XQuery as a data integration language

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

Christian Grün, BaseX GmbH <christian.gruen@gmail.com>

Abstract

The appropriateness of the XQuery language for data integration is explored. The starting point is an assessment of integration capabilities in an XML-only environment. The next step is an evaluation of the degree to which one may extend these capabilities to heterogeneous environments with multiple media types and various data access protocols. This leads to the identification of a key challenge, which is the structured representation of non-XML data formats by items of the XQuery data model. The current support for such representation is reviewed, and a conceptual base is proposed for modeling the relationship between data model items and instances of non-XML formats. As special facets of data integration, the roles of REST and RDF in XQuery-based integration are discussed, and general limitations of XQuery as an integration language are acknowledged.

Table of Contents

Introduction

Speaking of data integration, we presuppose scenarios where information is distributed over several resources. These may have the same media type - e.g. XML, JSON or CSV - or different media types. Concerning the access protocol, they may be provided by the same platform – e.g. file system, a particular database system, or HTTP GET – or by several platforms. Every general purpose language (Java, C#, Python, Perl, …) has various APIs for accessing resources exposed on different platforms and using different media types. Therefore any general purpose language can be used in order to accomplish tasks of data integration. So why should one assign to a particular language the quality of being a “data integration language”?

The key aspect is simplicity. A language particularly well-suited for data integration enables particularly simple solutions for typical operations of data integration. In a nutshell, it enables simplicity when dealing with multiplicity and heterogeneity. We shall investigate the degree to which XQuery ([14]) matches this requirement. The question is relevant to the potential role which XQuery may play in common IT landscapes. The answers may change basic views of the XQuery language: its main purpose, its scope and the course of its further voyage. If the potential for data integration is large, extended support for data integration should be a major goal when defining future versions of XQuery as a standard, as well as new versions of XQuery processors as products.

XQuery, seen from the outside

Many people are interested in data integration, but in this context we are not aware of any particular attention paid to XQuery. The prevalent feeling seems to be that XQuery is a specialized language for extracting content from XML documents. Interest in such functionality is greatly reduced by the fact that in many environments XML documents are not dealt with directly, as they are hidden by tools and technologies. For example, many web services (REST and SOAP) consume and produce XML, but developers of client and server applications usually deal with objects (e.g. JAXB objects), rather than the message documents themselves. As a consequence, XQuery has the status of a niche language offering support for uncommon tasks.

Basic facts about XQuery

Data integration is concerned with the selection of information from multiple and heterogeneous resources, combining and transforming it into new information, which is either delivered to receiving software components or stored in multiple and heterogeneous resources. These requirements should be used as a backdrop when considering a few basic facts about XQuery.

XQuery as an information oriented language

- XQuery is a language focused on the expression of information in both ways – as a selection of existent information, and as a construction of new information
- *Selection and construction of information are key operations of data integration*

XQuery as an aggregation-friendly language

- The data model is built on the key abstraction of an item sequence, rather than on a single item
- As a consequence, many operations can be applied to an item sequence, rather than only to an individual item; in particular, path expressions can be applied to any number of input documents “simultaneously”
- *Aggregated processing of sets of resources is a key operation of data integration*

Expanding scope of resource access

- XQuery 1.0 restricted resource access to XML resources identified via a single URI (in practice: files, XML database entries and documents retrievable via HTTP GET and ftp)
- XQuery 3.0 enables the reading of text resources
- XQuery 3.1 enables the parsing of JSON text into a structured representation
- Vendor-specific extension functions extend the scope of resource access further (parsing CSV, reading the contents of archives, access to SQL and NoSQL databases, support for HTTP POST messages, …)
- *Access to multiple data platforms and media types is a precondition of data integration*

The integration of XML data

The current section focuses on a scenario in which all relevant resources are XML documents. This setting hides the specific challenges of processing non-XML data with an XML-centric language, as well as the general challenges of dealing with heterogeneous data formats. Our goal is to highlight the ability of XQuery to deal with structured information distributed over several resources. The possibility of keeping these abilities in a more general context - when dealing with non-XML data and multiple formats - will be explored afterwards (the section called “Resource access” and the section called “XDM binding”).

Independently of the media types involved, several operations tend to be important in the context of data integration:

- Finding information
- Constructing information
- Resource exploration - gathering information about unknown or little known resources
- Resource validation - assessing the conformance of resources to expectations

Finding information

Before data can be integrated, they must be found and selectively addressed.

The XPath navigation model

XQuery is built around the XPath navigation model, which represents the selection of information contained by XML documents as a traversal of tree-structured information. An XML document is modelled as a tree of nodes, which represent the elements, their text content and their attributes. The traversal is accomplished by a single or several steps, where each step expresses a selection of nodes, and each non-initial step maps the preceding selection to a new selection. In the common case of *axis steps*, the selection of items consists in the combined operations of mapping and filtering: mapping each node of the previous selection to all nodes found in a particular structural relationship – e.g. being child nodes or descendant nodes – and filtering them by name or node kind and optionally by further criteria. For example, the following expression

```
/descendant::booking/flight[cabinClass= 'economy']/arrival/@airport
```

is a traversal which

- starts at the root of an XML document containing the current context item
- moves on to all descendant nodes, filtered by name which must be equal to “booking”

- moves on to all child nodes, filtered by name which must be equal to “flight” and by the additional condition that the node must have a child element with a name equal to “cabinClass” and a text content equal to the string “economy”
- moves on to all child nodes, filtered by name which must be equal to “arrival”
- moves on to all "airport" attributes found on one of the “arrival” elements reached by the previous step

Note the conciseness: a single expression specifies a very complex selection of information. Also note the simplicity of the expression - it allows an intuitive understanding which approaches the exact semantics rather closely. Of special interest in the context of data integration is the fact that the complexity of the involved documents is hidden: the expression assumes the existence of four sets of items found in a particular structural relationship to each other, and apart from that it does not assume any details about the document being traversed, like the number of hierarchical levels and the names of any elements and attributes not used as selection criteria. The XPath navigation model uncouples the complexity of selection from the complexity of the documents to which the selection is applied, and this may simplify the tasks of data integration significantly.

Bulk navigation

Although navigation is often applied to single documents, it is not constrained to such local usage. The steps of a path are not necessarily axis steps (as in the previous example), but can be arbitrary expressions, provided they yield only nodes (in the case of a non-terminal step) or either only nodes or only atomic values (in the case of a terminal step). In particular, a step may produce nodes contained by *several* documents. For example, the expression

```
$msgs/descendant::booking/flight[cabinClass='economy']/arrival/@airport
```

applies a complex navigation to all nodes bound to the variable `$msgs`. Similarly, the expression

```
(doc('msg1.xml'), doc('msg2.xml'), doc('msg3.xml'))/descendant::booking
```

extracts elements from three documents returned by the first step of the path expression.

A general pattern for bulk navigation consists in streaming a sequence of URI values into a path expression whose first step is a `doc()` function call applied to the context item:

```
UFACTORY ! doc(.)/STEPS
```

where UFACTORY is an expression returning a sequence of URIs and STEPS is a path expression. Some examples:

```
tokenize('msg1.xml msg2.xml msg3.xml')                    ! doc(.)/descenda
unparsed-text-lines('fileNames.txt')                      ! doc(.)/descenda
file:list('/dir1/dir2/dir3') ! concat('dir1/dir2/dir3/', .) ! doc(.)/descenda
```

Another interesting pattern is given by the use of “document catalogs” - simple XML documents containing a collection of document URIs, for example:

```
<docs>
   <doc href='/a/b/c/foo.xml'/>
   <doc href='/a/b/c/bar.xml'/>
   <doc href='/x/y/z/poo.xml'/>
   <doc href='http://example.com/msgs/zoo.xml'/>
</docs>
```

An easy way to create such catalogs relies on the `file:list` function, an extension function defined by the EXPath community group ([3]):

```
<docs>{
   let $dir := '/a/b/c' return
   file:list($dir, true(), '*.xml') !
   <doc href="{concat($dir, '/', .)}"/>
}</docs>
```

The function call returns the names of all files found in a specified directory and matching a specified name pattern. If the second call parameter has the boolean value `true`, the file search is extended to all sub directories of the specified directory. A slight modification of the code suffices to feed the catalog with the contents of *several* directories and their sub directories:

```
<docs>{
   for $dir in tokenize('/a/b/c /x/y/z') return
   file:list($dir, true(), '*.xml') !
   <doc href="{concat($dir, '/', .)}"/>
}</docs>
```

After storing such a catalog as a file (e.g. `docs.xml`), an expression like

```
doc('docs.xml')//@href/doc(.)/descendant::booking
```

navigates the sum total of referenced documents.

