# GNQuery

## Introduction

GNQuery is a new way to write queries in GN4, Tark4 and GNPortal systems. It is based on the XPath and XQuery standards, it can express complex queries in a very concise way and supports a broad variety of expressions, relationships and attribute types.

GNQuery is a replacement for the existing ‘search conditions’ queries written as custom XML files. In versions 1.5 and 1.6 the implementation is partial, and works only for some commands (see below for details). In 2.0 the implementation is almost full, i.e. all the searches go through GNQuery, but are not available in the configuration files. Since the version 2.1, all GNQuery features are available also on the configuration level.

The search conditions queries are still available since version 1.5 and in all newer versions for the backward compatibility.The following documentation assumes that the reader is familiar with XPath (version 1.0) and with the standard XML representation used by GN4, Tark4 and GNPortal systems.

## Overview

All the data contained in a GN4, Tark4 or GNPortal system is represented in a standard way as XML, GNQuery uses this XML representation to write queries as XPath and XQuery expressions. In other words all the data contained in a system is seen as a virtual XML ‘file’:

<gn4:story id="obj33478" name="635e2bcd-7803-4a7d-b681-a7d6f865f417" . . .>

<gn4:title>From mobile 2</gn4:title>

. . .

</gn4:story>

<gn4:story id="obj33476" name="0941b3a9-87ab-4c18-8c17-71270d6600ea" . . .>

<gn4:title>From mobile 1</gn4:title>

. . .

</gn4:story>

. . . (all other stories) . . .

<gn4:image id="obj33477" name="image\_47a560fd-1d45-4bde-861b-1a2a816c0f99" . . . >

<title>articleVersions</title>

. . .

</gn4:image>

. . . (all other images) . . .

. . . (all other objects of different types) . . .

and queries are XPath expressions executed on this XML – e.g. the expression

gn4:story

returns ALL stories in the system – being an XPath that matches all the story in the above XML. To find a story with a specific title:

gn4:story[gn4:title='My title']

that is an XPath matching all the gn4:story elements with a gn4:title sub-element containing the text 'My title'.

Query expressions specify objects attributes using their standard XML representation, so in the example above the story title is ‘gn4:title’ – because it is represented as a XML element; to get all the stories modified since August 1st use:

gn4:story[@modifiedDate>='2012-08-01T00:00:00Z']

because the last modification date is represented as an XML attribute:

<gn4:story id="obj33476" modifiedDate="2012-14-08T17:33:07Z" . . .>

. . .

</gn4:story>

Use the same logic to specify attributes with complex values, for example keywords are represented in XML like this:

<gn4:story id="obj33476". . .>

. . .

<gn4:keywords>

<gn4:item>architecture</gn4:item>

<gn4:item>bridges</gn4:item>

</gn4:keywords>

. . .

</gn4:story>

so to find all stories with a specific keyword use:

gn4:story[gn4:keywords/gn4:item='bridges']

To find the stories with ANY keyword use:

gn4:story[gn4:keywords/gn4:item]

i.e. all stories that have at least a ‘gn4:item’ sub-sub-element under their ‘gn4:keywords’ sub-element.

Note that the ‘gn4:keywords’ sub-element is always present – even if the story does not have any keyword, so

gn4:story[gn4:keywords]

is the same as

gn4:story

because the condition inside the brackets is always true.

### Testing queries

All srv4, cmd4 and do.ashx commands that accept search conditions can use GNQuery expressions. For example, using the ‘srv4 export’ command, pass the query expression as the ‘-conditions ’ option – so:

srv4 export -conditions "gn4:story[gn4:title='From mobile1']" -out c:\temp\q.xml

exports all the stories with title 'From mobile 1' in the ‘c:\temp\q.xml’ file.

Srv4, cmd4 and do.ashx detects that the condition is a GNQuery expression and not an XML conditions file checking the beginning of the string: if it starts with a query namespace prefix (‘gn4:’, ‘nav:’) it is assumed to be a GNQuery expression.

Specifically, the commands accepting GNQuery expressions are:

* srv4 export
* srv4 index
* srv4 unindex
* srv4 spike
* cmd4 index
* cmd4 spike
* cmd4 unpublish
* cmd4 recompute
* cmd4 export
* cmd4 get
* cmd4 triggerwf
* do.ashx?cmd=search
* do.ashx?cmd=unpublish
* do.ashx?cmd=spike
* do.ashx?cmd=triggerwf

### Namespaces and prefixes

The virtual XML uses the same namespaces of the standard ‘navigation’ XML (e.g. the XML generated by the do.ashx?cmd=feed command), associating them with standard prefixes:

|  |  |
| --- | --- |
| **Prefix** | **Namespace** |
| gn4 | system schema namespace - typically ‘urn:schemas-teradp-com:gn4tera’ |
| nav | navigation extensions namespace ‘http://www.teradp.com/schemas/GN4/1/ObjNav.xsd’ |
| fn | extension functions namespace ‘http://www.teradp.com/schemas/GN4/1/Xslt’ |
| lc | login context namespace ‘http://www.teradp.com/schemas/GN4/1/LoginContext.xsd’ |
| oc | operation context namespace ‘http://www.teradp.com/schemas/GN4/1/OperationContext.xsd’ |

These prefixes can (and should) be used directly in the GN4 query expressions.

### XML root

The virtual XML contains all searchable items (objects in the system etc.) at the root level - this means that is not a valid standard XML because it has multiple root elements, but that does not matter for GNQuery and makes the query expressions simpler.

The GNQuery expressions are evaluated assuming that the current node is the root of the virtual XML – this means that the expressions

gn4:story

and

/gn4:story

are equivalent – they both match all ‘gn4:story’ elements under the root, i.e. all stories in the system.

### Object types hierarchy

Each object in a system belongs to its specific object type, but it can be also considered belonging to the ancestors of its object type. For example the object type ‘story’ derives from the object type ‘package’ that in turn derives from ‘archiveObject’ and so on; this mean that a story object can be also considered a package or an archiveObject (and so on).

GNQuery accommodates this placing the same object multiple times in the virtual XML – once for each of its possible object types:

. . .

<gn4:story id="obj33478" nav:objectType="story" . . .>

<gn4:title>From mobile 2</gn4:title>

. . .

</gn4:story>

<gn4:package id="obj33478" nav:objectType="story" . . .>

<gn4:title>From mobile 2</gn4:title>

. . .

</gn4:package>

<gn4:archiveObject id="obj33478" nav:objectType="story" . . .>

<gn4:title>From mobile 2</gn4:title>

. . .

</gn4:archiveObject>

. . .

in this way the query expression

gn4:archiveObject

returns ALL possible archive objects in the system – including all the stories, as well as packages, images and all other object of types derived (directly or indirectly) from ‘archiveObject’.

Use the XML attribute ‘nav:objectType’ to test for the real type of an object, for example the query expression:

gn4:archiveObject[@nav:objectType='story']

returns all the archive objects that are actually stories – i.e. it is the same thing as:

gn4:story

(This can be useful when writing code that generates query expressions dynamically appending conditions to a common root condition).

### Paths

Query expressions can use all the standard XPath node selectors:

* element names – e.g. ‘gn4:story’, ‘nav:referencing’;
* attribute names – e.g. ‘@name’, ‘@nav:mimeExt’;
* all nodes – ‘node()’;
* text nodes – ‘text()’;
* root node – ‘/’;
* current node– ‘.’;
* the ‘\*’ wild-card – either for the entire name (e.g. ‘gn4:story/\*’ – all sub-elements of ‘gn4:story’) or just the local name (e.g. ‘gn4:story/nav:\*’ – all sub-elements of ‘gn4:story’ with a name in the ‘nav’ namespace).

