1.3. XQuery

XQuery[2] Motivation

The mission of the XML Query project is to provide flexible query facilities to extract data from real and virtual documents on the World Wide Web, therefore finally providing the needed interaction between the Web world and the database world. Ultimately, collections of XML files will be accessed like databases.

[2] http://www.w3.org/XML/Query

- XQuery, akin as relational algebra, is a functional language.
- XQuery queries have a declarative semantics; a given query is reduced to a value, i.e. the queries answer.
- XQuery is strongly typed.
- A XQuery value is a sequence of no or several *items*.
- An item is an atomic value or a (element-, attribute-, text-)node.

XQuery can also be used to construct XML-documents which do not correspond to a subtree of the input XML-document.

Element node creation

- <Order PONumber = "1800">
  Dummy
  </Order>
- element Order {attribute PONumber {"79100"}, "Dummy"}

Element node creation

```
<Order>
{ doc("Orders.xml")//LineItems }
</Order>
```

- Expression doc("Orders.xml") locates the root-node of the respective XML-document.
- Braces '{, }' are used as wrappers around XML-expressions to distinguish them from values.

Why?

```
SELECT x.* FROM
  XMLTABLE('element Order {attribute PONumber {"79100"}, "Dummy"}') x;

SELECT x.* FROM
  XMLTABLE('element Order {attribute PONumber {"79100"}, Dummy}') x;

SELECT x.* FROM
  XMLTABLE('<a att="as"> Dummy </a>') x;

SELECT x.* FROM
  XMLTABLE('<a att="as"> "Dummy" </a>') x;

SELECT x.* FROM
  XMLTABLE('<a att="as"> {"Dummy"} </a>') x;

SELECT x.* FROM
  XMLTABLE('<a att="as"> {Dummy} </a>') x;
```

Oracle XMLQuery: XML-Type column

```
CREATE TABLE OrderTableA (ID NUMBER PRIMARY KEY, XMLDOC XMLType);

INSERT INTO OrderTableA VALUES (1, xmltype(
'<?xml version="1.0"?>
<Orders>
<Order PONumber="1600">
<Reference>ABULL-20140421</Reference>
<Requestor>Alexis Bull</Requestor>
<CostCenter>A50</CostCenter>
<ShippingInstructions>
<name>Alexis Bull</name>
<Address> ... </Address>
<Phone> ... </Phone>
</ShippingInstructions>
<SpecialInstructions/>
<AllowedPartialShipment>false</AllowedPartialShipment>
<LineItems>
<Item>1</Item>
<Item>2</Item>
</LineItems>
</Order></Orders>'
));
```

Oracle XMLQuery: XML-Document

```
DECLARE res BOOLEAN;

empsxmlstring VARCHAR2(1000):=
'<?xml version="1.0"?>
<Orders>
<Order PONumber="1600">
<Reference>ABULL-20140421</Reference>
<Requestor>Alexis Bull</Requestor>
<CostCenter>A50</CostCenter>
<ShippingInstructions>
<name>Alexis Bull</name>
<Address> ... </Address>
<Phone> ... </Phone>
</ShippingInstructions>
<SpecialInstructions/>
<AllowedPartialShipment>false</AllowedPartialShipment>
<LineItems>
<Item>1</Item>
<Item>2</Item>
<Item>1</Item>
</LineItems>
</Order></Orders>';

BEGIN
res := DBMS_XDB.createResource('/public/Orders.xml', empsxmlstring);
END;
/
```

FLWOR-Expression

- A FLWOR-expression is built out of `for`-, `let`-, `where`-, `order`- and `return`-clauses. It evaluates to a stream of tuples, where the tuples are (variable-name, value) pairs.
  - `let` creates a single tuple: one variable, one value.
  - `for` creates a stream of tuples, one tuple for each item in the selection.
  - `where` filters the stream of tuples so only those tuples that satisfy the condition are retained.
- `order by` defines a sorting of the tuples.
- `return` defines the structure of the result.

XML-column

```
SELECT x.* FROM  XMLTABLE('
    for $a in (1,2,3),
        $b in ("a", "b", "c"),
        $c in (10, 20, 30)
return
<Z>{$a, $b, $c}</Z>') x;
```

XMLTable maps the result of an XQuery evaluation into relational rows and columns. More than one column can be filled using XPath (see Oracle Documents).

