
Nexus Query Language

Nexus is a Resource Description Framework (RDF) [1,2] implementation designed for merging and transforming models. Models are input and output from Nexus as XML. A subset of the RDF syntax is supported, however non-RDF, XML documents can be mined for statements and a wide range of XML documents can be extracted.

Information is extracted from the Nexus as a hierarchy of elements, each with a set of attributes. In general, many such hierarchies can be imposed on the Nexus contents because the Nexus is organized as a directed, labeled graph.

The query mechanism is responsible for extracting hierarchical views from the graph. (Note that the hierarchical form of the results makes the query mechanism different from that of a relational database, which produces tabular results.)

Queries are expressed as transformations of the model. The transformation capabilities are similar to a subset of Extensible Stylesheet Language for Transformations (XSLT) [3]. Once obtained, query results may be subjected to larger scale restructuring by standard transformation engines, as required.

Information Model

For the purpose of query execution, the model is viewed as *nodes*. The term *node* refers to something that can be matched by a template. (That is, the XSLT concept of a node and not the RDF concept.)

A node consists of the property and value of a single RDF statement. Template matching criteria are defined in terms of the identity of the property and the identity or type of its value (if the value is a resource).

Each node has a set of *related nodes* obtained as follows. Consider the properties of the value of a node. The combination of each of these properties with each of its values defines a related node. (Note that a property can be many-valued.)

Query execution takes place in context of a network of nodes:

- The *root node* is made up of the special property `query:root` and an anonymous resource as its value. Template processing begins with the root node as the current node. (This node is analogous to the root node in the XML information set.) The root node has one related node, other than its self node (see below): the seed node.
- The *seed node* is made up of the special `query:seed` property and a seed resource explicitly designated when the query is executed. A template that matches the seed property is required in every (non-degenerate) query. (The seed node is analogous to the document node in the XML information set.)

Note that there are several ways the seed resource could be processed by the initial template. Two of the possibilities are:

- If the initial template calls the `query:insert` and `query:format` actions (see below), then the seed resource generates a singleton first-level element and its attributes, below the document element.
- If the seed template calls the `query:traverse` function then other resources, reachable from the seed resource by further template matching, will become the

first-level elements in the output.

- The remaining nodes consist of the related nodes of the seed node (if any) and their related nodes, recursively. These are candidates for further template matching, assuming the initial template triggers this.

Nodes are of two kinds: those whose value is a resource and those whose value is an atomic type such as integer or string. The latter nodes have no related nodes (except the self node, see below). These two kinds of node are analogous to element and attribute nodes respectively in the XML information set.

In addition to the nodes described above, every node has a related node call the self node. A node and its related self node have the same value. The self node is made up of the special property `query:self` and this value.

Processing Model

A query consists of templates and actions. The input to the query execution function consists of the URI of the query, and that of a seed resource. The system of nodes described above is then (conceptually) constructed and a round of template matching is performed.

Template processing takes place in the context of a current node, its parent and ancestors, a current mode, and a current output element. The context of the initial round consists of the root node, the default mode, and the document element.

Each related node of the current node is examined to determine if it matches any of the templates belonging to the current mode. The related node may match at most one template. The highest priority template will be matched if the other criteria qualify more than one template.

Each related node that matches a template is selected for processing. The related node becomes the current node and the former current node is referred to as its *parent node*. The actions associated with the matching template are executed in the context of the new current node. The parent node is then (conceptually) restored as the current node and the next selected node is processed.

The template actions may:

- Generate one or more new elements, with sub-elements, in the output.
- Generate one or more attributes for the current element or the new elements.
- Trigger a new round of template matching in the context of the now-current node.
- Select a new current mode for the new round of template matching.

Element generation occurs in the context of the current element. The current element is the last element generated which is not yet closed. A new element is nested in the current element and then becomes the current element. When the new element is closed, the former current element is restored.

All elements generated by a template are closed before processing of that template is complete.

Recursion

In general, the nexus content will contain loops. (That is, a resource refers to itself, possibly via intervening resources.) Therefore, the following rule is added to ensure that template expansion will terminate:

A candidate node cannot match a given template if the current node or one of its ancestors matched the same template with the same property and value.

Query Structure

The design of queries resembles a subset of XSLT. A query consists of a plan element that contains a set of template elements. Each template element may contain a hierarchy of action elements.

A query can be created directly in XML, or by translation from a query language or a style sheet. Once created, the query is stored in the Nexus as a resource.

Plan Element

The document element of a query is expressed in XML as follows:

```
<query:plan rdf:about="qname-or-URI" result="qname-or-URI">
  <!-- query:template elements go here -->
</query:plan>
```

The `rdf:about` attribute identifies this query so that it may be subsequently referenced in queries. If this attribute is omitted, the query is assigned an arbitrary name within a namespace reserved for nexus internal resources.

The `result` attribute controls the document element produced when this query is executed. Queries are processed as if there were a built-in template for generating the document element. The document element type is given by the `result` attribute. If `result` is omitted it defaults to the value of `rdf:about`.

The `plan` element may also provide namespace prefix declarations (via `xmlns` attributes). When the query is executed, the resulting document will carry the same declarations except those that declare the query plan namespace.

The `plan` element may contain any number of `template` elements, described next.

Template Element

The template element is expressed in XML as follows:

```
<query:template
  match="qname-or-URI-list"
  type="qname-or-URI"
  id="qname-or-URI-list"
  priority="number"
  mode="qname-or-URI">
  <!-- query:call elements go here -->
</query:template>
```

The purpose of a template is to specify a set of criteria to be applied to a node (see section 4.1). The template associates those criteria with a set of actions to be executed in the context of a matching node.