To summarize, XQuery support for bulk navigation is excellent, as the semantics of navigation are independent of the cardinality and actual contents of the nodes to which it is applied. Any path expression P can be effortless “promoted” to bulk navigation: just add one or several initial steps (S) producing a sequence of documents:

```
P   =>   S/P
```

or add an initial step `doc(.)`, make the extended path the right-hand operand of a simple map expression (E1 ! E2) and add a left-hand operand (some expression E) producing a stream of URIs:

```
P   =>   E ! doc(.)/P
```

A similar pattern consists in shifting the path expression into the `return` clause of a FLWOR expression and adding to the path an initial step referencing a FLWOR variable, e.g.:

```
P   => for $doc in E return $doc/P
P   => for $uri in E return doc($uri)/P
```

In the context of data integration, this ability to apply navigation to single and multiple resources alike is obviously very useful.

Boundless navigation

As pointed out in the previous section, a non-terminal path step may be any expression provided it yields only nodes. This implies that navigation may “leap” across document boundaries. A simple pattern combines initial steps leading to items which contain document URIs, a subsequent `doc(.)` step resolving the URIs and trailing steps which move into the resulting documents:

```
P1/doc(.)/P2
```

where P1 is a path expression yielding document URIs and P2 is a path expression navigating into the documents referenced by these URIs. Example:

```
/descendant::booking/flight/brand/@href/doc(.)//name[@lang='fr']
```

Remembering that any navigation may be turned into bulk navigation by just adding initial steps to the path, we can easily combine bulk navigation with cross-document navigation:

```
doc('msgs.xml')//@href/doc(.)/descendant::booking/flight/brand/@href/doc(.)/
```

In summary, the XPath model of navigation is essentially boundless – navigation can have starting points distributed over several documents and can cross document boundaries any number of times.

Compound navigation (FLWOR)

As path steps are (almost) arbitrary expressions, they can also reference the values bound to variables, and these may in turn be the result of path expressions. FLWOR expressions can therefore be used as a convenient "workbench" for constructing compound navigation. The following code example …

- navigates into a configuration document, extracting a date value (`$since`)
- navigates into an inventory of codes, selecting codes annotated to have a creation date greater or equal to the `$since` date
- navigates into a set of message documents, selecting flight elements containing new airport codes

```
let $since := doc('cfg.xml')//since
let $newCodes := doc('codes.xml')//code[@since ge $since]
return
   doc('msgs.xml')//@href/doc(.)/
      descendant::booking/flight[arrivalAirport/@code=$newCodes]
```

Weighing the complexity of the operation against the amount and complexity of code, support for compound navigation may be regarded as a distinct strength of the XQuery language.

Aggregating information (FLWOR/group by)

Data integration often requires aggregation – reducing the information content of related resources to an aggregated representation. The `group by` clause of FLWOR expressions is well suited for such purposes. To give an example, the following code extracts from a set of documents all `flight` elements, groups them by the departure airport and reports for each departure airport the respective arrival airports:

```
<routes>{
   for $f in doc('msgs.xml')//@href/doc(.)//flight
   group by $airport := $f/departureAirport/@code
   let $to := sort(distinct-values($f/arrivalAirport/@code))
   return
      <route from="{$airport}" to="{$to}"/>
}</routes>
```

Constructing information

In information processing, the finding of existent and the construction of new information play equally fundamental roles.

Embedded expressions

In the context of data integration, construction is particularly important as a means to collect and recombine distributed information, producing new entities of augmented value. Such construction must transfer existent structures – either as faithful copies or transformed into different, often condensed representations – into an emerging new structure. The ideal is seamless integration of information retrieval into construction. The XQuery capabilities in this regard are provided by node constructor expressions. A key feature is the concept of embedded expressions, which are like placeholders to be replaced at run time by the expression values. In particular, the embedding of path expressions amounts to an extremely concise reuse of existent structures, and for a literal merging of the operations of navigation and construction. Consider the following code:

```
<bookings>{doc('msgs.xml')//@href/doc(.)//booking}</bookings>
```

which constructs a new document containing copies of all `booking` elements found in a set of documents. The elements are copied into the result document, including attributes and all descendant nodes. The next example does not copy the elements, but represents them by short summary elements:

```
<bookings>{
  doc('msgs.xml')//@href/@doc(.)//booking/
     <booking kind='{@bookingType}' provider='{brand}'/>
}</bookings>
```

Note that the attribute constructors (e.g. `kind='…'`) are in turn embedded expressions navigating into the elements selected by the preceding path step. Bringing into play the XQuery support for bulk navigation and adding to the constructed elements information about their provenance, we achieve an interesting result with very little code:

```
let $dir := '/data/msgs'
let $bookings :=
   file:list($dir, true(),'msg*.xml') ! concat($dir, '/', .) ! doc(.)//booki
   <booking kind='{@bookingType}' provider='{brand}' src='{root()/document-u
return
   <bookings count='{count($bookings)}'>{$bookings}</bookings>
```

The examples showed that the XQuery abilities for concise navigation and for concise construction complement each other. They enable an elegant mapping of distributed information onto new, useful entities of information.

Transformation

When new information is essentially a modified representation of information already existent, construction is often perceived as transformation. A common use case in the context of data integration – where applications deal with a multitude of resources, formats and format versions - is the normalization of data sources, mapping different format versions to a single reference format expected by downstream components. Consider the following code:

```
//booking/f:bookingToRefBooking(.)/f:processRefBooking(.)
```

which assumes the existence of two user-defined functions, one transforming `booking` elements to a normalized format and the other processing a normalized `booking` element. The code navigates to `booking` elements and feeds a normalized version of each one to a processing function. This example demonstrates the remarkable flexibility of path expressions which can combine navigation with any operation mapping nodes to other nodes - in this case navigation with transformation.

The XQuery language includes a `typeswitch` expression which enables transformation by recursive processing, copying some items unchanged and modifying or deleting others. For example, the following query creates a modified copy of the initial context document, deleting `@foo` attributes, changing the value of `@bar` attributes and adding to `<zoo>` elements a `<timeStamp>` child element:

```
declare function local:edit($n as node())  as node()? {
   typeswitch($n)

   case document-node() return
      document {$n/node() ! local:edit(.)}

   case element(zoo) return
      <zoo>{
         $n/@* ! local:edit(.),
         <timestamp>{current-dateTime()}</timestamp>,
         $n/* ! local:edit(.)
      }</zoo>

   case element() return
      element {node-name($n)}{$n/(@*,node())!local:edit(.)}

   case attribute(foo) return ()
   case attribute(bar) return attribute bar {upper-case($n)}
   default return $n
};
local:edit(.)
```

The code pattern “recursive function consisting of a `typeswitch` expression” enables convenient implementation of many transformation tasks. It cannot, however, compare with the possibilities offered by XSLT and its template model ([19]). XQuery 3.1 ([15]) therefore introduces an `fn:transform` function which invokes an XSLT stylesheet and returns the transformation result. As the result of an XSLT transformation thus becomes the value of an XQuery expression, XSLT-powered transformation is seamlessly integrated into the XQuery processing, as this example demonstrates:

```
transform(map{'stylesheet-uri' : '/apps/foo.xsl',
              'source-node'    : doc('/data/msgs/msg1.xml')})
//flight
```

The stylesheet may also be generated on the fly, by the XQuery code itself:

```
declare function local:buildXslt(…) {…};

transform(map{'stylesheet-node' : local:buildXslt(…),
              'source-node'     : doc('/data/msgs/msg1.xml')})
//flight
```

The XQuery programmer thus may resort to XSLT code as an additional means of expressing transformation.

Modification

The XQuery Update Facility, specified by a self-contained W3C recommendation (17), provides a set of expressions which do not produce a value, but effect an in situ modification of existent resources. Besides, the facility includes two expressions (`transform` and `copyModify`) which apply update expression to *copies* of existent nodes, returning the result and thus enabling the construction of new structure by expressing the modification of existent structure. The benefits of the Update Facility

are thus twofold – in situ update and elegant construction of new values by expressing modification. Note that the Update Facility is the only non-proprietary way to achieve in situ updates of documents residing in an XML database or the file system.