XML-Document

```
SELECT XMLQuery('
    for $a in (1,2,3),
        $b in ("a", "b", "c"),
        $c in (10, 20, 30)
return
<Z>{$a, $b, $c}</Z>' RETURNING CONTENT) FROM DUAL;
```

Extraction of elements

```
SELECT x.* FROM OrderTable,
     XMLTABLE('for $a in //LineItems/Item
               return <A>{$a}</A>'
               passing XMLDoc) x;
```

```
SELECT
    XMLQUERY ('for $a in doc("/public/Orders.xml")//LineItems/Item
               return <A>{$a}</A>'
               RETURNING CONTENT) FROM DUAL;
```

Injection of elements

```
SELECT x.* FROM OrderTable,
     XMLTABLE(
        '<mySpecial>{
            let $a := //Orders//*[name="Alexis Bull"]/Address/*
            return <A>{$a}</A>
        }</mySpecial>'
        passing XMLDoc) x;
```

Reversal of structure[3]

```
SELECT x.* FROM OrderTableA,
    XMLTABLE('let $a := //Order
      return
        <ShippingInstructions>{$a/ShippingInstructions/*}
        <Order>
           {for $b in $a/*
            where fn:not($b is $a/ShippingInstructions)
            return $b}
        </Order>
        </ShippingInstructions>'
passing XMLDoc) x;
```

[3] for `fn:not()` and `is` see later

Join

```
SELECT x.* FROM OrderTable,
     XMLTABLE('for $a in /Orders//LineItems/Item,
                   $b in /Orders//LineItems/Item
                   where $a/text() <  $b/text()
     return
       <result Left = "{ $a/ancestor/Order/attribute::PONumber }"
               Right = "{ $b/ancestor/Order/attribute::PONumber }"/>'
     passing XMLDoc) x;
```

Quantifiers some, every and conditional expressions

```
SELECT x.* FROM OrderTableA,
    XMLTABLE('for $a in //state["CA"]
    where some $b in $a/../zipCode/text()
    satisfies ($b > 1000)
    return $a/ancestor::Order'
    passing XMLDoc) x;
```

```
SELECT x.* FROM OrderTableA,
    XMLTABLE('for $a in //LineItems
    return
    if (count($a/Item) > 2)
    then <Result status="good"/> else <Result status="failure"/>'
    passing XMLDoc) x;
```

Position in a sequence

```
SELECT x.* FROM OrderTableA,
    XMLTABLE('for $a at $i in //LineItems/Item
    return
    <Result> {$a, $i} </Result>'
    passing XMLDoc) x;
```

XQuery built-in functions: E.g. aggregation

```
SELECT x.* FROM OrderTableA,
    XMLTABLE('for $a in //LineItems
    return <Result> {sum($a/Item) * count($a/Item)} </Result>'
    passing XMLDoc) x;
```

Comparison operators

- Atomic values: `eq,ne,lt,le,gt,ge`
- Sequences of atomic values: `=,!=,<,<=,>,>=`

  Evaluates to `true` if the comparison holds for at least one pair of values of the given sequences. Relation steht.

  Thus `(1,2)=(2,3)` and `(2,3)=(3,4)`, but `(1,2)!=(3,4)`.
- Nodes: `is`
  where it is referred to identity.
- Document order: `<<,>>`.
- Nested structure: `fn:deep-equal()`

Negation: `fn:not()`

Examples

```
SELECT XMLQuery(' ( 1, ( 1, "eins", 1 ), 1 ) '
RETURNING CONTENT) FROM DUAL;

SELECT XMLQuery(' distinct-values(( 1, ( 1, "eins", 1 ), 1 )) '
RETURNING CONTENT) FROM DUAL;

SELECT XMLQuery('
if ((1,2) = (2,3) )
then <Result> equal nodes </Result> else <Result> unequal nodes </Result>'
RETURNING CONTENT) FROM DUAL;

SELECT XMLQuery('let $a := doc("/public/Orders.xml")//LineItems[1]/Item[1]
                 let $b := doc("/public/Orders.xml")//LineItems[2]/Item[2]
return if ($a << $b )
then <Result> ok </Result> else <Result> not ok </Result>'
RETURNING CONTENT) FROM DUAL;

... return if ($a is $b )  ...

... return if (fn:deep-equal($a, $b) ) ...
```