In particular the ‘\*’ wild-card can be used to create queries across multiple object types, for example

gn4:\*[@pixWidth>0]

returns all objects that have a ‘pixWidth’ attribute greater than 0; when using the standard Tark4 schema these will be images and videos with a pixel width greater than 0.

As already seen node selectors can be concatenated with ‘/’ or restricted with a condition within brackets. They can also be combined with the union operator ‘|’ – e.g.

(gn4:image|gn4:img)[@pixHeight>1000]

returns all the images and editorial images (‘img’) with a pixel height greater than 1000.

The ‘//’ XPath operator and the XPath axes ‘ancestor’, ‘ancestor-or-self’, ‘descendant’, ‘descendant-or-self’, ‘following’, ‘following-sibling’, ‘namespace’, ‘preceding’ and ‘preceding-sibling’ are NOT supported.

### Operators

GNQuery expression can use all standard XPath 1.0 operators (see <http://www.w3schools.com/xpath/xpath_operators.asp>), so for example

gn4:image[@pixWidth\*@pixHeight > 10000]

returns all images with an area greater than 10,000 square pixels, whereas

gn4:image[@pixWidth > 0 and @pixWidth div @pixHeight > 1]

returns all images with an aspect ratio (height divided by width) greater than 1.

As per XPath standard the divisions are always non-integer – i.e. they return fractional (floating-point) values even when dividing integer quantities. For integer divisions there is an operator ‘idiv’ (as in XPath 2.0), so

gn4:image[@pixWidth > 0 and @pixWidth idiv @pixHeight > 1]

returns all images with an aspect ratio of at least 2 – because idiv returns 1 for any aspect ratio less than that.

As in standard XPaths the evaluation of relational operators (‘=’, ‘!=’, ‘<’, ‘>’, ‘>=’ and ‘<=’) uses special rules when using node-sets, e.g.

gn4:keywords/gn4:item

is a node-set containing all the ‘gn4:item’ sub-elements of the ‘gn4:keywords’ element, the expression

gn4:keywords/gn4:item='bridges'

is true if ANY of the nodes in the node-set matches the condition – so if there is at least one keyword ‘bridges’; similarly the comparison of two node sets like

gn4:objs/gn4:ref/gn4:title=gn4:objs/gn4:ref/gn4:caption

is true if any of the nodes in the first set matches any of the nodes in the second set – i.e. if there is any of the titles equal to any of the captions.

The arithmetic operators on the other hand simply use the first value in any node set. The logical operators (‘and’ and ‘or’) convert the node-sets to booleans first (non-empty = true, empty = false) and then combine the results.

### Functions

GNQuery implements some- but not all – of the standard XPath functions, and quite an extensive list of non-standard ‘extension’ function useful in queries.

Note that when functions are applied to object attributes GNQuery does not compute them directly, but converts them to SQL expressions that are then computed by the underlying SQL server. This means that the precise semantics of the functions can be different from what is specified by the XPath standard, for example according to the XPath standard

substring(gn4:title, 1.5, 2.6)

should return three characters of the title – from the second to fourth, but if you execute it using a Microsoft SQL Server based system the result is actually the first and second character of the title.

#### Standard XPath functions

GNQuery implements the following standard XPath functions:

**boolean(*expr*)**

Converts *expr* to a Boolean value.

true()

Returns the ‘true’ Boolean value.

false()

Returns the ‘false’ Boolean value.

not(*expr*)

Converts *expr* to a Boolean value and negates it.

number(expr)

Converts *expr* to a number. If it is not a valid number returns the special ‘NaN’ (not a number) value.

ceiling(*expr*)

Converts *expr* to a number and computes its ceiling – i.e. the smallest integer value that is greater or equal the expression.

floor(*expr*)

Converts *expr* to a number and computes its floor – i.e. the greatest integer value that is smaller or equal the expression.

sum(*expr*)

Converts each node in the *expr* node-set to a number and adds them all.

min(*expr*)

Returns the minimum value of the *expr* node-set.

max(*expr*)

Returns the maximum value of the *expr* node-set.

avg(*expr*)

Converts each node in the *expr* node-set to a number and returns the average value.

round(*expr*)

Converts *expr* to a number and rounds it to the nearest integer value.

concat(*expr1*, *expr2*, . . .)

Converts all the expression to strings and concatenates them.

contains(*expr1*, *expr2*)

Returns true if *expr2* is contained as a sub-string in *expr1.*

In standard XPath *expr1* and *expr2* are first converted into strings and then the test takes place; GNQuery on the other hand uses the same logic of the relational operators: if one of the expression is a node-set executes the test for each node, and returns true if any of the test is satisfied. This means that in GNQuery the expression

contains(gn4:keywords/gn4:item,'bridges')

is true if any of the keywords contains ‘bridges’, whereas in standard XPath is true only if the first keyword contains ‘bridges’.

starts-with(*expr1*, *expr2*)

Returns true if *expr1* (as a string) starts with *expr2.*

GNQuery uses the same non-standard evaluation logic as for the ‘contains()’ function above.

string(*expr*)

Converts *expr* to a string.

string-length(expr)

Converts *expr* to a string and returns its length.

substring(*expr1*, *expr2*)

Converts *expr1* to a string, *expr2* to a number and returns all the characters in the *expr1* string starting at the *expr2* position. Positions start from 1.

substring(*exp1*, *expr2*, *expr3*)

Converts *expr1* to a string, *expr2* and *expr3* to numbers and returns *expr3* characters in the *expr1* string starting at the *expr2* position. Positions start from 1.

count(*expr*)

Returns the number of nodes in the node-set *expr*. For example the query

gn4:story[count(gn4:keywords/gn4:item)>=2]

returns all the stories with at least two keywords.

position()

Returns the position, or index number, of the current node relative to all the matching nodes in the node list. It is used to limit the number of results returned by a query, for example

gn4:story[position()<=5] order by @modifiedDate descending

returns the five most recently modified stories: search all the stories, order them by modification date with the most recent first, get only the first five. Another example:

gn4:story[contains(gn4:title,’test’)][5<=position() and position()<=10]

returns from all the stories whose title contains ‘test’ those from position 5 to position 10 (both included). When – as in this case – there is no specified ordering the system uses ordering by object id.

The ‘position()’ function can be used only in comparisons with numeric constants; these comparison cannot use the ‘!=’ (not equal to) operator; these comparisons can be combined only using ‘and’; comparisons using ‘position()’ cannot be mixed with other conditions. These limitations mean that all the following queries are illegal

gn4:story[position()<$p0] -- Comparison with a variable instead than a constant

gn4:story[position()<@nWords] -- Comparison with a path instead than a constant

gn4:story[position()<10 and contains(gn4:title,’test’)] -- Mixed conditions

gn4:story[position()<10 or position()>20] -- Comparisons not combined with ‘and’

gn4:story[position()!=10] -- Comparison using the ‘!=’ operator

#### Extension functions

GNQuery implements also the following extension functions (in the namespace ‘http://www.teradp.com/schemas/GN4/1/Xslt’ with standard prefix ‘fn’):

fn:objectIdFromString(*expr*)

Converts an object id string into the corresponding numeric id – e.g. converts the string ‘obj1234’ in the number ‘1234’. Returns ‘NaN’ if the string cannot be converted.

fn:dataIdFromString(*expr*)

Converts a data id string into the corresponding numeric id – e.g. converts the string ‘data678’ in the number ‘678’. Returns ‘NaN’ if the string cannot be converted.

fn:in(*expr*, *expr1*, *expr2*, . . . )

Returns true if *expr* is equal to any of the values *expr1*, *expr2* etc. The type of *expr* controls the type of comparison used: if *expr* evaluates to a number the values are converted to numbers, if *expr* evaluates to a string the values are converted to strings.

