Node search preceding node construction - XQuery inviting non-XML technologies

Hans-Jürgen Rennau, parsQube GmbH
<hans-juergen.rennau@parsqube.de>

Abstract

We propose an approach how to complement XPath navigation with a node search which does not require node construction. Node search is based on a set of external properties (a "p-face") which a node may assume in the context of a node collection. Being external, these properties can be retrieved without node construction, and being stored outside the nodes they can be maintained and queried by non-XML technologies, e.g. relational and NOSQL databases. A small set of concepts, carefully aligned with the XQuery data model, allows the seamless integration of various non-XML technologies driving node selection, without introducing any dependencies of XQuery code on any particular technology. A first implementation of the concepts is presented.

Table of Contents

Introduction

XPath navigation is based on node properties (document-uri, node-name, parent, children, attributes, typed-value, …). If the documents have not already been translated into an internal representation (e.g. by loading them into an XML database), access to node properties requires the parsing of document text into node trees, which is a very expensive operation. The consequence is a severe scaling problem, as you quickly run out of available memory and available time. Navigation requires node construction, and there is no general concept of a *search* as an operation to be distinguished from navigation and possibly preceding node construction.

The W3C data models of XML (infoset [2], XDM [7]) do not recognize entities "above" the single document. True, XPath and XQuery ([4], [5]) take into account the existence of collections, which are sequences of nodes. A collection may have a URI or be assigned the status of being the “default collection”, but is has no properties apart from that.

If we recognize collections as entities in their own right, we can use them as *context* within which to associate member nodes with additional, collection-scoped properties. For example, documents representing logged events might be associated with a property “timestamp”. A collection might thus associate its member nodes with additional properties in the same way as a map associates its entries with a map key. And such additional properties serve the same purpose as map keys: to enable a selection among the contained units which does not require inspection of the units themselves. As the additional node properties are conferred by – and scoped to - the containing collection, let us call them *external properties*. Each external property has a name and a value. The value may echo document contents literally (be a copy of the value found in a particular attribute or element), be derived from document contents (e.g. counts and existence flags), or be independent of contents (e.g. the timestamp of document receipt). Excepting the last case, a property value can be viewed as the value of an XQuery expression evaluated in the context of the node in question.

External properties are stored outside the node itself, and property access does not require node construction. The external properties which a collection confers to its member nodes can thus be used to filter the collection without constructing any member nodes. This setup means a potential gain in efficiency, as in some scenarios the amount of node construction may be reduced drastically. Consider this document:

```
<resources>
  <r uri="file:/a/b/c1" t="2014-02-05T08:16:48" status="green"/>
  <r uri="file:/a/b/c2" t="2014-02-06T13:09:03" status="red"/>
  <r uri="file:/a/b/c3" t="2014-02-06T13:11:51" status="green"/>
  …
</resources>
```

It can be interpreted as the definition of a collection which associates XML documents – identified by their URI – with two external properties named `t` and `status`. The first might be the document creation time, and the second some kind of processing status. The creation time has perhaps been extracted from a particular location within the document, and the status has perhaps been determined by applying an XQuery expression to document contents. If an intended processing concerns only those resources whose creation time is after 2014-02-05 and whose status is “red”, a very simple processing of the <resources> element (`$rs`) will yield the required input, without accessing any resources not needed:

```
$rs/r[@t ge "2014-02-06"][@status eq "red"]/@uri/doc(.)
```

The example used an XML document to represent a collection conferring external properties to its member nodes. If we constrain external properties to be name/value pairs and values to be atomic sequences, other technologies may be used as well, e.g. relational or NOSQL databases. Such technologies may be leveraged when filtering the node collection in order to restrict node construction to those members actually required by the processing.

This paper explores the concept of a node search based on external node properties. The goal is a new XPath function offering node search, which is generic and yet enables the hidden integration of

non-XML technologies into the operation of node search. We report a first implementation of the new concepts which is part of an open source framework for the development of XQuery command-line tools [6].

