERE count($S) > 20
        RETURN $B
    </LARGEBRANCHES>
```

22

XSLT

- Xalan transform tool: A free XSL processor, implemented in Java, from Apache (http://www.apache.org/)
- Put a stylesheet PI at the top of your XML document and use browsers



23

23

hotels.xml

```
<?xml version="1.0" standalone="no" ?>
<hotels>
  <hotel>
    <city>Paris</city>
    <name>La Splendide</name>
    <type>Budget</type>
    <price>100</price>
  </hotel>
  <hotel>
    <city>London</city>
    <name>The Rilton</name>
    <type>Luxury</type>
    <price>300</price>
  </hotel>
  <hotel>
    <city>Paris</city>
    <name>Marriott Rive Gauche</name>
    <type>Luxury</type>
    <price>350</price>
  </hotel>
  <hotel>
    <city>New York</city>
    <name>The Imperial</name>
    <type>Standard</type>
    <price>150</price>
  </hotel>
  <hotel>
    <city>Paris</city>
    <name>Passy Eiffel</name>
    <type>Standard</type>
    <price>240</price>
  </hotel>
</hotels>
```

24

Paris.xsl

```
<?xml version="1.0" encoding="utf-8"?>
<xsl:stylesheet version="1.0" xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
 <xsl:output  method="html" indent="yes" version="4.0" />
 <xsl:template match="/">
   <table border="1">
     <xsl:for-each select="//hotel[city='Paris']">
       <tr ><td><xsl:value-of select="name"/></td>
       <xsl:choose>
         <xsl:when test="type='Budget'">
              <td style="background:red"><xsl:value-of select="price"/></td>
         </xsl:when>
         <xsl:when test="type='Luxury'">
              <td style="background:lightblue"><xsl:value-of select="price"/></td>
         </xsl:when>
         <xsl:otherwise>
             <td><xsl:value-of select="price"/></td>
         </xsl:otherwise>
       </xsl:choose></tr>
     </xsl:for-each>
   </table>
   <br />Total: <xsl:value-of select="count(//hotel[city='Paris'])"/>
   <br />Average Price: <xsl:value-of select="sum(//hotel[city='Paris']/price) div count(//hotel[city='Paris'])"/>
 </xsl:template>
</xsl:stylesheet>
```

25

Output Document

```
<table border="1">
<tr>
<td>La Splendide</td><td style="background:red">100</td>
</tr>
<tr>
<td>Marriott Rive Gauche</td><td style="background:lightblue">350</td>
</tr>
<tr>
<td>Passy Eiffel</td><td>240</td>
</tr>
</table>
<br>Total: 3<br>Average Price: 230
```

26

26

hotels.xml

```
<?xml version="1.0" standalone="no" ?>
<?xml-stylesheet type="text/xsl"
    href="Paris.xsl"?>
<hotels>
  <hotel>
    <city>Paris</city>
    <name>La Splendide</name>
    <type>Budget</type>
    <price>100</price>
  </hotel>
  <hotel>
    <city>London</city>
    <name>The Rilton</name>
    <type>Luxury</type>
    <price>300</price>
  </hotel>
  <hotel>
    <city>Paris</city>
    <name>Marriott Rive Gauche</name>
    <type>Luxury</type>
    <price>350</price>
  </hotel>
  <hotel>
    <city>New York</city>
    <name>The Imperial</name>
    <type>Standard</type>
    <price>150</price>
  </hotel>
  <hotel>
    <city>Paris</city>
    <name>Passy Eiffel</name>
    <type>Standard</type>
    <price>240</price>
  </hotel>
</hotels>
```

27

JSON

- JSON stands for “JavaScript Object Notation”
- This is a data format that is useful for transmission between server and client
- It’s primary benefit is that it represents data as JavaScript data structures, so that once received on the client, the data can be directly manipulated by JavaScript code without any extraction or manipulation
- However, JSON parsers are not standard on most computer languages.
- Support to manipulate JSON data is limited.

28

28

JSON

- JSON is
  - easy for humans to read and write
  - easy for machines to parse and generate
  - a subset of JavaScript (Standard ECMA-262 3rd Edition - December 1999)
  - for data-interchange only (no variables or control structures)
- JSON is built on two structures:
  - An *unordered* collection of name/value pairs. In various languages, this is realized as an *object*, record, struct, dictionary, hash table, keyed list, or associative array.
  - An *ordered* list of values. In most languages, this is realized as an *array*, vector, list, or sequence.
- See www.json.org

29

29

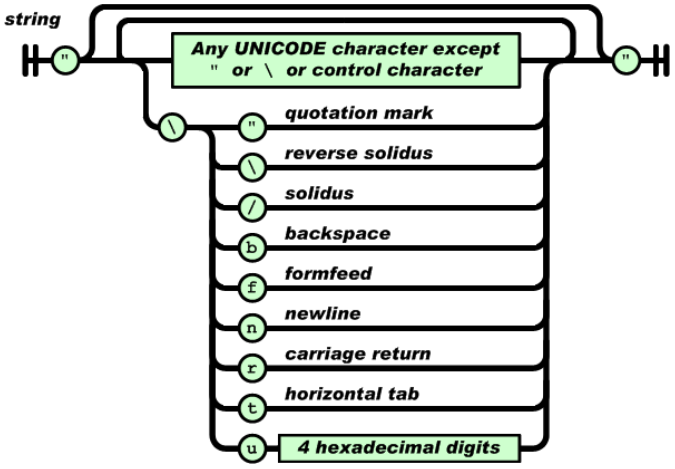
JSON Datatypes (JSON is Typed)

- Value
  - String (double-quoted)
    - Special characters: \", \b, \n, \f, \r, \t, \u, \\, \/
  - Number
  - Boolean value (true/false)
  - null
  - **Object**
  - **Array**
- Object: a collection of string/value pairs
- Array: a list of values

30

30

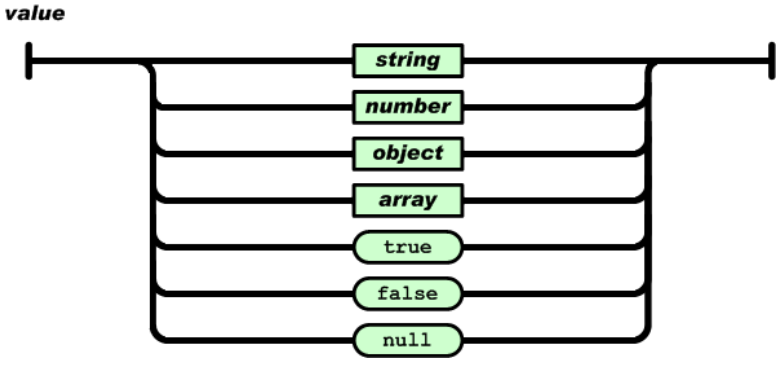
JSON String



31

31

JSON Value



32

32

Objects in JavaScript and in JSON

JavaScript

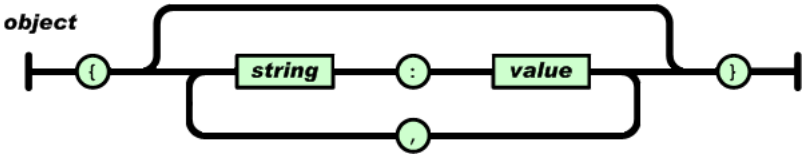
```
var member = new Object();
      member.name = "Jobo";
      member.address = "123 xyz Rd";
      member.isRegistered = true;
```