If *expr* is a node-set the function returns true if any of the nodes is equal to any of the expression, so that the query

gn4:story[fn:in(gn4:keywords/gn4:item,'bridges','tunnels')]

returns all the stories that have at least a keyword equal to either ‘bridges’ or ‘tunnels’.

fn:ends-with(*expr1*, *expr2*)

Returns true if *expr1* (as a string) ends with *expr2.*

GNQuery uses the same non-standard evaluation logic as for the ‘contains()’ function above.

fn:dateFromDateTime(*expr*)

Converts a date-time value to the corresponding date – stripping the time part.

fn:dateTimeFromString(*expr*)

Converts a string into the corresponding date-time. The string is expected to be in the standard format ‘YYYY-MM-DDThh:mm:ssZ’. If the string is not a valid date-time returns the minimum possible date-time.

fn:dateFromString(*expr*)

Converts a string into the corresponding date. The string is expected to be in the standard format ‘YYYY-MM-DDZ’. If the string is not a valid date returns the minimum possible date.

fn:dateTimeFromTimeString(*expr*)

Converts a time string into the corresponding date-time. The string is expected to be in the standard format ‘hh:mm:ssZ’. If the string is not a valid time returns the minimum possible date-time.

fn:inRelativeYears(*expr*, *fromexpr*, *toexpr*)

Checks that the date-time or date represented by the *expr* string is within *fromexpr* and *toexpr* years relative to the current one. *fromexpr* and *toexpr* must be numeric constants or variable references (‘$name’), a value of 0 represent the current year, positive values the years after the current one (+1 = next year etc.), negative values the years before the current one (-1 = previous year and so on). For example

gn4:story[fn:inRelativeYears(@modifiedDate,0,1)]

returns all the stories that have been modified from the beginning of the current year (‘0’) to the beginning of next year (‘1’).

The test includes the lower bound and excludes the upper one. Each day is considered as starting at the time specified by the ‘StartDayTime’ value in the GlobalConfig configuration – by default midnight. The years start date-times are in the local time of the client executing the query. This means that if today is August 20th, 2012, ‘StartDayTime’ in GlobalConfig is ‘6:00:00’ (6 AM) and the client time-zone is US Easter Standard (UTC-5), the example above is equivalent to

@modifiedDate >= ‘2012-01-01T11:00:00Z’ and @modifiedDate < ‘2013-01-01T11:00:00Z’

where the ‘11:00:00Z’ comes from converting 6 AM from US Eastern Time to UTC.

Use ‘infinite’ values to generate open-ended tests – i.e. all date-time before or after a certain relative date. For example

gn4:story[fn:inRelativeYears(@modifiedDate,-1 div 0,0)]

returns all the stories modified before the current year because ‘-1 div 0’ returns negative infinite, so the ‘from’ part of the condition is always true, and the actual test is just

@modifiedDate < ‘2012-01-01T11:00:00Z’

Similarly, to find all tasks with a due date after the end of this year use

gn4:task[fn:inRelativeYears(@dueDate,1,1 div 0)]

that is equivalent to a test

@dueDate >= ‘2013-01-01T11:00:00Z’

because ‘1 div 0’ returns positive infinite, so the ‘to’ part of the condition is always true.

If *expr* is a node-set the function returns true if any of the nodes matches the condition, so that the query

gn4:story[fn:inRelativeYears(gn4:objs/gn4:ref/nav:refObject/gn4:\*/@modifiedDate,0,1)]

returns all the stories that have at least a related objects that has been modified this year.

fn:inRelativeYearsZ(*expr*, *fromexpr*, *toexpr*)

A variant of the ‘inRelativeYears()’ function that uses UTC times – i.e. that does not convert from the client time-zone to UTC. For example

gn4:story[fn:inRelativeYearsZ(@modifiedDate,0,1)]

if today is August 20th, 2012 and ‘StartDayTime’ in GlobalConfig is ‘6:00:00’ (6 AM), generates this test:

@modifiedDate >= ‘2012-01-01T06:00:00Z’ and @modifiedDate < ‘2013-01-01T06:00:00Z’

regardless of the client time-zone

fn:inRelativeYears0(*expr*, *fromexpr*, *toexpr*)

A variant of the ‘inRelativeYears()’ function that does not uses the start day time – i.e. that consider always the days as beginning at midnight. For example

gn4:story[fn:inRelativeYears0(@modifiedDate,0,1)]

if today is August 20th, 2012 and the client time-zone is US Easter Standard (UTC-5), generates this test:

@modifiedDate >= ‘2012-01-01T05:00:00Z’ and @modifiedDate < ‘2013-01-01T05:00:00Z’

regardless of the value of ‘StartDayTime’ in GlobalConfig

fn:inRelativeYearsZ0(*expr*, *fromexpr*, *toexpr*)

A variant of the ‘inRelativeYears()’ function that uses UTC times and does not uses the start day time. For example

gn4:story[fn:inRelativeYearsZ0(@modifiedDate,0,1)]

if today is August 20th, 2012, generates this test:

@modifiedDate >= ‘2012-01-01T00:00:00Z’ and @modifiedDate < ‘2013-01-01T00:00:00Z’

regardless of the client time-zone and of the value of ‘StartDayTime’ in GlobalConfig

fn:inRelativeMonths (*expr*, *fromexpr*, *toexpr*)

Checks that the date-time or date represented by the *expr* string is within *fromexpr* and *toexpr* months relative to the current one. For example executing

gn4:story[fn:inRelativeMonths(@modifiedDate,-1,0)]

returns all stories modified from ‘-1’ = beginning of the previous month to ‘0’ = beginning of the current month – i.e. if today is August 20th, 2012 returns all stories modified in July 2012.

See the explanation of fn:inRelativeYears() above for details about the handling of time-zones, the start of the day hour and the use of infinite values to generate open-ended tests.

As for fn:inRelativeYears() if *expr* is a node-set the function returns true if any of the nodes matches the condition.

fn:inRelativeMonthsZ(*expr*, *fromexpr*, *toexpr*)

A variant of the ‘inRelativeMonths()’ function that uses UTC times. See explanation of the similar function ‘inRelativeYearsZ()’ above.

fn:inRelativeMonths0(*expr*, *fromexpr*, *toexpr*)

A variant of the ‘inRelativeMonths()’ function that does not uses the start day time. See explanation of the similar function ‘inRelativeYears0()’ above.

fn:inRelativeMonthsZ0(*expr*, *fromexpr*, *toexpr*)

A variant of the ‘inRelativeMonths()’ function that uses UTC times and does not uses the start day time. See explanation of the similar function ‘inRelativeYearsZ0()’ above.

fn:inRelativeWeeks (*expr*, *fromexpr*, *toexpr*)

Checks that the date-time or date represented by the *expr* string is within *fromexpr* and *toexpr* weeks relative to the current one. For example executing

gn4:story[fn:inRelativeWeeks(@modifiedDate,0,2)]

returns all stories modified from ‘0’ = beginning of this week to ‘2’ = beginning of the week after the next – i.e. if the first day of the week is Sunday and today is Monday August 20th, 2012 the query returns all stories modified on or after Sunday, August 19th 2012 and before Sunday, September 2nd 2012.