Basic concepts

First we introduce several basic concepts found useful in the design of a generic node search API which can be implemented using non-XML technologies.

external property

A mapping of a node to a property name and a property value. The name is constrained to be an NCName, the value is a sequence of atomic items, which may be empty. The mapping is defined in the context of a particular node collection. Different collections can associate the same node with different sets of external properties.

p-face

All external properties of a given node, conferred to it by a given node collection. Within a p-face, property names are unique. A p-face can therefore also be viewed as a collection-scoped mapping of names (= property names) to atomic sequences (= property values).

p-model

A specification how the nodes of a collection are mapped to their p-faces. A possible form is a set of property names associated with XQuery expressions, implying that the property value is the atomized result of evaluating the expression in the context of the node in question. Another form is a single XQuery expression resulting in a map of property names and values. A third form declines the possibility to derive the p-face from node contents and stipulates that a p-face is set from without, added to the node when it is inserted into the collection. The first two forms may also allow *additional* properties with unpredictable values and possibly also unpredictable names.

Note: a more formal definition of the p-model might be achieved by introducing the concept of a node insertion function and representing the p-model as a function argument.

node descriptor

A string from which a node can be constructed. A node descriptor may be the serialized node content, a URI or some other, proprietary kind of pointer providing node access.

p-node

Entity of information containing a node descriptor and a p-face.

p-collection

A sequence of p-nodes. By implication, also a sorted mapping of nodes to p-faces.

p-test

A condition imposed on an external property, defined in terms of (1) a property name, (2) a test value, (3) an operator relating property value and test value (e.g. =, <, >, …). A test value is a sequence of one or more atomic items. Example of a p-test, represented as text:

```
creationDate > 2013-12-31
```

p-filter

A condition imposed on a p-face, defined in terms of one or more p-tests and logical operators “and”, “or”, “not”. Example of a p-face, represented as text:

```
creationDate > 2013-12-31 && status = green
```

node search

The operation of applying a p-filter to a p-collection. The result consists of those collection members whose p-face matches the p-filter.

Node search API

The new search API which we propose consists of a single XPath function `filteredCollection`. The input identifies a p-collection and specifies a p-filter. The output is the sequence of nodes contained by the p-collection and matching the p-filter. The p-collection is identified by a URI, and the p-filter can be supplied either as a structured representation (a `<pfilter>` element), or as an equivalent filter descriptor string. Informally, the function can be presented with two different signatures:

```
node()* filteredCollection( $pcollection as xs:anyURI,
                            $pfilter as element(pc:pfilter)?)
node()* filteredCollection( $pcollection as xs:anyURI,
                            $pfilter as xs:string?)
```

Formally, there is only one signature whose second parameter is typed as `item()?`.

The structured representation is an element tree composed of elements representing p-tests (`<p>`) and logical operations (`<and>`, `<or>`, `<not>`). Example:

```
<pfilter xmlns="http://www.infospace.org/pcollection">
   <and>
      <p name="status" value="green,closed" sep=";"/>
      <or>
         <p name="date" op=">" value="2013-12-31"/>
         <p name="supplier" op="~" value="X*"/>
      </or>
   </and>
</pfilter>
```

Besides the obvious operators (=, !=, <, <=, >, >=) there is a further operator (~) which represents pattern matching. If a test value comprises several items, they can be alternatively represented as a concatenated string (accompanied by attribute @sep which specifies the item separator) or by placing each item in an `<item>` child element of the `<p>` element.

Every filter element can be represented by an equivalent descriptor string composed of name/operator/value triples and logical operators (`&&`, `||`, `not`). Parentheses are used for controlling operator precedence and for delimiting a sequence of comma-separated test value items. Example:

```
status = (green,closed) && (date > 2013-12-31 || supplier ~ X*)
```

It is important to note that the function signature is generic: it avoids any dependency on the technologies involved in the internal representation of the collection, the filtering operation and the construction of selected nodes. The nodes may be constructed by parsing files, but they may also be obtained by parsing strings retrieved from a database (relational or NOSQL), or perhaps they are retrieved as nodes from an XML database. Likewise, the association between nodes and their external properties may be represented by XML structures, by relational tables, by NOSQL documents

or yet in different ways. It is the function implementation which must handle the involved artifacts appropriately.

Node management API

The new node search API requires the definition and maintenance of p-collections. Key operations are:

- the creation of p-collections (instantiating artifacts)
- the loading of nodes into these collections (including their external properties)
- the removal of nodes from these collections
- the deletion of p-collections

Like node search, node management should be supported by a generic API which hides any implementation details and in particular the technologies actually used. We postpone the formal definition of this API to a later section, as it requires some further conceptualizations which we shall introduce first.