A striking feature of update expressions (used for in situ update or the construction of new values) is the clean separation and seamless integration of selection and modification:

- a sub expression selects the target nodes to be modified
- another sub expression specifies the modification itself

as illustrated by a code example:

```
insert node
   <Underwriter>?</Underwriter>
as last into
   //booking[@btype='insurance'][not(Underwriter)]
```

As a consequence, the transition from single document modification to bulk modification is as easy as the transition from single document navigation to bulk navigation, for example:

```
for $doc in file:list('/data/msgs', true(), '*.xml') !
   concat('/data/msgs/', .) ! doc(.) return

insert node
   <Underwriter>?</Underwriter>
as last into
   $doc//booking[@btype='insurance'][not(Underwriter)]
```

A query effecting in situ updates cannot produce any result value: a query is either updating and not producing a result, or producing a result and not updating. Such a limitation can make it impossible to solve a complex task of data integration with a *single* query, and the question arises how several queries can be orchestrated into a coherent processing. The problem is related to the general problem of how to reconcile the XQuery processing model with side effects as they are required within a larger processing context. We will return to this subject (the section called “Issue: the snapshot model”).

Resource exploration

Data integration often requires some preparatory exploration of little or unknown resources. XQuery’s capabilities of bulk navigation, aggregation and document construction enable a very efficient development of general as well as special purpose reporting tools. For example, let us assume we are interested in the XML resources found in some directory tree (a directory and all direct and indirect sub directories). The following query reports all document types (root element names and namespaces):

```
declare variable $dir external;
declare variable $pattern external := '*.xml';
string-join(sort(distinct-values(
   file:list($dir, true(), $pattern) ! concat($dir, '/', .) !
   doc(.)/*/concat(local-name(.), '@', namespace-uri(.))
)), '&#xA;')
```

The following query yields for an input document a list of all element names:

```
sort(distinct-values(//*/local-name()))
```

and a list of all element data paths can be obtained like this:

```
string-join(sort(distinct-values(
   //*/string-join(ancestor-or-self::*/local-name(), '/')
)), '&#xA;')
```

An extended report of all document types found in a directory tree, including for each type a list of all data paths, can be achieved by few lines of XQuery code:

```
declare variable $dir external;
declare variable $pattern external := '*.xml';
string-join(
   for $doc in
      file:list($dir, true(), $pattern) ! concat($dir, '/', .) ! doc(.)
   group by $doctype :=
      $doc/*/concat(local-name(.), '@', namespace-uri(.))
   let $paths := sort(distinct-values(
      $doc//*/string-join(ancestor-or-self::*/local-name(), '/')))
   order by $doctype
   return ($doctype, $paths ! concat('   ', .)
), '&#xA;')
```

Note that the report is document type oriented, not document oriented: it integrates the information content of all documents sharing a particular document type and found within a directory tree.

XQuery is also well-suited for writing small ad hoc queries answering specific questions. Example: what is the minimum / maximum number of <booking> elements contained by <msg> elements, considering the documents located in a given directory tree? The following query gives the answer:

```
let $dir := '/data/msgs'
let $counts :=
   file:list($dir, true(), '*.xml') ! concat($dir, '/', .) !
   doc(.)//msg/count(.//booking)
return (min($counts), max($counts))
```

The following query can be used for listing the names of files matching some particular condition – for example containing a <msg> element which does not contain any <booking> elements:

```
let $dir := '/data/msgs' return

string-join(('files: ', '======', file:list($dir, true(),'*.xml ') ! concat($di
    //msg[not(.//booking)]
 /root()/document-uri(.)), '&#xA;')
```

To get the names of files matching a different condition, just replace the following expression by a different one:

```
//msg[not(.//booking)]
```

Resorting to an extension function for dynamic evaluation of XQuery expressions - as offered by several XQuery processors - the condition can be turned into a string parameter and the query is truly generic. The following code uses an extension function (`xquery:eval`) supported by the BaseX processor:

```
declare variable $dir external := '/data/msgs';
declare variable $cond external := '//msg[not(.//booking)]';
```

```
string-join(('files: ', '======', file:list($dir, true(),'*.xml ') ! concat($
   /xquery:eval($cond, map{'':.})
/root()/document-uri(.)), '&#xA;')
```

As illustrated by several examples, XQuery is an appropriate language for creating tools exploring XML resources and answering general as well as specific and ad hoc questions.

Resource validation

Data integration often involves the validation of resources against expectations. Validation reports may be a primary goal in its own right, and the exclusion of invalid resources from further processing may be required in order to ensure data integrity, operational robustness and efficiency.

Schema validation

The standard function library of XQuery (13) does not yet contain any functions for validating XML documents against XSD schemas. As an optional feature, the XQuery language does include a `validate` expression, which returns a copy of the validated document with type annotations added, or raises an error in case of validation errors. Two points should be noted. First, the goal to gather information about validity is different from the goal to construct a type-annotated document, and it is the former goal which is common, not so much the latter. Second, implementation of the `validate` expression is not common among popular XQuery processors, especially among open source products. We think that an XQuery function offering XSD validation is desirable in general, and highly desirable in the context of data integration. For the time being, several XQuery processors offer vendor-specific extension functions for XSD-validation of XML documents.

Bulk schema validation

The typical signature of an extension function for XSD-validation has two parameters, one supplying an instance document and one supplying a schema document. Interestingly, such a signature suffices to enable bulk validation – the validation of many instance documents against a collection of candidate schemas, applying to each instance document the appropriate schema document and integrating the results into a single report. The ease of accomplishing this follows from the navigational capabilities and the aggregation-friendliness of XQuery:

- for a given instance document select the appropriate schema by matching the namespace and local name of the document root element against the schema target namespace and the names of the top-level element declarations
- collect single document validation results in a simple FLWOR expression
- integrate the single document validation results using an embedded expression

A typical implementation will comprise only few lines of code. The following query is an example, relying on the BaseX extension function `validate:xsd-info`. The query validates all XML documents found in a directory tree and matching a file name pattern, against a schema which is for each document automatically selected from all schemas found in a different directory tree:

```
declare variable $dir external;
declare variable $fname external;
declare variable $xsdDir external;

let $xsds :=
   file:list($xsdDir, true(), '*.xsd') ! concat($xsdDir, '/', .) ! doc(.)
let $reports :=
   for $uri in file:list($dir, true(), $fname) ! concat($dir, '/', .)
   let $doc := doc($uri)
   let $xsdURI := $xsds[xs:schema/xs:element/@name = $doc/local-name(*)]
```

```
                        [string(xs:schema/@targetNamespace) eq $doc/namespace
                        [1]/document-uri(.)
   let $msgs := validate:xsd-info($doc, $xsdURI) ! <msg>{.}</msg>
   return
      <report doc="{$uri}" xsd="{$xsdURI}" errorCount="{count($msgs)}">{
         $msgs
      }</report>
return
   <reports>{$reports}</reports>
```

Expression-based validation

It is a well-known fact that grammar-based validation like XSD-validation cannot cover (or not cover well) typical business rules, which constrain the presence and data contents of items in response to the presence and contents of other items. For example, XSD 1.0 (18) cannot express the rule that an element must either have a `@fooRef` attribute and no child elements, or have child elements but no such attribute. Similarly, there is no way to assert that an element must have an `@approvedBy` attribute if the <amount> child element contains a value larger 1000.