The first day of the week is the one specified in the locale settings of the client.

See the explanation of fn:inRelativeYears() above for details about the handling of time-zones, the start of the day hour and the use of infinite values to generate open-ended tests.

As for fn:inRelativeYears() if *expr* is a node-set the function returns true if any of the nodes matches the condition.

fn:inRelativeWeeksZ(*expr*, *fromexpr*, *toexpr*)

A variant of the ‘inRelativeWeeks()’ function that uses UTC times. See explanation of the similar function ‘inRelativeYearsZ()’ above.

fn:inRelativeWeeks0(*expr*, *fromexpr*, *toexpr*)

A variant of the ‘inRelativeWeeks()’ function that does not uses the start day time. See explanation of the similar function ‘inRelativeYears0()’ above.

fn:inRelativeWeeksZ0(*expr*, *fromexpr*, *toexpr*)

A variant of the ‘inRelativeWeeks()’ function that uses UTC times and does not uses the start day time. See explanation of the similar function ‘inRelativeYearsZ0()’ above.

fn:inRelativeDays (*expr*, *fromexpr*, *toexpr*)

Checks that the date-time or date represented by the *expr* string is within *fromexpr* and *toexpr* days relative to the current one. For example executing

gn4:story[fn:inRelativeDays(@modifiedDate,-2,-1)]

returns all stories modified from ‘-2’ = beginning of two days ago to ‘-1’ = beginning of yesterday – i.e. if today is August 20th, 2012 the query returns all stories modified on August 18th 2012.

See the explanation of fn:inRelativeYears() above for details about the handling of time-zones, the start of the day hour and the use of infinite values to generate open-ended tests.

As for fn:inRelativeYears() if *expr* is a node-set the function returns true if any of the nodes matches the condition.

fn:inRelativeDaysZ(*expr*, *fromexpr*, *toexpr*)

A variant of the ‘inRelativeDays()’ function that uses UTC times. See explanation of the similar function ‘inRelativeYearsZ()’ above.

fn:inRelativeDays0(*expr*, *fromexpr*, *toexpr*)

A variant of the ‘inRelativeDays()’ function that does not uses the start day time. See explanation of the similar function ‘inRelativeYears0()’ above.

fn:inRelativeDaysZ0(*expr*, *fromexpr*, *toexpr*)

A variant of the ‘inRelativeDays()’ function that uses UTC times and does not uses the start day time. See explanation of the similar function ‘inRelativeYearsZ0()’ above.

fn:inRelativeHours (*expr*, *fromexpr*, *toexpr*)

Checks that the date-time represented by the *expr* string is within *fromexpr* and *toexpr* hours relative to the current one. For example executing

gn4:story[fn:inRelativeHours(@modifiedDate,0,6)]

returns all stories modified from ‘0’ = beginning of the current hour to ‘6’ = beginning of the sixth hour from now – i.e. if today is August 20th, 2012 at 5:40 PM local time the query returns all stories modified on August 20th 2012 between 5:00 PM (included) and 11 PM (excluded) local time.

See the explanation of fn:inRelativeYears() above for details about the use of infinite values to generate open-ended tests.

As for fn:inRelativeYears() if *expr* is a node-set the function returns true if any of the nodes matches the condition.

When using negative values keep in mind that 0 is **not** the current time, but the beginning of the current hour. If you want to get all the stories modified in the last hour you should use a condition like:

gn4:story[fn:inRelativeHours(@modifiedDate,-1,1)]

fn:inRelativeHoursZ (*expr*, *fromexpr*, *toexpr*)

A variant of the ‘fn:inRelativeHours()’ function that uses UTC times. For example executing

gn4:story[fn:inRelativeHoursZ(@modifiedDate,0,6)]

if today is August 20th, 2012 at 3:40 PM UTC the query returns all stories modified on August 20th 2012 between 3:00 PM (included) and 9 PM (excluded) UTC.

fn:inRelativeMinutes(*expr*, *fromexpr*, *toexpr*)

Checks that the date-time represented by the *expr* string is within *fromexpr* and *toexpr* minutes relative to the current one. For example executing

gn4:story[fn:inRelativeMinutes(@modifiedDate,0,15)]

returns all stories modified from ‘0’ = beginning of the current minute to ‘15’ = fifteen minutes from now – i.e. if today is August 20th, 2012 at 5:40 PM local time the query returns all stories modified on August 20th 2012 between 5:40 PM (included) and 5:55 PM (excluded) local time.

See the explanation of fn:inRelativeYears() above for details about the use of infinite values to generate open-ended tests.

As for fn:inRelativeYears() if *expr* is a node-set the function returns true if any of the nodes matches the condition.

fn:inRelativeMinutesZ(*expr*, *fromexpr*, *toexpr*)

A variant of the ‘fn:inRelativeMinutes()’ function that uses UTC times. For example executing

gn4:story[fn:inRelativeMinutes(@modifiedDate,0,15)]

if today is August 20th, 2012 at 3:40 PM UTC time the query returns all stories modified on August 20th 2012 between 3:40 PM (included) and 3:55 PM (excluded) UTC.

fn:inDays (*expr*, *fromexpr*, *toexpr*)

Checks that the date-time or date represented by the *expr* string is within the *fromexpr* (included) and *toexpr* (excluded) days. *fromexpr* and *toexpr* must be strings or string parameters in the format YYYY-MM-DD - to use the client time zone – or YYYY-MM-DDZ – to use UTC. For example

gn4:story[fn:inDays(@creationDate,'2012-08-28','2012-08-29')]

returns all the stories that have been created on August 28th, 2012 – client local time, whereas

gn4:story[fn:inDays(@creationDate,'2012-08-28Z','2012-08-29Z')]

returns all the stories that have been created on August 28th, 2012 UTC time.

Each day is considered as starting at the time specified by the ‘StartDayTime’ value in the GlobalConfig configuration – by default midnight. If for example ‘StartDayTime’ is set to 6 AM the last example generates this test:

@modifiedDate >= ‘2012-08-28T06:00:00Z’ and @modifiedDate < ‘2012-08-29T06:00:00Z’

Use empty strings for *fromexpr* or *toexpr* to generate open-ended tests, for example

gn4:story[fn:inDays(@creationDate,'2012-08-28','')]

returns all the stories that have been created on or after August 28th, 2012, and

gn4:story[fn:inDays(@creationDate,'','2012-08-29')]

returns all the stories that have been created before August 29th, 2012.

fn:inDays0 (*expr*, *fromexpr*, *toexpr*)

A variant of the ‘inDays()’ function that does not uses the start day time – i.e. that consider always the days as beginning at midnight. For example

gn4:story[fn:inDays0(@creationDate,'2012-08-28Z','2012-08-29Z')]

generates this test:

@modifiedDate >= ‘2012-08-28T00:00:00Z’ and @modifiedDate < ‘2012-08-29T00:00:00Z’

regardless of the value of ‘StartDayTime’ in GlobalConfig

fn:fullText(*attributeexpr*, *searchexpr*)

Full-text search function. a*ttributeexpr* indicates a full-text searchable attribute, *searchexpr* is a string describing the a full-text search expression and the function returns true when the content of the attribute matches the search expression. For example:

gn4:story[fn:fullText (gn4:title,'Obama')]

returns all the stories with a title containing the word ‘Obama’

See the full-text searches chapter below for more details.