To bridge the gap

The concept of p-faced collections is based on the abstract notion of name/value pairs and their association with nodes, avoiding any assumptions about the implementation – the artifacts used to represent these pairs and associations. While it is not difficult to design a particular implementation, this would amount to an application-level solution only, because we still lack concepts how to integrate alternative implementations into a single and coherent model. We face a certain gap between the abstract notion of p-faced collections and any concrete implementation approach, a lack of terms which would allow us to speak about those implementations in a unified way. To bridge this gap, we propose three further concepts: *NCAT* (node catalog), denoting the artifact(s) which represent the collection contents; *NCAT model*, the information required to manage and use those artifacts; and *NODL* document (Node cOllection Description Language), a machine-readable description of a p-collection, including its p-model and its NCAT model.

NCAT

The artifact (or set of artifacts) used to represent a p-faced collection is called an NCAT (node catalog). Examples of an NCAT are an XML file, a set of relational database tables and a NOSQL database.

Software implementing the filtering of a p-collection must access the NCAT and apply to it operations of selection and retrieval. This presupposes information about the NCAT. Such information can be conceptualized as the NCAT model.

NCAT model

A p-collection has a logical structure which can be summarized as a sequence of entries containing a node and its external properties. An NCAT model defines a pattern how to map this logical structure to a technology-dependent representation (an NCAT). The model also specifies the configuration data required in order to manage and filter the NCAT. An example of such configuration data might be the connection data of a relational database which harbours the NCAT. It should also be noted that the model may define dependencies on the p-model, that is, the names and types of external properties used within the p-collection. An example would be a dependency of the number and names of relational tables to be used, and of the names and types of their columns.

NODL

Software tasked with filtering or managing a p-collection deals with its NCAT and must be supplied with information about it. The information should be received in a standardized and machine-readable

form, which is a NODL document (Node cOllection Description Language). A NODL specifies the NCAT model and supplies any configuration data which it defines. A NODL also includes a description of the p-model, as details of the NCAT may depend on the number, names and types of external properties.

Note. The term NODL was chosen in order to stress the conceptual analogy to WSDL and WADL.

NODL documents

A NODL has the following general structure:

```
<nodl xmlns="http://www.infospace.org/pcollection">
   <collection>…
   <pface>…
   <nodeDescriptor>…
   <ncat>…
</nodl>
```

The <collection> element

Attributes of the <collection> element supply general information about the p-collection:

- @name - a collection name
- @uri - a collection URI
- @format - a list of formats in which collection members may be delivered
- @doc - a short description

The <pface> element

The <pface> element supplies the p-model. Each <property> child element defines an external property in terms of a name, a data type including cardinality, an optional length constraint and an XQuery expression yielding the property value. Some examples:

```
<property name="cr" type="xs:dateTime"
          expr="ShoppingCart/@CreationDate"/>
<property name="bookingID" type="xs:string*" maxLength="40"
          expr="//BookingId"/>
<property name="travellerCount" type="xs:integer"
          expr="//Travellers/count(Traveller)"/>
```

The <pface> element can also have an <anyProperty> child element. When present, it signals that nodes may have additional external properties with unpredictable names and values.

The <nodeDescriptor> element

The <nodeDescriptor> element specifies how the NCAT represents collection nodes – as URI, as serialized node text, or as a non-URI accessor string.

The <ncat> element

The <ncat> element identifies the NCAT model and provides its configuration data. The element has a single child element whose name and contents depend on the NCAT model used. Currently, two NCAT models are supported, so that the <ncat> element contains either an <xmlNcat> or a <sqlNcat> element. The contents of these elements are described in the following section.

First NCAT models

This section summarizes first versions of two NCAT models, defining the structure of NCATs based on XML documents and relational databases, respectively.