It is interesting to note how very simple it is to perform such checks using XQuery expressions. In this context one should remember the core concept of the Effective Boolean Value (15), which is a generic mapping of arbitrary expression values to a Boolean value. Consider the following user-defined XQuery function which checks a sequence of XML documents against two rules and delivers an integrated validation report:

```
declare function local:checkDocs($docs as document-node()*)
     as element(errors)? {
   for $doc in $docs
   let $errors := (
      <error msg="No product found."/>
         [empty($doc//*:Product)]
      ,
      <error msg="Travel start date &gt; end date."/>
         [$doc//travel[@start gt @end]]
   )
   return
      if (not($errors)) then () else
      <errors uri="{document-uri($doc)}">{$errors}</errors>
};
```

The code follows a simple pattern, within which the only specific elements are the actual check expressions and error messages. Therefore it is simple to write a generic version of such a check function, which is configured by a collection of expression/message pairs. As an example, any collection of assertions stored in a file like the following:

```
<assertions>
   <assertion msg="No product found."
              expr="empty(//*:Product)"/>
   <assertion msg="Travel start date &gt; end date."
              expr="//travel[@start gt @end]"/>
</assertions>
```

could be applied to the contents of a directory tree by the following generic query:

```
declare variable $dir external;
declare variable $fname external;
```

```
declare variable $assertions external;

declare function local:checkDocs($docs as document-node()*,
                                 $assertions as element(assertion)*)
        as element(errors)* {
   for $doc in $docs
   let $context := map{'':$doc}
   let $errors :=
      $assertions[xquery:eval(@expr, $context)]/<error msg="{@msg}"/>
   return
      <errors uri="{document-uri($doc)}">{$errors}</errors> [$errors]
};

let $docs := file:list($dir, true(), $fname) ! concat($dir, '/', .) ! doc(.)
let $checks := local:checkDocs($docs, doc($assertions)/*/*)
return
   <errorReport>{$checks}</errorReport>
```

The example uses a vendor-specific extension function (`xquery:eval`, provided by the BaseX processor) for the dynamic evaluation of XQuery expressions. Similar functions are offered by various XQuery processors (e.g. 1, 2, 9). As of this writing, the standard function library of XQuery (13) does not yet contain any similar function, and it is to be hoped that future versions will amend this. As the example demonstrated, dynamic evaluation enables the use of XQuery expressions as *configuration parameters*, on which very powerful and flexible XQuery programs may be based. Note that the example was kept very simple in order to be illustrative. A tool for production use would be more elaborate. It might for example support the arrangement of assertions in a tree whose inner nodes are associated with guard expressions, controlling whether the respective sub-tree of assertions is evaluated or ignored. Another important refinement would be support for looping validation: a check expression can be accompanied by a "target expression" providing a range of context items, to be used successively during repeated evaluation of the check expression.

Resource access

The previous section took resource access for granted and concentrated on the evaluation of resource contents. The current section analyses the capabilities of XQuery to access resources.

Introduction

Data integration is based on the ability to access data resources – entities of information like files, database entries and service responses. In the context of XQuery, any information is represented by XDM values, and access to a resource is equivalent to an XDM value representing the resource contents:

resource access = XDM value representing the resource contents

Such a value is either supplied from without, or it is received as the value of a function call. If supplied from without, it is the initial context item (accessible via the context item expression) or an external parameter (accessible via variable reference expression). In principle it is possible that a resource provided from without would not be accessible via function call, but such scenarios are uncommon and of little interest for the purposes of this discussion. Therefore the capability of resource access can be equated with the availability of functions returning a resource representation.

Resource access patterns

For a given resource, different XDM representations may be considered. For example, a JSON document might be represented (a) as a string, (b) as nested maps and/or arrays, (c) as an XML document. As an unstructured representation (string) is far less valuable for processing purposes than

a structured representation, an evaluation of access capabilities must distinguish between access as unstructured text and access as structured information. If, however, for a given media type a function is available which parses a string into a structured representation, access as unstructured text is virtually equivalent to access as structured information, as text access and text parsing can be combined by nested function calls, for example:

```
json-to-xml(unparsed-text('msg.json'))
```

Resource accessing functions can be categorized by their main input:

- access by URI – a function which receives a URI parameter and returns a resource representation
- access by text – a function which receives the text of a resource as a string and returns a structured representation
- access by message – a function which receives a request message plus a recipient URI and returns a response message
- access by query – a function which receives a query plus connection data and returns a search result

Access by URI does not constrain the URI scheme and hence the access protocol (file, HTTP, ftp, …). The function signature treats the URI as an opaque item of information, and the actual scope of access depends on the XQuery processor as well as the operational context (e.g. internet and WLAN access). In practice, access to the file system and the resolving of URIs which are HTTP or ftp URLs can be expected. If the XQuery processor is part of an XML database system, URI access is also used for accessing database entries.

Access by text is usually applied to the result of access by a different pattern – by URI, message or query.

Access by message is important as it includes all *web services* requiring a request message sent per HTTP POST. Namely, this category contains all SOAP services. In a service-oriented environment, the appropriateness of a language for data integration may stand and fall with access to all web services, including SOAP services.

Access by query is required in order to work with *databases*, unless they are exposed as web services. In many environments, relational and/or NoSQL databases play a dominant role, so that resource access by query is an indispensible prerequisite for data integration.

Scopes of availability

Resource access functions can also be categorized by the scope of availability:

- standard functions – provided by every conformant XQuery processor (13)
- EXPath functions – unofficially standardized by a specification produced by the EXPath community group (3)
- vendor functions – provided by XQuery processors as a product-specific extension function

Resource access functions

The following table summarizes resource access provided by XQuery functions. Besides standard functions (13), several extension functions are also taken into account, provided they have been defined by the EXPath community group or are available in one of the following XQuery processors: SaxonHE, SaxonPE, eXistDB, BaseX. Functions defined by the EXPath group are marked by the postfix (EXP), BaseX defined functions by the postfix (BSX).

Table 1.

Resource access functions - an overview. The Source value * indicates that the function is part of the XQuery standard / version 3.1; Source value (*) indicates an EXPath-defined function and other values are the names of the respective XQuery processor. The names of non-standard functions are displayed in italics.

Recource type	**Input**	**Function**	**Source**	**XDM representation**
XML document	URI	fn:doc	*	document-node()
	URI	fn:collection	*	node()*
	text	fn:parse-xml	*	document-node()
	text	fn:parse-xml-fragment	*	document-node()
HTML document	text	*html:parse*	BaseX	document-node()
	text	*util:parse-html*	eXist-db	document-node()
	text	*xdmp:tidy*	MarkLogic	document-node()
JSON document	URI	fn:json-doc	*	map|array
	text	fn:parse-json	*	map|array
	text	fn:json-to-xml	*	document-node()
	text	*json:parse*	BaseX	document-node()
	text	*util:parse-json*	eXist-db	document-node()
CSV	text	*csv:parse*	BaseX	document-node()
text	URI	fn:unparsed-text	*	xs:string
	URI	fn:unparsed-text-lines	*	xs:string*
binary	URI	*file:read-binary*	(*)	xs:base64Binary
	URI	*util:binary-doc*	eXist-db	xs:base64Binary
	URI	*fetch:binary*	BaseX	xs:base64Binary
archive-entry	URI	*archive:extract-text*	(*)	xs:string
	URI	*archive:extract-binary*	(*)	xs:base64Binary
	URI	*xdmp:zip-get*	MarkLogic	document-node() | text() | xs:base64Binary
SQL result set	query	*sql:execute*	BaseX	element()*
MongoDB result set	query	*mongo:find*	(*)	element()*
HTTP response	message	*http:send-request*	(*)	element()|xs:string

Note that XML documents can be accessed via URI and also by parsing XML text. The latter possibility is important, as the text of an XML document is not necessarily an independent resource, but may be *embedded* in another resource. It may, for example, be retrieved from a relational database column or a NoSQL database field, and sometimes unparsed XML text is even encountered as text content of XML elements. An example of XML containing unparsed XML are the test definitions and test logs created by JMeter (6), a popular open source framework for testing web applications and other IT components.

XQuery is focused on text resources. Nevertheless, also binary data can be represented by an XDM item, and therefore can be accessed. Using the EXPath-defined `file` module (4) or the BaseX-defined `archive` module, the contents of binary resources are read into an `xs:base64Binary` representation, and binary data can also be created by preparing an `xs:base64Binary`

representation and writing it into a file or archive, which then receives the binary data corresponding to the `xs:base64Binary` representation.

Standard versus extension functions

The table of resource access functions (Table 1, “

Resource access functions - an overview. The Source value * indicates that the function is part of the XQuery standard / version 3.1; Source value (*) indicates an EXPath-defined function and other values are the names of the respective XQuery processor. The names of non-standard functions are displayed in italics.

”) shows a fairly broad scope of resource access, covering various media types (XML, HTML, JSON, CSV, plain text) and data platforms (file system, HTTP, ftp, web services, relational databases, NoSQL databases, archives). One may regard this scope as a reasonable base for tackling tasks of data integration in serious.