### Sorting

By default the objects returned by a query are sorted in ascending order by their id. Use the ‘order by’ clause to specify a different ordering, e.g.:

gn4:story order by gn4:title

returns all the stories in the system sorted in ascending order by their title (a story titled ‘A’ comes before a story titled ‘Z’). Use the ‘descending’ modifier to reverse the sort, e.g.:

gn4:story order by gn4:title descending

returns all the stories in the system sorted in descending order by their title (a story titled ‘Z’ comes before a story titled ‘A’). There is also an ‘ascending’ modifier – e.g.:

gn4:story order by gn4:title ascending

returns all the stories in the system sorted in ascending order by their title as the first example, because sorts by default are ascending.

To sort using multiple attributes just list them separated by comma - each one optionally followed by the ‘descending’ modifier, e.g.:

gn4:story order by gn4:authors descending, gn4:title

returns all stories sorted first by their author names in descending order, and then by their title in ascending order (for those stories that have the same authors).

### XQuery expressions

Besides the XPath syntax used in examples so far, GNQuery supports also a very simple version of the XQuery syntax. For example, the XPath query returning all stories with a specific title:

gn4:story[gn4:title='My title']

can be written using XQuery syntax as:

for $s in gn4:story where $s/gn4:title='My title'

See <http://www.w3schools.com/xquery/default.asp> for a quick introduction to XQuery.

Compared with full-blown XQuery implementation GNQuery supports only the ‘for’, ‘where’, ‘order by’ (already seen above) and ‘return’ clauses, uses XPath 1.0 for its expressions instead than XPath 2.0 and does not implement tuples or any templating to return complex XML values.

The ‘return’ clause modifies queries to return only specific values – e.g.:

for $s in gn4:story where $s/gn4:title='My title' return $s/@name

returns the names of all the stories with the specified title. The same query can be written using an XPath as:

gn4:story[gn4:title='My title']/@name

Another example:

for $i in gn4:image return $i/@pixWidth\*$i/@pixHeight

returns the pixel area of all images in the system. Note that this query can be written only using an XQuery expression and not with a single XPath.

Queries returning specific values cannot be used when fetching objects – e.g. by ‘srv4 export’ – but only internally by code expecting those values.

When writing XQuery expressions the ‘dummy variable’ declared in the ‘for’ clause can be omitted in the ‘where’, ‘order by’ and ‘return’ clauses – this is not the case in standard XQuery implementations. For example

for $s in gn4:story where $s/gn4:title='My title' return $s/@name

can be written as

for $s in gn4:story where gn4:title='My title' return @name

### Spiked objects

The virtual XML used by GNQuery includes also the spiked (logically deleted) objects. They are indentified by an extra ‘nav:spiked’ XML element – so for example a spiked story appears in the virtual XML as:

<gn4:story id="obj33478". . .>

<nav:spiked Code="Normal" Time="..." LoginGuid="..." PurgeTime="...">

<gn4:user id="obj4">

<nav:refObject>

<gn4:user . . .>

. . .

</gn4:user>"

</nav:refObject>

</gn4:user>

</nav:spiked>

. . .

</gn4:story>

Check for the presence – or absence – of this element to write queries returning only spiked or non-spiked objects, for example:

gn4:image[not(nav:spiked)]

returns all non-spiked images.

As seen in the example above, the ‘nav:spiked’ element includes various details: the spiking code in the ‘Code’ attribute, when the object has been spiked in the ‘Time’ attribute, the associated purge time (if any) in the ‘PurgeTime’ attribute and the complete details of the user that spiked the object in the ‘gn4:user’ sub-element. These details can be used to write more specific queries for spiked objects, for example

gn4:image[nav:spiked/@Code='Auto']

returns all images that have been spiked with code ‘Auto’;

gn4:image[nav:spiked/@Time>='2012-01-01T00:00:00Z']

returns all images that have been spiked since the beginning of 2012 (UTC);

gn4:image[nav:spiked/@PurgeTime>='2012-09-01T00:00:00Z']

returns all images that are going to be purged on or after September 1st 2012 (UTC);

gn4:image[nav:spiked/gn4:user/nav:refObject/gn4:user/@name='Editor']

returns all images that have been spiked by the user ‘Editor’.

System commands that fetch objects like ‘srv4 export’ usually automatically add a ‘non spiked’ condition – so

srv4 export -conditions "gn4:image" -out c:\temp\i.xml

exports all non-spiked images, even if there are no explicit conditions on ‘nav:spiked’ in the query expression.

### Audit

The virtual XML used by GNQuery includes also audit informations. They are indentified by an extra ‘nav:audit’ XML element containing ‘nav:actionDesc’ elements.

Example:

<gn4:story id="obj33478". . .>

<nav:audit handle="1">

<nav:actionDesc Action="…" Time="…" LoginGuid="…" UserId="4" UserDesc="…">

<user idref="…">

<nav:refObject idref="…" >

<user … />

</nav:refObject>

</user>

</nav:actionDesc>

</nav:audit>

. . .

</gn4:story>

Get all the stories modified by a specific user:

gn4:story[nav:audit/nav:actionDesc/@UserDesc='user desc' and nav:audit/nav:actionDesc/@Action='Update']

or:

gn4:story[nav:audit/nav:actionDesc/gn4:user/nav:refObject/gn4:user/@name='user' and nav:audit/nav:actionDesc/@Action='Update']

Get all the stories modified after a specific date time:

gn4:story[nav:audit/nav:actionDesc/@Time >= '2012-08-03T00:00:00.00Z' and nav:audit/nav:actionDesc/@Action='Update']

It is also possible to query the audit nodes directly to do query like (this example counts the total actions done by a user):

count(nav:audit/nav:actionDesc[@UserDesc='user desc'])

or count the objects created by a user after a specific date time:

count(nav:audit/nav:actionDesc[@UserDesc='user desc' and @Action='Create' and @Time >= '2012-08-03T00:00:00.00Z'])

### Check-out info

The virtual XML used by GNQuery includes also check-out informations. They are indentified by an extra ‘nav:checkOuts’ XML element containing ‘nav:checkOut’ elements.

Example:

<gn4:story id="obj33478". . .>

<nav:checkOuts>

<nav:checkOut Code="Content" Time="..." UserId="..." ClientSpecId="...">

<user idref="...">

<nav:refObject idref="..." />

</user>

<clientSpec idref="...">

<nav:refObject idref="..." />

</clientSpec>

</nav:checkOut>

<nav:checkOut Code="Normal" Time="..." LoginGuid="..." UserId="...">

<user idref="...">

<nav:refObject idref="..." />

</user>

</nav:checkOut>

</nav:checkOuts>

. . .