The XML NCAT model

The model defines the structure of an NCAT implemented as an XML document. The model is best illustrated by a little example of such an NCAT:

```
<pnodes xmlns="http://www.infospace.org/pcollection" name="scarts"
      uri="" formats="xml" nodeDescriptor="uri" count="32048">
   <pnode node_uri="..." t="2014-11-13T12:29:56" sID="02Z9ROU5">
      <tourOp>AB</tourOp>
      <check>
         <item>Address</item>
         <item>DoubleBook</item>
         <item>WatchList</item>
      </check>
   </pnode>
   <pnode node_uri="..." ...>...</pnode>
   <pnode node_uri="..." ...>...</pnode>
   ...
</pnodes>
```

The NCAT is implemented by a `<pnodes>` element containing `<pnode>` elements which represent the collection nodes. Every `<pnode>` element has a @node_uri attribute providing a URI which can be resolved to the node in question. The external properties are represented by further attributes and/or child elements with names equal to the property name. The first entry of the NCAT shown above thus represents a node which has four external properties: `t`, `sID`, `toupOp` and `check`. A single-valued property is represented by either an attribute or a child element with simple content. A multiple-valued property is represented by a child element which has one `<item>` child per value item.

Note that a program processing such an NCAT cannot predict whether to find the property values in attributes or elements. This does not mean any ambiguity, as property names are unique within each p-face. Given a p-node $pnode, the following expression retrieves the value of property `foo` reliably:

```
$pnode/(@foo, foo[not(item)], foo/item)
```

If the filtering of p-nodes resulted in a sequence $pnodes of selected p-nodes, the following expression returns the selected nodes themselves:

```
$pnodes/@node_uri/doc(resolve-uri(., base-uri(..)))
```

Configuration data

Configuration data specify the document URI of the NCAT document and, optionally, the names of properties which should always be stored as elements, rather than attributes. By default, single-valued properties are stored as attributes.

NODL representation

When the XML NCAT model is used, it is represented by an `<xmlNcat>` element with empty content, a mandatory @documentURI attribute and an optional @asElems attribute. The latter contains a whitespace-separated list of property names or name patterns. Example:

```
<xmlNcat documentURI="/ncats/xsds.ncat" asElems="*remark* airport"/>
```

Note. The current version of the XML NCAT model does not support any other representation of the node than by URI. This restriction does not apply to the SQL NCAT model, because it can make much sense to store serialized documents within the NCAT itself if it *is* a database.

The SQL NCAT model

Displaimer

The model as described and implemented has only been used with MySQL databases, version 5.6. An extension enabling its use with other database systems is planned.

The NCAT is implemented by a set of tables harboured by a single database. Given a p-model, the following tables are used (where ${cname} denotes the collection name found at `$nodl/collection/@name`):

${cname}_ncat	The main table, containing the node descriptor (e.g. node URI) and all single-valued properties.
${cname}_ncat_${pname}	One such table is used for every property $pname whose cardinality allows multiple values.
${cname}_ncat___dyn	This table is only used if the p-model allows properties with arbitrary names and values (thus if `$nodl/pface/anyProperty` exists).

The main table ${cname}_ncat contains the following columns:

nkey	an integer primary key identifying the node
node_uri	the node URI (only if `$nodl/nodeDescriptor/@kind` = ‘uri’)
node_text	the serialized node (only if `$nodl/nodeDescriptor/@kind` = ‘text’)
node_access	the node accessor (only if `$nodl/nodeDescriptor/@kind` = ‘accessor’)
${pname}	for every property $pname whose cardinality precludes multiple values: the property value

If one of the columns `node_uri` or `node_access` exists, it is associated with an index enforcing unique values. Every column $pname (which represents a single-valued property) is associated with an index whose length is the minimum of the length constraint found in the p-model (`$nodl/pface/property[@name eq $pname]/@maxLength`) and the value 200.

Every table ${cname}_ncat_${pname} represents a multiple-valued property $pname. It has these columns:

nkey	a foreign key referencing the `nkey` column in the main table
pkey	an integer primary key
${pname}	the value of an item of property $pname

The column $pname is associated with an index whose length is the minimum of the length constraint found in the p-model and the value 200.

The table ${cname}_ncat__dyn is used only if the p-model admits arbitrary property names. It has the following columns:

nkey	a foreign key referencing the `nkey` column in the main table
pkey	an integer primary key
pname	a property name

pvalue	a property value item

The columns `pname` and `pvalue` are both associated with an index whose length is 100 and 200, respectively.

Configuration data

Configuration data specify the RDBMS, the database connection (host, user name, password) and the database name.

NODL representation

When the SQL NCAT model is used, it is represented by an `<sqlNcat>` element with empty content and the following attributes: @rdbms, @rdbmsVersion, @host, @user, @password, @db. Example:

```
<sqlNcat rdbms="MySQL" rdbmsVersion="5.6" host="localhost"
        user="guest" password="guest123" db="pcol"/>
```

First implementation

This section summarizes a first implementation of the concepts introduced in this paper. The implementation is part of `ttools` [6], an open source framework for the development of XQuery command-line tools.