But the scope of resource access depends to a considerable degree on extension functions, some EXPath-defined, others vendor-specific. Relying on standard functions, only XML and JSON documents can be accessed as structured information, and the following data platforms are *inaccessible*: relational databases, NoSQL databases, SOAP web services, archives. Besides, HTML and CSV resources can only be accessed as plain text, not as structured information. On the other hand, resource access and structured representation play a crucial role in data integration. It may be concluded that the XQuery working group of the W3C has not yet identified data integration as a key field of XQuery usage.

XDM binding

The previous section reported the current support which XQuery offers for accessing resources. The account should be regarded as a snapshot capturing a particular version of the language specification (3.1), particular versions of popular XQuery processors and the results of EXPath activities up to a particular date. Now we attempt to treat the topic of XQuery access to resources in a more abstract way, analyzing potentials and limitations.

Introduction

An XQuery program is an expression which is recursively composed of sub expressions. XQuery expressions have an input - expression operands and possibly the context item - and an output, which is the expression value. Every input value and the output value are XDM values. Any resource *access* is therefore implemented as an expression using an XDM representation of the resource as operand or context item.

Using XQuery 1.0, the only accessible resources are XML documents, which the function `fn:doc` represents as a node tree. XQuery versions 3.0 and 3.1 broadened the scope of resource access significantly: any text-based resource can be accessed as a string, and JSON documents can be accessed as structured information. EXPath-defined as well as vendor-specific extension functions add further possibilities (the section called “Resource access”). We prefer to regard these emerging functions not as independent and unrelated features, but as aspects of a long-term evolution shifting the XQuery language from an XML processing language towards a platform of information processing and data integration.

If such a development should indeed be pursued by the W3C XQuery working group and XQuery implementers, the representation of resources by XDM values should be based on general concepts which might help to assess alternative approaches and protect consistency.

XDM binding, XDM equality

A structured representation of resources is the foundation enabling navigation and selection of information. But what exactly does the term “representation of a resource” mean? Are there general

rules how to define and characterize representations? In this context we are not interested in the trivial case of a single string (or a sequence of strings corresponding to text lines). We contemplate structured representations, composed of distinct items which represent the logical building blocks of the resource.

In general, resource representation by an XDM value should be “lossless”, but lossless means without loss of *significant* information, rather than a faithful representation of the exact sequence of characters. When a resource is parsed into an XDM representation and subsequently serialized into an instance of the resource’s media type, the result need not be the same sequence of characters as the original instance, but it should have the same information content. If, for example, the resource is a JSON document, the result of round-tripping may differ with respect to whitespace characters surrounding value separators, but it should not differ with respect to member names, member values or datatypes.

It is problematic to define lossless representation in general terms. We cannot assume all resource formats of interest to be described by a rigorous information model defining the abstract information content conveyed by the text of a format instance. Even when the format is defined very precisely, as is the case with JSON defined by RFC 7159 (5), the distinction between significant and not significant information is not necessarily unambiguous. In RFC 7159, the significance of object member ordering is discussed as follows:

“ *JSON parsing libraries have been observed to differ as to whether or not they make the ordering of object members visible to calling software. Implementations whose behavior does not depend on member ordering will be interoperable in the sense that they will not be affected by these differences.* ”

This reads like a recommendation not to rely on order, but we find no definitive statement declaring the insignificance of order. (Interestingly, the popular NoSQL database MongoDB uses JSON in an order-sensitive way.) The lesson to be learned from this example is that the *definition* of an XDM representation should be explicit about all aspects of information which may not be preserved.

In order to give the definition of resource representations a formal footing, we propose the introduction of a new term, the **XDM binding** of a data format, defined as follows. Given a data format F, an XML binding consists of two XPath functions, a parsing function (P) and a serialization function (S). The parsing function maps a format instance to an XDM instance, the serialization function maps XDM instances to format instances. Parsing and serialization functions must together meet the following constraints:

1. P maps every format instance to an XDM instance

2. S maps every XDM instance X1, which has been obtained by applying P to a format instance F1, to a second format instance F2 so that applying P to F2 yields an XDM instance X2 so that X1 and X2 are deep-equal (as defined in (13)); formally:

   For every string F1 which is a valid instance of format F, the following expression is `true`:

   ```
   deep-equal (P(F1), P(S(P(F1))))
   ```

An XDM binding is identified by a URI which is obtained by concatenating the namespace and local name of the parsing function.

Examples: The specification “XPath and XQuery Functions and Operators 3.1” (13) defines two XDM bindings for the media type “JSON”:

- http://www.w3.org/2005/xpath-functions#parse-json

- http://www.w3.org/2005/xpath-functions#json-to-xml

Note that this abstract definition translates the informal concept of “lossless representation” into the formally defined deep-equality of XDM values. In this sense, an XDM binding is neither lossless or not lossless, but axiomatically defines a binding-based equivalence between format instances: given a particular XDM binding, two format instances are equivalent if their representing XDM values are deep-equal. It is a different question in how far this equivalence matches the operational requirements of a lossless representation.

The equivalence of format instances which have deep-equal XDM representations should be captured by a formally defined term. Given a data format for which an XDM binding B has been defined, two instances of this data format are said to be **XDM-equal (by binding B)** if the XDM values obtained from function B are deep-equal. Inspecting the XDM bindings `fn:json-doc` and `fn:json-to-xml`, one may observe these facts:

- XDM-equal (by `fn:json-to-xml`) implies XDM-equal (by `fn:parse-json`), but not the other way around, as differences in the order of object members preclude XDM-equality by `fn:json-to-xml`, but not by `fn:parse-json`.
- Two JSON documents which are XDM-equal by at least one of these bindings are classified as "without significant differences" by the JSON spec RF7159.

The proposed definition of an XDM binding has an interesting implication. Given an XDM binding, every XDM instance can either be mapped to a format instance in a round-trippable way (that is, subsequent parsing would yield an XDM value deep-equal to the original one), or *cannot* be mapped to a format instance.

An XDM binding creates the possibility to achieve the transformation of format instance F1 into an instance F2 by transforming the XDM instance X1 obtained for F1 into an XDM instance X2 which would be obtained by parsing F2. The result of serializing X2 may not be identical to F2, but it is XDM-equal to F2. The imprecision incurred by constructing a resource's XDM representation, rather than an instance of the resource format itself, is bounded by the XDM-equality of the alternative results.

From the definitions of XDM binding and XDM-equality, we derive the definition of **format resolution**. Given a data format with two XDM bindings B1 and B2, the bindings are said to have equal format resolution if every pair of format instances F1 and F2 which is XDM-equal by B1 is also XDM-equal by B2, and vice versa. One binding B2 is said to have *higher* format resolution if every pair of format instances which is XDM-equal by B2 is also XDM-equal by B1, and some pairs are XDM-equal by B1, but not by B2. As an example, the XDM binding `fn:json-to-xml` has a higher format resolution than the binding `fn:parse-json`.

From the definition of format resolution we derive the further definition of **XDM binding consistency**, which SHOULD be preserved when for a given data format several XDM bindings exist. These bindings are consistent if for any pair of bindings the bindings have either equal format resolution, or one binding has a higher format resolution.

An example of a further data format for which different XDM bindings are conceivable is CSV. The content of a CSV record might be captured by an array of arrays (representing the sequence of rows), and it might be represented by an XML document. Both bindings should probably have equal format resolution; for example, they should both capture the character sequences of all values, and they should both ignore differences of syntactical representation like the choice of the separator character.

XML binding

The only complex items defined by the XQuery data model XDM (16) are nodes, maps, arrays and functions. A structured representation of a resource is therefore expected to be either a node tree or a set of nested maps and/or arrays which is rooted in a single map or array. (Note: a conceivable exception would be a *sequence* of items, possibly evaluated by special functions playing the role of an interface to a logical internal structure.) An XDM binding which represents the resource as a node tree we call an **XML binding**. This section compares XML binding with its main alternative, the binding to maps and/or arrays.