</gn4:story>

Get all the articles checked out by a specific user:

gn4:article[nav:checkOuts/nav:checkOut/gn4:user/nav:refObject/gn4:user/@name='TeraDP\\mimo' and nav:checkOuts/nav:checkOut/@Code = 'Content']

Get all the articles checked out after a specific time:

gn4:article[nav:checkOuts/nav:checkOut/@Time >= '2012-08-03T00:00:00.00Z']

### Taks and TaskLogs

The virtual XML used by GNQuery includes also tasks and taskLogs informations. They are indentified by an extra ‘nav:taks’and ‘nav:taskLogs’ XML element containing ‘nav:task’ and ‘nav:taskLog’ elements.

Example:

<nav:tasks handle="1">

<nav:task

id="..." userId="..." time="..." priority="..." type="..." objectId="..."

profile="..." numRetries="..." cmd="...">

<user idref="...">

<nav:refObject idref="...">

<user id="..." name="..." . . . />

</nav:refObject>

</user>

<object idref="...">

<nav:refObject idref="...">

<pubDest id="..." name="..." path="..." ... />

</nav:refObject>

</object>

</nav:task>

...

</nav:tasks>

<nav:taskLogs handle="1">

<nav:taskLog

id="..." time="..." result="..." type="..." priority="..."

execProc="..." profile="..." objectid="..." cmd="..." userId="...">

<message>

...

</message>

<user idref="...">

<nav:refObject idref="..." />

</user>

<object idref="...">

<nav:refObject idref="..." />

</object>

</nav:taskLog>

...

</nav:taskLogs>

Get all objects that have at least an associated pending task:

gn4:\*[nav:tasks/nav:task]

Get all videos that had at least one associated task that failed:

gn4:video[nav:taskLogs/nav:taskLog/@result='Error']

### Object type attributes

The system schema can define attributes associated with the object types themselves – also known as meta-attributes. These attributes are rendered in a special ‘nav:objectTypeAttrs’ element in the object’s XML, e.g.:

<audio nav:objectType="audio" id="obj44013" . . .>

<nav:objectTypeAttrs mediaType="Audio" . . . />

. . .

</audio>

Use this element to write queries based on object type attributes, for example:

gn4:archiveObject[nav:objectTypeAttrs/@mediaType='Audio']

returns all the archive objects with a media type ‘Audio’, and

gn4:txt[nav:objectTypeAttrs/@typeKind='body']

returns all article’s text of kind ‘body’.

### Parameters

It is possible to pass parameters to queries – i.e. values identified by a name that can be used in the query expressions. For example the query

gn4:story[gn4:title=$title]

returns all stories with a title equal to the parameter ‘title’. Parameters are specified in srv4 export using the ‘-pars’ option, e.g.:

srv4 export -conditions "gn4:story[gn4:title=$title]" -pars title:Test -out c:\temp\s.xml

exports all stories with title ‘Test’. All the parameters used in the query expression must be specified – missing parameters cause an error – e.g.:

srv4 export -conditions "gn4:story[gn4:title=$title]" -out c:\temp\s.xml

causes:

Error in <string>: Ln: 1 Col: 27

gn4:story[gn4:title=$title]

^

Unknown variable 'title'

(ERR1332)

Parameters can be either type or un-typed. Un-typed parameters are simply strings that GNQuery will try to convert to the appropriate type depending on where they are used in the query expression. Parameters passed via command line or URL are always un-typed.

In the example above ‘$title’ was compared to a string, so it remained a string – in this case though:

gn4:story[@nWords>=$minWords]

‘$minWords’ is compared to a numeric value, so the un-typed string will be converted by GNQuery to a number, and there will be an error if such string is not a valid number.

Similarly, un-typed parameters that are compared to date-time attributes are automatically converted to a date-time, e.g. in:

gn4:story[@modifiedDate>=$from]

the value of ‘$from’ is converted to a date-time. If the string does not explicitly specify an UTC date-time using the ‘Z’ suffix it is assumed to be a local date-time, and GNQuery automatically converts it to a UTC value using the time-zone of the current login.

Single parameters can specify a list of values when they are passed to functions expecting multiple values of the same type. For example in

gn4:story[fn:in(@id,$ids)]

the un-typed parameter ‘$ids’ is expected to be a list of comma-separated strings because ‘fn:in’ accepts multiple (unlimited) parameters of the same type, so

srv4 export -conditions "gn4:story[fn:in(@id,$ids)]" -pars ids:obj312,obj313 –out . . .

exports the stories with database id 312 and 313. To specify the database ids as number use the ‘fn:objectIdFromString()’ function – e.g.:

srv4 export -conditions "gn4:story[fn:in(fn:objectIdFromString(@id),$ids)]"

-pars ids:312,313 –out c:\temp\s.xml

has the same result of the previous example.

Typed parameters can be used only internally from code; they are values of different types (integers, strings, date-time, array of integers etc.) that are passed to GNQuery and that are used as-is, without conversions. As such they must match the expected type within the query expression – so in

gn4:story[@nWords>=$minWords]

the typed parameter ‘$minWords’ must be an integer, and in

gn4:story[fn:in(fn:objectIdFromString(@id),$ids)]

the typed parameter ‘$ids’ must be an array of integers.

Typed date-time parameters with kind ‘Local’ or ‘Unspecified’ are automatically converted to UTC using the time zone of the current login.

### Query context

Beside the parameters explicitly passed to a query (see above) there is also a pre-defined parameter ‘$context’ containing various data about the user and the client that is executing the query.

‘$context’ contains an XML fragment like this:

<lc:LoginContext

xmlns:lc="http://www.teradp.com/schemas/GN4/1/LoginContext.xsd"

UserId="obj1705"

UserName="teradp\mimo"

HomeFolderId="1431"

HomeFolderName="mimo"

HomeFolderPath="/system/users/mimo">

<Pars>

<add key="..." value=="..."/>

...

</Pars>

</lc:LoginContext>

<oc:OperationContext

xmlns:oc="http://www.teradp.com/schemas/GN4/1/OperationContext.xsd"

UtcNow="2012-08-22T15:43:02.002Z"

Now="2012-08-22T10:43:02.002"/>

Use this information to write queries based on the current user or on the execution date-time. For example

gn4:story[gn4:creatorRef/@idref=$context/lc:LoginContext/@UserId]

returns all stories created by the current user, whereas

gn4:task[@dueDate < $context/oc:OperationContext/@UtcNow]

returns all tasks that have a due-date in the past.

gn4:variantType[@name=$context/lc:LoginContext/lc:Pars/lc:add[@key='archiveVariantName']/@value]

returns the variant type specified by the archiveVariantName parameter of the current user.

Note how ‘$context’ in GNQuery has the same structure of the ‘$context’ parameter available in XSL transformation, but it contains less information.

## Attribute types

In general it is possible to create searches using attribute of any type just looking at their XML representation, and then writing XPath expressions implementing the desired search conditions against this XML. What follows is a more detailed description on how to create search conditions using the various attribute types supported by the system, including some tricks and limitations.

### String attributes

The XML representation of a string attribute is the string itself as either the value of an XML attribute:

<gn4:story name="story name". . .>

or the text inside an XML element:

<gn4:story>

<gn4:title>Story title<gn4:title>

. . .

</gn4:story>

Use the equality operator on this XML attributes or elements to search for string attributes with a specific value, e.g.:

gn4:story[@name=’story name’]

returns all stories named ‘story name’.

Use the ‘contains()’, ‘starts-with()’ and ‘fn:ends-with()’ functions to search for string attributes containing / starting with / ending with specific values, e.g.

gn4:story[contains(gn4:title,'title')]

returns all stories with a title containing ‘title’ in any position.