Like the framework as a whole, the implementation of p-collections is pure XQuery – no changes of the XQuery processor code were made. Support for the SQL NCAT model currently requires use of the BaseX processor [1], as access to relational databases is implemented using the BaseX extension functions `sql:connect` and `sql:execute`. Support for other processors offering equivalent extension functions will be added shortly.

Node search API

Applications developed with `ttools` have access to the function `filteredCollection` discussed in section Node search API. Example code:

```
tt:filteredCollection($nodl, "airport=(DUS,CGN) && lowcost=true")
```

Assuming that $nodl is a `<nodl>` element node, the function call returns all nodes found in the collection described by $nodl and which have within this collection an external property `airport` with a value equal “DUS” or “CGN” and also an external property `lowcost` with a value `true`. If $nodl uses an XML NCAT model, both of the following entries would contribute nodes to the result:

```
<pnode node_uri="…" lowcost="true" airport="DUS"/>
<pnode node_uri="…" lowcost="true">
   <airport>
      <item>DUS</item>
      <item>FRA</item>
   </airport>
</pnode>
```

If $nodl uses an SQL NCAT model, the matching entries would be determined by a SQL select command with a `where` clause referencing a `lowcost` and an `airport` column. The details of the command text depend on the collection name and on the cardinalities of the external properties (implied by `$nodl/pface/property/@type`). Assuming that the collection name is `offers`, `lowcost` is single-valued and `airport` is multiple-valued, the following command would be used behind the scenes:

```
SELECT node_uri FROM offers_ncat AS t1
```

```
LEFT JOIN offers_ncat_airport AS t2 ON t1.nkey = t2.nkey
WHERE `lowcost` = 'true' and `airport` in ('DUS', 'CGN')
```

Input parameter type docSEARCH

Using the `ttools` framework, application code does not receive external variables as supplied by the invocation, but the results of a pre-processing which the framework applies to the input. A `ttools` based application has one single external variable (named `request`), which represents the tool invocation and which is parsed into an operation name and a set of named input parameters. Here comes an example of a `request` value, invoking an operation called `typeReport` and supplying two input parameters, `comp` and `xsds`:

```
typeReport?comp=ctype, xsds=/projects/xsd/*.xsd
```

The pre-processing is controlled by annotations contained by the XQuery modules, which associate parameter names with a type and possibly further information. Application code retrieves a parameter by passing its name to a `getParam` function, and it receives a value constructed from the input text (a substring of the external variable `request`) in accordance with the type annotation. Given the following annotations:

```
<operation name="typeReport">
   <param name="xsds" type="docDFD"/>
   <param name="comp" type="xs:string">
</operation>
```

the application could retrieve the parameter values like this:

```
let $comp as xs:string := tt:getParam($request, "comp")
let $xsds as document-node()* := tt:getParam($request, "xsds")
```

Note that the delivered value of parameter `xsds` is a sequence of document nodes, whereas the supplied value consists of a file name pattern. The transformation of supplied parameter text to delivered parameter value is controlled by the type annotation (`docDFD`).

`ttools`' support for p-collections is reflected by the parameter type `docSEARCH` which is available for use in parameter annotations. Supplied parameter values must consist of a URI and a p-filter, separated by a question mark:

```
nodlURI?pfilter
```

The delivered value is the sequence of document nodes obtained by applying the p-filter to the p-collection defined by the NODL document found at the specified URI. For example, given these annotations:

```
<operation name="typeReport">
   <param name="xsds" type="docSEARCH"
   <param name="comp" type="xs:string"/>
</operation>
```

and the following invocation:

```
typeReport?comp=ctype, xsds=/nodls/xsds.nodl?ctype~*lang*
```

the application code would receive the filtered p-collection implied by the parameter value:

```
let $xsds as document-node()* := tt:getParam($request, "xsds")
```

Note that the application code does not even call `tt:filteredCollection` – the parameter value is *delivered* as the filtered collection specified, to know: the sequence of all documents which are contained by the p-collection defined by the NODL found at `/nodls/xsds.nodl` and which have an external property `ctype` with a value containing the string “lang”.