- The `match` attribute gives the primary criterion. This attribute is required. The `match` attribute gives a list of qualified names or URI's. In the simplest case a single `qname-or-URI` is given. This is compared to the property of candidate node. The template match fails if they are not the same resource. Additional `qname-or-URI` values may be prepended. Reading right to left, these are compared to the property of the parent node, its parent's property, and so on. The template match fails if any of the resources compared are not identical.
- The `type` attribute specifies the `qname-or-URI` of a class in the nexus RDF schema. This attribute is optional. If present, the specified type is compared to the type of the value of the candidate node. (The

type of a resource is defined by its `rdf:type` property.) The template match fails if the value is not of the specified type or a derived type.

- The `id` attribute specifies the *qname-or-URI* for each of a list of resources. This attribute is optional. If present, and if the value of the candidate node is not in this list, then the template match fails.
- The `priority` attribute specifies a floating-point number for the template priority. This attribute is optional. The default priority for all templates is 1.0. If several templates match based on other criteria, then the template with the highest numerical priority matches and the others do not. (It is undefined which template matches when two or more have the same priority.)
- The `mode` attribute specifies the *qname-or-URI* of a template-matching mode. The template match fails if the specified mode is not the current mode. This attribute is optional. If omitted, the special mode, `query:default_mode`, is implied. This is also the initial template-matching mode.

Actions

The actions taken when a template is matched are given by its subordinate elements. Each action element has the following XML syntax:

```
<query:call name="qname-or-URI" action-specific-attributes>
  <!-- query:call elements for nested actions go here -->
</query:call>
```

The **name** attribute is required and identifies the action to be performed. The actions are executed in the order they appear. Nested actions are executed in the context established by their parents. Top-level actions are executed in the context established by the template.

The actions are:

- `query:insert` generates a new element in the output. The element type is specified by the `rename` attribute. If this is omitted an arbitrary element type is generated. Additional actions may generate attributes and sub-elements of the new element. These actions should be nested within the insert action, as shown above.
- `query:traverse` triggers a new round of template matching in the present context. The `shift` attribute specifies a template matching mode to use. If omitted, the default mode is used. Each related node of the current node is compared to the set of templates. Matches may generate attributes and sub-elements of the current element.
- `query:format` generates a new attribute for the current element. The name of the generated attribute is specified by the `rename` attribute. If this is omitted an arbitrary name is generated. The value is specified by the `literal` and/or `value` attributes as described in the next section. If these are both omitted, the value of the current node is used.

Generating Text

The `format` action generates attribute value text based on the current node and the `literal` and `value` attributes of the action, if present.

- If the `literal` attribute is present, its value is used. (But see below for formatting rules).
- Otherwise, if the `value` attribute is present, it is evaluated in the context of the current node and the result is used. (Evaluation rules are given below.)

- Otherwise, if neither attribute is present, the value of the current node is used.

Formatting Rules

The literal attribute supplies a text string, optionally containing C sprintf-style format directives. Any format directives are expanded before the string is used as the generated attribute value. A format directive has the following form:

`%(expression)s`

The flag, *s*, may be one of `%`, `c`, `s`, `i`, `d`, `u`, `o`, `x`, `X`, `e`, `E`, `f`, `g`, `G`. It may be prefixed with a width and precision (per sprintf).

The expression in parenthesis should evaluate to a string, integer, or floating value depending on which format directive is used.

Expression primaries are:

Expression Primary	Result
<code>self()</code>	The value of the current node. (A simple type or a resource.)
<code>parent()</code>	The value of the current node's parent. (A resource)
<code>value()</code>	The result of evaluating the value attribute of the action. (A simple type or a resource.)

These primaries may be converted via the following functions:

Conversion Function	Result Type
<code>int(p)</code>	Integer (suitable for <code>%d</code> formats).
<code>float(p)</code>	Floating (suitable for <code>%f</code> formats).
<code>str(p)</code>	String, (suitable for <code>%s</code> formats). This conversion is implicit. For resources, the URI string is returned.
<code>qname(p)</code>	A qualified name in the context of the current element. The argument should be a resource.
<code>lname(p)</code>	A local name (the qualified name without namespace prefix). The argument should be a resource.

Note that the python interpreter is used to evaluate these expressions. Thus, the full power of python expression syntax and its built-in functions are also available, at least on an experimental basis.

Value Attribute

The value attribute is used to gain access to the value of nodes other than the current node and its parent. The value attribute takes a sequence of qualified names or URI's. Each of these should be a property, or the special operator, `query:parent`.

The sequence is evaluated as a path beginning at the current node and navigating to an adjacent node with each step. Only single-valued properties should be used.

Each property step yields a node made up of the property and its value, evaluated in the current context. Each parent step yields the parent node of the current node.

In evaluating this path, it is possible to reach nodes that have not or will not be matched by templates.

References

- [1] “Resource Description Framework (RDF) Model and Syntax Specification”, W3C Recommendation, February 1999, <http://www.w3.org/TR/REC-rdf-syntax/>
- [2] “Resource Description Framework (RDF) Schema Specification”, W3C Candidate Recommendation, March 2000, <http://www.w3.org/TR/rdf-schema/>
- [3] “XSL Transformations (XSLT)Version 1.0”, W3C Recommendation November 1999, <http://www.w3.org/TR/xslt/>