String attributes with a maximum length of more than 1300 characters are stored in the database as text columns instead than normal string columns; this limits the kind of tests that can be used when creating queries, because text columns cannot be directly compared to a value. For example the ‘summary’ attribute of the story objects in the standard schemas has a maximum length of 100,000, hence it is stored in a text column, and so a query using direct comparison:

gn4:story[gn4:summary='the summary']

fails with the error

'gn4:summary' cannot be compared: it evaluates to a long text instead than a string (ERR1332)

It is still possible to do searches on such attributes using the functions listed above to do partial comparisons, e.g.

gn4:story[contains(gn4:summary,'summary')]

works fine, returning all stories with a summary containing the word ‘summary’.

Note that 1300 is the length limit in version 1.5, it could change in future versions.

### Date-time attributes

The XML representation of date-time attributes is as an XML attribute or XML element containing a standard string rendering of the date-time in UTC: yyyy-mm-ddThh:mm:ss.ffffZ – e.g. ‘2013-01-02T17:30:00.000Z’ for five thirty PM of January 2nd, 2013.

When comparing date-time attributes to constants:

gn4:story[@modifiedDate>’2013-01-01T15:00:00Z’]

GNQuery converts the string constant into a date-time, that is then used in the comparison with the database column. If ‘Z’ is not explicitly specified the date-time is assumed to be local, and it is converted to UTC using the current login time-zone before being passed to the SQL server. This means that the query above returns all the stories modified after 3 PM of January 1st 2013 UTC, whereas:

gn4:story[@modifiedDate>’2013-01-01T15:00:00’]

returns all the stories modified after 3 PM of January 1st 2013 local time.

### XML attributes

Searches on XML attributes are currently not supported, attempts to access such attributes in query expressions fail – e.g.

gn4:story[gn4:xmlText]

fails with the error

There is no element 'gn4:xmlText' in '/gn4:story' (ERR1332)

### Reference attributes

The XML representation of a reference attribute is an XML element named as the attribute, with the database id of the referenced object in its ‘idref’ attribute, and containing a sub-element ‘nav:refObject’ holding the complete XML representation of the referenced object; e.g. something like this:

<gn4:story id="obj33478". . .>

. . .

<gn4:folderRef idref="obj31860">

<nav:refObject idref="obj31860">

<gn4:folder id="obj31860" . . . >

. . .

</gn4:folder>

</nav:refObject>

</gn4:folderRef>

. . .

</gn4:story>

for the folder reference of a story. Based on this XML representation, the query to return all stories in a folder with database id 31860 is:

gn4:story[gn4:folderRef/@idref='obj31860']

Note how the id must be specified as a string with the ‘obj’ prefix, because this is its XML representation. Use the ‘nav:refObject’ sub-element to create conditions based on the attributes of the referenced object, for example

gn4:story[gn4:folderRef/nav:refObject/gn4:folder/@path='/system/users/Guest']

returns all the stories in the folder with path ‘/system/users/Guest’.

The XML representation of every object includes a ‘nav:referencing’ sub-element listing all the objects that reference the current one (again, like the standard navigation XML), e.g. for a folder:

<gn4:folder id="obj31860" . . .>

. . .

<nav:referencing>

<nav:referencingBy attrName="story.folderRef">

<nav:refs handle="1">

<nav:refObject idref="obj33477">

<gn4:story . . .>

. . .

</gn4:story>"

</nav:refObject>

. . .

</nav:refs>

</nav:referencingBy>

. . .

</nav:referencing>

. . .

</gn4:folder>

Use this element to create queries based on the referencing (as opposed to the referenced) objects, e.g.

gn4:folder[nav:referencing/nav:referencingBy[@attrName='story.folderRef']

/nav:refs/nav:refObject/@idref='obj33477']

returns the folder containing (i.e. referenced by) the story with database id 33477, and

gn4:folder[nav:referencing/nav:referencingBy[@attrName='story.folderRef']

/nav:refs/nav:refObject/gn4:story/gn4:title='My title']

returns all the folders containing stories titled ‘My title’.

Note that the referencing element used by GNQuery lists ALL possible referencing attributes, whereas the navigation XML does not contains the derived referencing attributes – so the GNQuery XML contains ‘folderObject.folderRef’, ‘archiveObject.folderRef’, ‘story.foldeRef’, ‘image.folderRef’ etc. (all attributes derived from the root ‘folderObject.folderRef’), the navigation XML contains only ‘folderObject.folderRef’.

### Multiple reference attributes

The XML representation of multiple reference attributes is as a sequence of reference elements – each one including also the sub-attributes (if any); e.g. something like this:

<gn4:story id="obj41196">

. . .

<gn4:objs nav:handle="1">

<gn4:ref idref="obj41197">

<gn4:title>Sub title</gn4:title>

<gn4:caption>Sub caption</gn4:caption>

<nav:refObject idref="obj41197">

<gn4:image id="obj41197" . . .>

</gn4:image>

</nav:refObject>

</gn4:ref>

. . .

</gn4:objs>

. . .

</gn4:story>

for the assets associated with a story via the ‘objs’ multiple reference attribute. Based on this representation, the query to find all the stories associated with the object with database id 41197 is

gn4:story[gn4:objs/gn4:ref/@idref='obj41197']

The query to find all the stories associated with an image titled ‘Test’ is:

gn4:story[gn4:objs/gn4:ref/nav:refObject/gn4:image/gn4:title='Test']

Sub-attributes can be used as well, so

gn4:story[gn4:objs/gn4:ref/gn4:caption='Sub caption']

returns all stories with an associated object caption ‘Sub caption’. Conditions on sub-attributes and referenced objects can be combined of course, for example

gn4:story[gn4:objs/gn4:ref[@idref='obj41197' and contains(gn4:caption,'Israel')]]

returns all stories associated with the object with database id 41197 with a caption containing the word ‘Israel’. Note that this is not the same thing as

gn4:story[gn4:objs/gn4:ref/@idref='obj41197' and

contains(gn4:objs/gn4:ref/gn4:caption,'Israel')]

that returns all stories that are associated object with database id 41197 and that also have an associated object (not necessarily the same one) with a caption containing the word ‘Israel’.

The same ‘nav:referencing’ representation is available for multiple reference objects, so the query

gn4:image[nav:referencing/nav:referencingBy[@attrName='story.objs']

/nav:refs/nav:refObject]

returns all images associated to at least one story, and

gn4:image[nav:referencing/nav:referencingBy[@attrName='story.objs']

/nav:refs/nav:refObject/@idref = 'obj41196']

returns all images associated with the story with database id 41196.

### Children attributes

The XML representation of a children attribute is an XML element named as the attribute directly containing the XML representation of the children objects contained in the attribute; e.g. something like this:

<article id="obj41143" . . .>

. . .

<txts>

<head id="obj41172" . . .>

. . .

</head>

<body id="obj41144" . . . >

. . .

</body>

</txts>

. . .

</article>

for the texts of an article. Based on this XML representation, the query to return all articles with a ‘head’ text is:

gn4:article[gn4:txts/gn4:head]

Similarly, to find the articles with a ‘head’ text with a specified name use:

gn4:article[gn4:txts/gn4:head/@name='Copy of New story,hd']

To find all articles containing 8 or more texts use:

gn4:article[count(gn4:txts/gn4:\*)>=8]

and to find all articles with at least 4 photos (‘photoCaption’ texts) use:

gn4:article[count(gn4:txts/gn4:photoCaption)>=4]

The XML representation of every child object includes ‘nav:referencing/nav:parentBy’ sub-elements listing its parent objects (again, like the standard navigation XML), e.g. for a ‘head’ text:

<head id="obj41172" . . .>

. . .