Node management API

Applications developed with `ttools` have access to a node management API consisting of the following functions:

```
tt:createNcat ($nodl as element(pc:nodl))
tt:feedNcat   ($nodl as element(pc:nodl), $nodes as node()*)
tt:feedNcat   ($nodl as element(pc:nodl), $dirFilter as xs:string+)
tt:copyNcat   ($nodl as element(pc:nodl), $pfilter as element(pc:pfilter)?,
               $toNodl as element(pc:nodl)
tt:deleteNcat ($nodl as element(pc:nodl))
```

Preconditions for their use are:

1. The application was created using a `flavour` parameter set to `basex79` or `basex80`
2. The supplied NODL document uses either an XML NCAT model or an SQL NCAT model
3. If the NODL uses an SQL NCAT model:
   a. the RDBMS MySQL is installed (version >= 5.6)
   b. the official JDBC driver for MySQL (MySQL Connector/J) has been downloaded and placed in the lib directory of the BaseX installation

Function `tt:feedNcat` can be supplied with *directory filters*, which are strings with a `ttools`-defined syntax. Using directory filters, one may specify complex filters on file system contents in a very concise way, specifying directories which are searched recursively or shallowly, positive and/or negative file name patterns and optional constraints controlling the inclusion/exclusion of subdirectories encountered during recursive search.

A small sample application

This section describes the creation of a small sample application which illustrates the use of p-collections. Let us assume we want to create a command-line tool which offers various reports about XSD documents specified by the command-line call. Our application will support the possibility to specify the input XSDs as a filtered p-collection. The actual reporting is kept trivial to avoid distraction from the main point, which is the use of p-collections.

After downloading the `ttools` framework from [6], we use it to create a command-line application named `xspy`. Its first version shall have a single module of application code (`items.mod.xq`) and support a single operation `tns`. The operation will report the target namespaces found in a set of XSDs.

Having installed `ttools` in a directory `/ttools`, we want to create an initial version of our new application in a directory `/apps/xspy`. We achieve this with the following call:

```
basex -b "request=new?dir=/apps/xspy,mod=items,ops=tns, flavor=basex79"
   /ttools/ttools.xq
```

The call creates the application directory and fills it with XQuery modules supplied by the framework, as well as a generated first version of the application module `items.mod.xq`.

What we shall do now is:

1. create a p-collection containing XSDs, which are associated with a useful set of external properties
2. write some application code producing a simple report on target namespaces
3. invoke the application, supplying it with a filtered p-collection

In order to create a p-collection containing XSDs, we first create a NODL document similar to the following:

```
<nodl xmlns="http://www.infospace.org/pcollection">
   <collection name="xsds" uri="" formats="xml" doc="A collection of XSDs."/>
```

```
    <pface>
      <property name="tns" type="xs:string?" maxLength="100"
                expr="//xs:schema/@targetNamespace"/>
      <property name="stype" type="xs:string*" maxLength="100"
                expr="//xs:schema/xs:simpleType[@name]/@name"/>
      <property name="ctype" type="xs:string*" maxLength="100"
                expr="//xs:schema/xs:complexType[@name]/@name"/>
      <property name="elem" type="xs:string*" maxLength="100"
                expr="//xs:schema/xs:element/@name"/>
      <property name="att" type="xs:string*" maxLength="100"
                expr="//xs:schema/xs:attribute/@name"/>
      <property name="group" type="xs:string*" maxLength="100"
                expr="//xs:schema/xs:group/@name"/>
      <property name="agroup" type="xs:string*" maxLength="100"
                expr="//xs:schema/xs:attributeGroup/@name"/>
      <property name="enum" type="xs:string*" maxLength="100"
                expr="//xs:enumeration/@value"/>
      <anyProperty/>
    </pface>
    <nodeDescriptor kind="uri"/>
    <ncat>
       <sqlNcat rdbms="MySQL" rdbmsVersion="5.6" host="localhost"
                user="guest" password="guest123" db="pcol"/>
    </ncat>
</nodl>
```

The p-collection will thus support the following external properties:

- tns – the target namespace
- stype – the names of contained simple type definitions
- ctype – the names of contained complex type definitions
- elem – the names of contained global element declarations
- att – the names of contained global attribute declarations
- group – the names of contained model group definitions
- agroup – the names of contained attribute group definitions
- enum – the values of contained enumeration values

Due to the `<sqlNcat>` element, the NCAT will be created in the form of relational tables inserted into a database `pcol`. Note that the table names are implied by the collection name (`xsds`) and the names of those external properties which may be multiple-valued (including all properties whose type has an occurrence indicator '*' or '+') .