```
var member =
{ name: "Jobo",
  address: "123 xyz Rd",
  isRegistered: true,
};
```

JSON

```
{ "name": "Jobo",
  "address": "123 xyz Rd",
  "isRegistered": true,
}
```

Note that in JSON, the field names are strings, with quotes.



33

33

Arrays

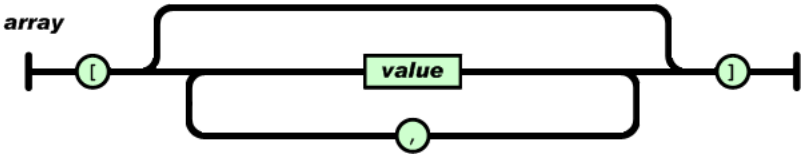
JScript

```
var myArray = new Array(1800, "Princes", "Hwy");
```

```
var myArray = [1800, "Princes", "Hwy"];
```

JSON

```
[1800, "Princes", "Hwy"]
```



34

34

Objects and Arrays

```
{"members": [
    { "name": "Jobo",
      "address": "123 xyz Rd",
      "isRegistered": true,
    },
    { "name": "Frodo",
      "address": "987 pqr Rd",
      "isRegistered": false,
    }
  ]
}
```

This is a more complex object.

We have a single string/value pair, where the string is "members", and the value is an array of two collections of three string/value pairs.

35

35

JSON vs XML

- JSON is an alternative to XML for representing structured data to be retrieved from an application server and manipulated on a client
- JSON is text, and is retrieved as such
- Once accessed on the client, JavaScript functions can directly manipulate the information, rather than using the DOM API, or XSLT transformation
- Often the JSON representation of data will be (a little) shorter than XML representation

36

36

XML

```
<!DOCTYPE glossary PUBLIC "-//OASIS//DTD DocBook V3.1//EN">
<glossary>
 <title>example glossary</title>
 <GlossDiv>
  <title>S</title>
  <GlossList>
   <GlossEntry ID="SGML" SortAs="SGML“>
    <GlossTerm>Standard Generalized Markup Language</GlossTerm>
    <Acronym>SGML</Acronym>
    <Abbrev>ISO 8879:1986</Abbrev>
    <GlossDef>
     <para>A meta-markup language, used to create markuplanguages such as DocBook.
     </para>
     <GlossSeeAlso OtherTerm="GML“/>
     <GlossSeeAlso OtherTerm="XML“/>
    </GlossDef>
    <GlossSee OtherTerm="markup“/>
   </GlossEntry>
  </GlossList>
 </GlossDiv>
</glossary>
```

37

37

Same Data in JSON

```
{"glossary": {
  "title": "example glossary",
  "GlossDiv": {
    "title": "S",
    "GlossList": {
      "GlossEntry": {
        "ID": "SGML",
        "SortAs": "SGML",
        "GlossTerm": "Standard Generalized Markup Language",
        "Acronym": "SGML",
        "Abbrev": "ISO 8879:1986",
        "GlossDef": {
          "para": "A meta-markup language, used to create markup languages such as DocBook.",
          "GlossSeeAlso": ["GML", "XML"]
        },
        "GlossSee": "markup“
      }
    }
  }
 }
}
```

Basically the same structure in the data, but rather than nested tags there are nested JSON objects, and the representation is somewhat terser.

38