Consider the following section of a FLWOR expression, which constructs different XDM representations of JSON resources:

```
let $m := json-doc('r1.json')                        (: map/array :)
let $s := unparsed-text('r2.json') ! json-to-xml(.)  (: XML, per standard fu
let $b := unparsed-text('r3.json') ! json:parse(.)   (: XML, per BaseX exten
```

The first representation (`$m`) consists of nested maps and arrays. Every JSON item – object, array, string, number, literal – is represented by a distinct map entry or array member. It can be individually addressed by chaining map and array lookups, which amounts to specifying a "path" of map entry names and/or array index values. Every JSON item at an *exactly known location* can thus be retrieved fairly easily. Examples:

```
let $customerFirstName := $m?customer?firstName
let $bookingId1    := $m?services?booking?1?bookingID
let $bookingIds    := $m?services?booking?*?bookingID
let $bookingIdLast := $m?services?booking?*[last()]?bookingID
```

Such simple retrieval requires exact knowledge of the internal document structure. A minor issue is that array access requires a separate path step (for example "`?1`" or "`?*`"). Therefore items cannot be addressed without exact knowledge which object members along the path are single-valued and which are array-valued. If, for example, the `booking` member is a single object, the following path might retrieve the `bookingID` member of the booking object:

```
$m?services?booking?bookingID
```

But if bookings are multiple – if the `booking` member is an array of objects - the path must be adapted:

```
$m?services?booking?*?bookingID
```

A *major* issue is the lack of built-in support for navigation along axes, as offered by path expressions. Assume we do not know the exact document structure, and we want to extract all booking ID values, defined as the `bookingID` member values of objects which are themselves `booking` member values. Dealing with a comparable XML document, the retrieval would correspond to this path expression:

```
//booking/bookingID
```

Dealing with a map/array representation of a JSON document, the retrieval would require application code similar to the following:

```
declare function local:descendant($item as item(), $fieldName as xs:string)
      as item()* {
   typeswitch($item)
   case map(*) return (
      let $field := $item($fieldName)
      return
         if ($field instance of array(*)) then $field?* else $field,
      map:keys($item)[. ne $fieldName] !
         local:descendant($item(.), $fieldName)
   )
   case array(*) return $item?* ! local:descendant(., $fieldName)
   default return ()
};
local:descendant($data, 'booking')?bookingID
```

The effort is considerable, and such navigation is error-prone. In comparison, using the standard XML representation of the JSON document (`$s`) as produced by function `fn:json-to-xml`, the booking IDs are retrieved by a simple, though not very intuitive, expression:

```
$s//*[@key eq 'booking']/(self::j:map, self::j:array/j:map)/*[@key eq 'booki
```

Using the vendor-specific XML representation of JSON offered by BaseX (`$b`), retrieval becomes even simpler:

```
$b//booking/(., _)/bookingID
```

The examples demonstrate that XQuery's outstanding support for data navigation applies to XML structures (node trees), but does not apply to nested maps and arrays. Remembering the extent to which XQuery's capabilities for data integration are based on navigation capabilities, the XDM representation of non-XML resources gains great importance. The standard function `fn:json-to-xml` as well as vendor-specific extension functions like BaseX-defined `json:parse` and `csv:parse` demonstrate that various non-XML media types may be represented as XML node trees, and the coexistence of the standard functions `fn:json-to-xml` and `fn:json-doc` shows that for a given media type different structured representations are possible. We acknowledge the fact that a map/array representation has its own benefits compared with a node tree representation – it is more light-weight, which means that it consumes less memory and simple access operations are probably faster. But we also note that the question whether a given media type can be represented by node trees or not decides whether the media type in question is or is not *within scope of XQuery's particular navigation capabilities*, and this comes close to saying that the availability of a node tree representation decides whether the media type is or is not *within scope of XQuery's particular integration capabilities*.

The role of REST

REST services can play a key role in data integration. They provide an infrastructure which may

- reorganize the URI space occupied by a set of related resources, creating clearly structured and intuitive URI trees
- enhance the resource space by binding new URIs to resource derivatives (reports, aggregations, …)

REST services can be implemented and consumed using many programming languages. XQuery is a particularly interesting choice for two main reasons:

- as a REST client, REST calls can be seamless integrated into data navigation (e.g. `fn:doc(REST-URI)//bookingID`)
- as a REST service implementation, the combined capabilities of navigation and construction enable a very elegant translation of resources into responses

As a general rule, REST services lower the barrier to access resources, and for this reason they increase the opportunities for XQuery to navigate and integrate resource contents. Several XQuery processors support the RESTXQ API (8), a set of XQuery *annotations* enabling very elegant implementation of REST services. This means that XQuery can be used to create the very infrastructure which increases the usefulness of XQuery to a particular degree. When facing challenges of data integration, commitments to using XQuery and to building REST services are synergetic.

XQuery and RDF

This section explores possible relationships of XQuery and RDF.

RDF and data integration

RDF is an information model which is almost by definition highly relevant to data integration. The basic unit of information is a triple, consisting of subject, predicate and object, where subject and predicate are URIs and the object is either a URI, a literal value (e.g. a string, a number or a date) or a so-called blank node, which is a URI-less subject of further triples. A crucial aspect is the independent and self-contained character of a triple: the information conveyed by a triple does not depend on a surrounding data resource and a location within it, as is the case for example with XML nodes, JSON

object members and CSV cells. Triples are implicitly related to each other, as one triple's object may be a URI which is another triple's subject, so that a set of triples is a graph consisting of nodes (URIs, literal values and blank nodes) and connecting arcs which are predicates.

Triples are typically constructed by an evaluation and transformation of non-RDF resources – e.g. spreadsheets, relational tables, XML and JSON documents. As the information content of a triple is self-contained and bears no traces of the original "source" from which it has been constructed, a graph of triples can be evaluated without considering any information except for the triples themselves. This means simplicity: given a set of triples, the number, physical distribution and media types of data resources which have contributed to the construction of the set has become invisible. A triple set can thus be viewed as a homogeneous extract gained from the processing of (potentially) multiple and heterogeneous data sources. It is a special kind of resource whose evaluation *is* an operation of data integration.

When evaluating the appropriateness of a programming language for data integration, the relationship between the language in question and RDF should be considered. Two important aspects of this relationship are:

- The use of RDF data as resources
- The creation of RDF data by evaluating non-RDF resources

XQuery and SPARQL

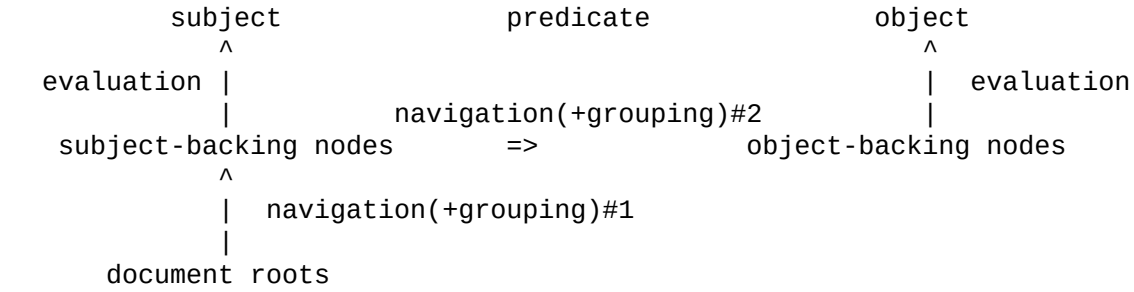
RDF data are queried using a special query language, SPARQL (12). A special type of web service, called SPARQL endpoint, receives SPARQL queries and returns the query result in a format of choice (e.g. XML/RDF (11) or Turtle (10)). SPARQL endpoints should be regarded as a particular kind of resources. Unfortunately, there is currently very little built-in support for XQuery access to SPARQL endpoints. As SPARQL endpoints are usually accessible via HTTP, the EXPath-defined `http:send-request` function can be used for pulling SPARQL result sets into XQuery processing.

It should be noted that XQuery (as any other programming language) might put SPARQL resources to special uses: it might query triple stores in order to discover the URIs of interesting (non-RDF) resources and subsequently retrieve these resources. This usage pattern – "SPARQL as a resource broker" - requires the availability of appropriate triple stores, which might be on purpose created as part of an infrastructure supporting data integration at a large scale. We regard this as an interesting approach, to be considered by organizations maintaining many and heterogeneous data resources. A seamless integration of SPARQL-based resource access into XQuery might be achieved along the lines described in (7), which provide a generic model for integrating non-XML technologies as resource brokers into XQuery.