The NCAT is created by the following builtin operation of our new application:

```
basex -b request=_createNcat?nodl=/nodls/xsds.nodl /apps/xspy/xspy.xq
```

Now we fill the NCAT with all XSDs found in or under the directory `/xsds/niem-2.1`:

```
basex -b "request=_feedNcat?nodl=/nodls/xsds.nodl,
          dirs=|/xsds/niem-2.1/*.xsd" /apps/xspy/xspy.xq
```

While the feed proceeds, we may observe how the MySQL tables are filled:

```
mysql –uguest –pguest123
mysql>use pcol
mysql>select node_uri from xsds_ncat;
=>
+-------------------------------------------------------------------------------
| node_uri
+-------------------------------------------------------------------------------
```

```
| file:/xsds/niem-2.1/ansi-nist/2.0/ansi-nist.xsd
| file:/xsds/niem-2.1/ansi_d20/2.0/ansi_d20.xsd
| file:/xsds/niem-2.1/apco/2.1/apco.xsd
...
+-------------------------------------------------------------------------------

mysql> select stype from xsds_ncat_stype;
=>
+-----------------------------------------------------+
| stype                                               |
+-----------------------------------------------------+
| AccidentSeverityCodeSimpleType                      |
| AlarmAcknowledgementCodeSimpleType                  |
| AlarmDescriptionCodeSimpleType                      |
| AlarmEventCategoryCodeSimpleType                    |
| AlarmEventLocationCategoryCodeSimpleType            |
...
+-----------------------------------------------------------------------+
```

Now we write some application code. We edit the generated function `f:tns` so that it returns a simple target namespace report: listing all target namespaces encountered in the input XSDs, and for each namespace the document URIs of all XSDs using it. We replace the generated dummy version of function `f:tns` by the following code:

```
declare function f:tns($request as element()) as element() {
    let $docs := tt:getParams($request, 'fdocs')
    let $tns :=
        for $xsd in $docs
        group by $tns := $xsd/xs:schema/@targetNamespace
        let $uris :=
            for $uri in $xsd/document-uri(.)
            order by lower-case($uri)
            return <xsd uri="{$uri}"/>
        order by lower-case($tns)
        return <tns uri="{$tns}">{$uris}</tns>
    return
        <z:tns countDocs="{count($docs)}">{$tns}</z:tns>
};
```

Note that the filtered XSDs are obtained simply by retrieving the input parameter `fdocs`:

```
   let $docs := tt:getParams($request, 'fdocs')
```

This is explained by the fact that the generated annotations (which we neither changed not removed) declare the operation to have a parameter `fdocs` of type `docSEARCH`:

```
   <operation name="tns" type="node()" func="tns">
      <param name="fdocs" type="docSEARCH*" sep="SC" pgroup="input"/>
      ...
```

The remainder of the function body creates a list of target namespaces found in the filtered collection. After these changes, the call

```
   basex -b request=tns?fdocs=/nodls/xsds.nodl?stype~*country*
      /apps/xspy/xspy.xq
```

produces a report of all target namespaces containing simple type definitions with a name matching `*country*`:

```
<z:tns xmlns:z="http://www.ttools/org/sql/ns/xquery-functions" countDocs="5">
```

```
  <tns uri="http://niem.gov/niem/domains/jxdm/4.1">
    <xsd uri="file:///C:/xsds/niem-2.1/domains/jxdm/4.1/jxdm.xsd"/>
  </tns>
  <tns uri="http://niem.gov/niem/domains/screening/2.1">
    <xsd uri="file:///C:/xsds/niem-2.1/domains/screening/2.1/screening.xsd"/>
  </tns>
  <tns uri="http://niem.gov/niem/fips_10-4/2.0">
    <xsd uri="file:///C:/xsds/niem-2.1/fips_10-4/2.0/fips_10-4.xsd"/>
  </tns>
  <tns uri="http://niem.gov/niem/iso_3166/2.0">
    <xsd uri="file:///C:/xsds/niem-2.1/iso_3166/2.0/iso_3166.xsd"/>
  </tns>
  <tns uri="urn:oasis:names:tc:ciq:xal:3">
    <xsd uri="file:///C:/xsds/niem-2.1/external/have/1.0/xAL-types.xsd"/>
  </tns>
</z:tns>
```