XQuery as a triple factory

In earlier sections we discussed the excellent capabilities of XQuery for navigation and the construction of information. These capabilities can be leveraged for the production of RDF triples from XML data: XQuery can be used as a powerful triple factory. In this section, we propose a generic model of the transformation of XML resources into a triple set. The model reduces the precise specification of such a transformation to a small number of XQuery expressions, each executing a particular sub task which the model integrates into an overall process of transformation.

So let us consider the goal of transforming XML resources into a set of RDF triples. In this scenario we can assume that the subject and the object of a triple are both related to XML nodes, and this enables us to view the creation of triples as a combination of navigation and the evaluation of expressions mapping nodes to URIs and values. Generally speaking, the subject of a triple is a resource which is represented by one or more **subject-backing nodes**. Similarly, the object URI or value is derived from one or more **object-backing nodes**. Subject-backing nodes are found by navigation starting at the document roots, and object-backing nodes are found by navigation starting at subject-backing nodes. Schematically:



Inspecting the two navigations more closely – (1) document roots to subject-backing nodes, (2) subject-backing nodes to object-backing nodes – we observe that they comprise an expression yielding nodes and an optional grouping of these nodes. The grouping of subject-backing nodes becomes necessary if we deal with subjects which have more than one subject-backing node. In general, the navigation step involved in finding the subject-backing nodes yields the *union* of all subject-backing nodes. In most cases each of these nodes serves as the single subject-backing node of a distinct subject, and grouping is not necessary. But if some subjects correspond to several subject-backing nodes, we must partition the union of all subject-backing nodes into subsets corresponding to distinct subjects, and we can define this partitioning by an expression used in a `group by` clause of a FLWOR expression iterating over the union of all subject-backing nodes:

```
for $node in $allSubjectBackingNodes
group by $key := SUBJECT_GROUPING_EXPRESSION
return array{$node}
```

Similar considerations apply to the navigation yielding object-backing nodes. The navigation step yields for each individual subject (represented by one or several subject-backing nodes) the *union* of all object-backing nodes, and in the common case each node represents the single object-backing node of a distinct object. If some objects are backed by several nodes, we partition the union of nodes with the help of a `group by` expression:

```
for $node in $allObjectBackingNodes
group by $key := OBJECT_GROUPING_EXPRESSION
return array{$node}
```

To illustrate the concept of an optional grouping, consider a transformation of XSD documents into (a) a set of triples describing schema elements, as well as (b) a second set of triples describing target namespaces. The subject-backing node of a schema element is the schema element itself, and the subject-backing nodes are obtained by applying to each input document the expression

```
xs:schema
```

Each node thus obtained represents a distinct subject, and grouping of subject-backing nodes is not required. The subject-backing nodes of a target namespace, on the other hand, comprise all schema elements sharing the target namespace. In this case, the subject-backing nodes are obtained in two steps, a navigation step and a grouping step. First, the union of all subject-backing nodes is obtained by applying again to each input document the expression

```
xs:schema
```

These nodes are then grouped, using the target namespace as a grouping key. Assuming that the schema elements are bound to a variable `$node`, the grouping expression is then:

```
$node/@targetNamespace
```

To summarize, subject-backing nodes and object-backing nodes are determined with the help of four expressions, two of which are optional:

- subject-provider expression, evaluated in the context of document roots, yielding the union of all subject-backing nodes
- subject-grouper expression, used in a `group by` clause in order to group subject-backing nodes
- object-provider expression, evaluated in the context of an individual subject's subject-backing nodes, yielding for that subject the union of all object-backing nodes
- object-grouper expression, used in a `group by` clause in order to group object-backing nodes

The construction of triples requires yet a mapping of subject-backing nodes to a subject URI, and a mapping of object-backing nodes to an object URI or an object value. The general model therefore includes three more expressions:

- subject-uri-constructor expression, evaluated in the context of an individual subject's subject-backing nodes and yielding the subject URI
- object-uri-constructor expression, evaluated in the context of an individual object's object-backing nodes and yielding the object URI (if the object is a URI, rather than a value)
- object-value-constructor expression, evaluated in the context of an individual object's object-backing nodes and yielding the object value (if the object is a value, rather than a URI)

A problem should be acknowledged. The mapping of nodes to URIs may not be possible if only relying on the content of the input documents. For example, nodes may be assigned URIs containing a counter whose value is determined by incrementing the current maximum value, which cannot be derived from document contents. A general model of transforming XML documents into triple sets can take this into account by assuming the availability of two XPath functions:

```
subjectUriConstructor($type as xs:anyUri, $backingNodes as node()*) as xs:any
objectUriConstructor ($type as xs:anyUri, $backingNodes as node()*) as xs:any
```

Up to this point we have considered a set of triples which describe a set of similar subject resources in a uniform way. The subject resources are assumed to

- have the same type (RDF type)
- have subject-backing nodes reached by the same navigation (starting at document roots)
- have the same mapping of subject-backing nodes to subject URIs
- have the same set of (considered) predicates
- have for each (considered) predicate the same navigation to object-backing nodes
- have for each (considered) predicate the same mapping of object-backing nodes to an object URI or value

The construction of such a coherent set of triples can be specified by a set of URIs and XQuery expressions:

- RDF type of the subject resources (a URI)
- subject-provider expression
- subject-grouper expression (optional)
- subject-uri-constructor

- a set of predicate URIs
- for each predicate URI:
  - object-provider expression
  - object-grouper expression (optional)
  - object-uri-constructor or object-value-constructor

Such an entity of information we call a **triple set constructor**. The generic model of an XML to RDF transformation assumes a configuration which consists of one or more triple set constructors. Such a configuration we call a **triple factory definition**.

Our model of transforming XML documents into triple sets is completed by

- an XML schema defining the XML representation of a triple factory definition
- further information concerning the various expressions of a triple set constructor, in particular:
  - the dynamic context in which the expression is evaluated
    - context item
    - variable bindings
  - trigger information – the expression is evaluated once per which entity

The following listing shows an example of a triple factory definition. It specifies the creation of a a set of triples describing the `xs:schema` elements found in a set of XSDs. The triple set constructor makes each `xs:schema` element the subject of various triples providing target namespace (predicate `x:tns`), document URI (predicate `x:docURI`), imported schemas (predicate `x:imports`) and other properties.

```
<tripleFactory prefixes="x:http://example.org/rdf/xsd#"
               xmlns="http://www.xquery-rdf.org">
    <!-- triple factory definition containing a single triple set constructor;
         defines triples describing the xs:schema elements found in a set
         of input documents (XSDs) -->
    <tripleSet subjectProvider="/xs:schema"
               subjectUri="$subjectUriConstructor()"
               type="x:schema">
        <predicates>
            <predicate uri="x:tns"
                       objectProvider="@targetNamespace"
                       objectValue="string(.)"/>
            <predicate uri="x:docURI"
                       objectProvider="root()"
                       objectValue="document-uri(.)"/>
            <predicate uri="x:imports"
                       objectProvider="xs:import"
                       objectUri="$objectUriConstructor()"/>
            <predicate uri="x:element"
                       objectProvider=".//xs:element"
                       objectURI="$objectUriConstructor()"/>
            <predicate uri="x:attribute"
                       objectProvider=".//xs:attribute"
                       objectURI="$objectUriConstructor()"/>
            <predicate uri="x:type"
```

```
                        objectProvider=".//(xs:simpleType, xs:complexType)"
                        objectURI="$objectUriConstructor()"/>
            <predicate uri="x:group"
                        objectProvider="xs:group"
                        objectURI="$objectUriConstructor()"/>
            <predicate uri="x:attributeGroup"
                        objectProvider="xs:attributeGroup"
                        objectURI="$objectUriConstructor()"/>
        </predicates>
    </tripleSet>
</tripleFactory>
```

A triple factory definition is essentially a collection of XQuery expressions playing particular roles in the production of triples. There are seven different roles, and details about each one are provided by Appendix A, *Triple set construction expressions*.