Note that to produce this report, only the five documents which are reported are parsed. Behind the scenes, their selection is accomplished by the following SQL query:

```
SELECT node_uri FROM xsds_ncat AS t1
   LEFT JOIN xsds_ncat_stype AS t2 ON t1.nkey = t2.nkey
   WHERE `stype` LIKE '%country%'
```

Also note that calling the application, we specified the p-collection by supplying its NODL. The NODL *encapsulates all implementation details*, of which we need not be aware in order to use the collection. To complete this little tutorial, we create a copy of the NODL (`xsds2.nodl`), in which we replace the `<sqlNcat>` element by an `<xmlNcat>` element:

```
<ncat>
   <xmlNcat documentURI="/ncats/xsds.ncat"/>
</ncat>
```

We create and populate the XML based NCAT exactly as we created the SQL based NCAT:

```
basex -b "request=_createNcat?nodl=/nodls/xsds2.nodl" /apps/xspy/xspy.xq
basex -b "request=_feedNcat?nodl=/nodls/xsds2.nodl,
          dirs=|xsds/niem-2.1/*.xsd" /apps/xspy/xspy.xq
```

Using the new NODL instead of the previous one, we obtain the same results, although now behind the scenes the documents are selected by evaluating the XML based NCAT.

Discussion

This work attempted to prove the concept of p-collections as a way of complementing XPath navigation with a node search which does not require node construction. In many scenarios, the use of XML databases seems to be a better alternative, mainly as the stored documents can be selected in a “deep” way which may respond to anything found anywhere within the documents, and in an “open” way as there is no limit to the possibilities of defining conditions. These are in fact the very strengths of XPath and XQuery, and in comparison the filtering of p-collections appears somewhat primitive, limited to the “flat” set of external properties and to the combination of atomic property tests via the logical operations of `and`, `or` and `not`.

Nevertheless, we believe that p-collections bear some promise. Major advantages are:

- non-XML technologies can be leveraged (and all their power, maturity and scalability)
- the possibilities of adding further technologies are not limited
- technology choices are on the level of single collections – for any collection the best fit can be used
- XQuery code using p-collections is independent of technology choices

- p-collections are simple to manage thanks to a generic node management API

The use of p-collections thus allows something which could be called “distributed technologies” and which, in spite of its severe limitations, may seriously compete, in specific scenarios, with the “monolithic” alternative of working with a particular XML database system, no matter how powerful.

The current version of the concepts constrains the names of external properties to be NCNames. We refrained from admitting QNames, rather than NCNames, as they would introduce a certain increase of complexity - how to represent them elegantly in non-XML resources and queries? But should the step yet be taken, external properties could be RDF triples, and the potential of this perspective remains to be explored.

Bibliography

[1] BaseX: XML database and XQuery processor. http://basex.org

[2] Cowan, John & Tobin R., eds. XML Information Set (Second Edition). W3C Recommendation 4 February 2004. http://www.w3.org/TR/xml-infoset/

[3] Rennau, Hans-Jürgen. "XQuery topic tools - concept, user interface, development framework." Presented at Balisage: The Markup Conference 2014, Washington, DC, August 5 - 8, 2014. In Proceedings of Balisage: The Markup Conference 2014. Balisage Series on Markup Technologies, vol. 13 (2014). doi: 10.4242/BalisageVol13.Rennau01.http://www.balisage.net/Proceedings/vol8/html/Rennau01/BalisageVol8-Rennau01.html.

[4] Robie, Jonathan et al, eds. XML Path Language (XPath) 3.0 W3C Recommendation 8 April 2014. http://www.w3.org/TR/xpath-30/

[5] Robie, Jonathan et al, eds. XQuery 3.0: An XML Query Language. W3C Recommendation 8 April 2014. http://www.w3.org/TR/xquery-30/

[6] Topic tools - a lightweight framework for developing powerful command-line tools with XQuery. https://github.com/hrennau/TopicTools

[7] Walsh, Norman et al, eds. XQuery and XPath Data Model (3.0). W3C Recommendation 8 April 2014. http://www.w3.org/TR/xpath-datamodel/