Integration issues

The exceptional ability of XQuery to express, filter and construct information is based on a peculiar processing model, which also implies serious limitations concerning what can be achieved using XQuery. The main limitations are

- The snapshot abstraction
- The inherent scaling problem

Issue: the snapshot model

The processing model of XQuery excludes changes of state. The processing model treats the sum total of accessible information – the system state - as a single snapshot. A query cannot, for example, first write a file and “later” read it – strictly speaking, there is no “before” or “after”, there is only a “now”. In principle, this would also preclude the possibility of first opening a data base connection, then using it, then closing the connection. In practice, however, there is a way around this limitation: if one expression E2 consumes as input a value supplied by another expression E1, then for all practical purposes E1 is evaluated before E2. This can be used deliberately by the “token pattern”:

```
let $conn := foo:openConnection(…)
let $result := foo:search($conn, …)
```

In this example, `foo:search` is evaluated after `foo:openConnection`, not because its `let` clause follows the other `let` clause – but because `foo:search` cannot be evaluated without the value of `foo:openConnection` – provided the code of `foo:search` really uses the input parameter.

A complex task of data integration will often require distinct phases, each one consisting of many operations, where one phase can only be executed when another phase is completely finished. In such scenarios, the ability to control sequence will quickly reach its limits, and the task cannot be solved by a *single* XQuery program. But if there are several queries, they cannot be bound together by yet another query, because this would effectively mean a single query again. Therefore it is not possible to implement multiple phases of state change using XQuery alone – some non-XQuery program must be in charge of calling the XQuery programs and orchestrating them. This limitation will often apply to typical scenarios of data integration.

Issue: scaling problems

The amazing capability of navigating trees and forests of information presupposes that those trees have been constructed *completely* and *before* the navigation starts. As the construction of trees is costly in

terms of memory and processing time, this can easily lead to fierce scaling problems. One solution consists in the use of XML data bases, which contain persistent representations of node trees, enabling navigation without parsing at navigation time. Another approach is the use of p-collections as described in (7). But it must be kept in mind that especially the capability of bulk navigation – searching items in a large number of trees - may in some situations be only a theoretical possibility.

XQuery as a data integration language - conclusions

Our view of the XQuery language is best captured by an image – the image of a person, its natural gifts, its education and its entry into professional life. Using this image, XQuery was born with very particular native gifts: to express information, find it and recombine it into new information. The gifts are amazing, but not very useful unless education broadens the view of the world and the scope of subject matter to which the gifts can be applied. Considering XQuery, the education includes above all the introduction to other, non-XML media types and to other resource access protocols than file and plain HTTP GET. As of version 3.1, XQuery has acquired a level of education that enables us to apply those natural gifts at a different scale than in the beginning. If the exceptional ability to express, find and recombine information can be evenly applied in a heterogeneous datascape, at an expanding scale – to which more media types and more access protocols are being added – the XQuery language becomes increasingly appropriate for *integrating* information. Version 3.1 has lead the language to a point where it seems legitimate to ask whether data integration might become a key field of using the language, if not its true profession.

A. Triple set construction expressions

(the section called “XQuery as a triple factory”) defines a generic model for the transformation of XML documents into triple sets. The model introduces seven kinds of XQuery expressions used as building blocks of the specification of a transformation. This appendix provides additional details about the expression kinds.

Table A.1.

subject-provider-expression

Property	Value
owned by	a triple set constructor
trigger	evaluated once for each document in the set of input documents
value	a sequence of nodes which is added to the union of all subject-backing nodes
context item	an input document
context variables	$documents: the set of all input documents

Table A.2.

subject-grouper-expression

Property	Value
owned by	a triple set constructor
trigger	evaluated once
value	used as a group key when grouping the concatenated values of {subject-provider-

Property	Value
	expression} into sets of subject-backing nodes (one set per subject)
context item	-
context variables	$documents: the set of all input documents
	$instance: a single node from the concatenated values of {subject-provider-expression}

Table A.3.

subject-uri-expression

Property	Value
owned by	a triple set constructor
trigger	evaluated once for each subject resource
value	the URI of this subject
context item	$sbn[1] (the first subject-backing node of this subject resource)
context variables	$documents: the set of all input documents
	$sbn: subject-backing nodes of this subject resource

Table A.4.

object-provider-expression

Property	Value
owned by	a predicate descriptor
trigger	evaluated once for each subject resource
value	a sequence of nodes which is added to the union of all object-backing nodes for this subject resource and this predicate
context item	$sbn[1] (the first subject-backing node of this subject resource)
context variables	$documents: the set of all input documents
	$sbn: subject-backing nodes of this subject resource

Table A.5.

object-grouper-expression

Property	Value
owned by	a predicate descriptor
trigger	evaluated once for each subject resource
value	used as a group key when grouping the concatenated values of {object-provider-expression} into sets of object-backing nodes (one set per object)
context item	-

Property	Value
context variables	$documents: the set of all input documents $sbn: subject-backing nodes of this subject resource $instance: a single node from the concatenated values of {object-provider-expression}

Table A.6.

object-uri-expression

Property	Value
owned by	a predicate descriptor
trigger	evaluated once for each subject resource (if predicate URI-valued)
value	the URI of the object backed by $obn
context item	$obn[1]: (the first object-backing node of this subject resource and for this predicate)
context variables	$documents: the set of all input documents $obn: the object-backing nodes of this subject resource and for this predicate

Table A.7.

object-value-expression

Property	Value
owned by	a predicate descriptor
trigger	evaluated once for each subject resource (if predicate literal-valued)
value	the value of the object backed by $obn
context item	$obn[1]: (the first object-backing node of this subject resource and for this predicate)
context variables	$documents: the set of all input documents $obn: the object-backing nodes of this subject resource and for this predicate

Bibliography

[1] BaseX - an open source XML database. Homepage. http://basex.org

[2] eXistdb - an open source XML database. Homepage. http://exist-db.org/exist/apps/homepage/index.html

[3] EXPath Community Group. Homepage. https://www.w3.org/community/expath/

[4] Gruen, Christian et al, eds. File Module 1.0 ExPath Module 20 February 2015. http://expath.org/spec/file/1.0

[5] Bray, Tim, ed. The JavaScript Object Notation (JSON) Data Interchange Format. Request for Comments: 7159. http://http://www.rfc-editor.org/rfc/rfc7159.txt

[6] Apache JMeter. Homepage. http://jmeter.apache.org/

[7] Rennau, Hans-Juergen. Node search preceding node construction - XQuery inviting non-XML technologies. XML Prague 2015, Conference Proceedings, p. 87-106. http://archive.xmlprague.cz/2015/files/xmlprague-2015-proceedings.pdf.

[8] Retter, Adam. RESTful XQuery. XML Prague 2012, Conference Proceedings, p. 91-124. http://archive.xmlprague.cz/2012/files/xmlprague-2012-proceedings.pdf.

[9] Saxonica - XSLT and XQuery processing. Homepage. www.saxonica.com

[10] Beckett, David et al, eds. RDF 1.1 Turtle. Terse RDF Triple Language. W3C Recommendation 25 February 2014. http://www.w3.org/TR/turtle/

[11] Gandon, Fabien and Guus Schreiber, eds. RDF 1.1 XML Syntax. W3C Recommendation 25 February 2014. http://www.w3.org/TR/rdf-syntax-grammar/

[12] Harris, Steve and Andy Seaborne, eds. SPARQL 1.1 Query Language. W3C Recommendation 21 March 2013. http://www.w3.org/TR/sparql11-query/

[13] Kay, Michael, ed. XPath and XQuery Functions and Operators 3.1. W3C Candidate Recommendation 18 December 2014. http://www.w3.org/TR/xpath-functions-31/

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

[15] Robie, Jonathan, Michael Dyck, eds. XQuery 3.1: An XML Query Language. W3C Candidate Recommendation 18 December 2014. http://www.w3.org/TR/xquery-31/

[16] Walsh, Norman et al, eds. XQuery and XPath Data Model 3.0. W3C Recommendation 8. April 2014 http://www.w3.org/TR/xpath-datamodel-30/

[17] Snelson, John and Jim Melton, eds. XQuery Update Facility 3.0. W3C Last Call Working Draft 19 February 2015. http://www.w3.org/TR/xquery-update-30/

[18] Thompson, Henry S. et al, eds. XML Schema Part 1: Structures Second Edition. W3C Recommendation 28 October 2004. http://www.w3.org/TR/xmlschema-1/

[19] Kay, Michael, ed. XSL Transformations (XSLT) Version 3.0. W3C Last Call Working Draft 2 October 2014. http://www.w3.org/TR/xslt-30/