<nav:referencing>

<nav:parentBy attrName="article.txts">

<nav:refs handle="1">

<nav:refObject idref="obj41143">

<article id="obj41143" . . . >

. . .

</article>

</nav:refObject>

</nav:refs>

</nav:parentBy>

. . .

</nav:referencing>

. . .

</head>

Use these elements to create queries based on the parents of an object, for example:

gn4:txt[not(nav:referencing/nav:parentBy[@attrName='article.txts']

/nav:refs/nav:refObject)]

returns all ‘txt’ objects that do not have any parent via the attribute ‘article.txts’ – i.e. all texts that are not in an article. This query:

gn4:txt[nav:referencing/nav:parentBy[@attrName='article.txts']

/nav:refs/nav:refObject/@idref = 'obj41224']

returns all texts that are in the article with the specified database id (41224), and this one:

gn4:txt[nav:referencing/nav:parentBy[@attrName='article.txts']

/nav:refs/nav:refObject/\*/@name = 'fashion1']

returns all texts that are in an article named ‘fashion1’ .

### Data content attributes

The XML representation of data content attributes is an XML element named as the attribute with XML attributes specifying the id, the size, the MIME type, the file extension and the Mac file type of the associated data, e.g.:

<image id="obj33477" . . .>

. . .

<thumbnail id="data11488" size="10649" mime="image/jpeg"

nav:mimeExt=".jpg" nav:mimeMacType="JPEG" />

. . .

</image>

Based on this XML representation, the query

gn4:image[gn4:thumbnail/@id='data843']

returns the image with a thumbnail with the specified data id (843); the query

gn4:image[gn4:thumbnail/@size>1024]

returns all images with a thumbnail bigger than 1 Kbyte (1024 bytes); the query

gn4:image[gn4:thumbnail/@mime='image/jpeg']

returns all images with a JPEG thumbnail.

If a data content attribute is empty the corresponding XML element is missing from the XML representation of the object, so the query

gn4:image[not(gn4:thumbnail)]

returns all images without a thumbnail.

### Extended referencing

The virtual XML used by GNQuery includes also extended referencing information. They are indentified by an extra extRefs element

<audio nav:objectType="audio" id="..." name="..." modifiedDate="..." creationDate="..." size="..." rightsStatus="..." eventDate="..." duration="..." sampleRate="..." album="..." xmlns="..." xmlns:nav="http://www.teradp.com/schemas/GN4/1/ObjNav.xsd">

<nav:referencing>

<nav:referencingBy attrName="package.objs">

<nav:extRefs handle="1">

<ref idref="...">

<title/>

<caption>...</caption>

<nav:refObject idref="...">

<story nav:objectType="story" id="..." name="..." modifiedDate="..."

creationDate="..." size="..." rightsStatus="..." nWords="..." nChars="..."/>

</nav:refObject>

</ref>

</nav:refs>

</nav:referencingBy>

</nav:referencing>

</audio>

It is possible to do query like:

Get all the audio referenced by a specific package or story with a caption containing a specific word:

gn4:audio[nav:referencing/nav:referencingBy[@attrName='package.objs']/nav:extRefs/gn4:ref[@idref='obj63942' and contains(gn4:caption,'caption')]]

Get all the users with Delete permissions on a specific folder:

gn4:user[nav:referencing/nav:referencingBy[@attrName='folder.access']/nav:extRefs/gn4:securityAccess[@idref='obj6909' and gn4:grant='Delete']]

Get all categories used by calendar objects with a weight under 50:

gn4:category[nav:referencing/nav:referencingBy[@attrName='calendarObject.cats']/nav:extRefs/gn4:ref[@weight<50]]

Get all pub modes used to publish a specific object (id 44043) on a specific pub dest (id 1595):

gn4:pubMode[nav:referencing/nav:referencingBy[@attrName='pubDest.objs.pubModeRef']/nav:extRefs/gn4:ref[@idref='obj44043' and gn4:pubModeRef/@idref='obj1595']]

Get all users that created a libel tag for a specific object (id 3626) with a description containing a specific word ('libel'):

gn4:user[nav:referencing/nav:referencingBy[@attrName='archiveObject.libelTags.creatorRef']/nav:extRefs/gn4:item[gn4:creatorRef/@idref='obj3626' and contains(gn4:description,'libel')]]

### Descendants and ‘in sub-tree’ queries

The virtual XML of objects with children attributes or with self-referencing attributes (reference attributes referencing the same object type) contain nav:descendant elements listing all the descendant object via a specified children or self-referencing attribute.

For example category objects have the children attribute categories containing all their sub-categories, that can contain sub-sub categories and so on, creating category trees. Their virtual XML is something like this:

<category id="obj123" . . .>

<nav:descendant attrName="categories">

<nav:refObject idref="obj123">

<category id="obj123" .="" .="" .=""> . . . </category>

</nav:refObject>

<nav:refObject idref="obj124">

<category id="obj124" .="" .="" .=""> . . .</category>

</nav:refObject>

<nav:refObject idref="obj125">

<category id="obj125" .="" .="" .=""> . . .</category>

</nav:refObject>

<nav:refObject idref="obj126">

<category id="obj126" .="" .="" .=""> . . .</category>

</nav:refObject>

</nav:descendant>

. . .

<categories>

<category id="obj124" . . .>

<categories>

<category id="obj125" . . .>

</category>

</categories>

</category>

<category id="obj126" . . .>

</category>

</categories>

. . .

</category>

Via its children attribute the category with id 123 has descendant categories with id 124, 125 and 126 – where the ones with id 124 and 125 are direct children and the one with id 125 is a ‘grand child’ – child of a child. The corresponding nav:descendant element is identified by its attrName=”categories” and contains one nav:refObject sub-element for each descendant category – including the root category itself (the one with id 123 in the example).

Similarly, folder objects have a folderParent attribute that references a folder – i.e. a self-referencing attribute. This attribute creates trees of folders: all the folder referencing a root one are its children, the folder referencing the referencing ones are the grand children and so on. The folder virtual XML is like this:

<folder id="obj7" . . .>

<nav:descendant attrName="folderParent">

<nav:refObject idref="obj7">

<folder id="obj7" . . .> . . . </folder>

</nav:refObject>

<nav:refObject idref="obj8">

<folder id="obj8" . . .> . . . </folder>

</nav:refObject>

<nav:refObject idref="ob1289">

<folder id="obj1289" . . .> . . . </folder>

</nav:refObject>

<nav:refObject idref="obj1290">

<folder id="obj1290" . . .> . . . </folder>

</nav:refObject>

</nav:descendant>

. . .

</category>

Where the folders with id 8, 9, 1289 and 1290 are all the folders descending – directly or indirectly – from the one with id 7 via the folderParent attribute. Note that as in the children case the folder itself is listed inside the nav:descendant element. Note also that the attribute name specified by attrName does not have the object type prefix, as opposed to the one used in nav:referencingBy (see above) – e.g. it is just “folderParent” instead than “folder.folderParent”.

This nav:descendant element is used to write ‘in sub-tree’ queries - i.e. queries returning the objects referencing an entire sub-tree of ‘container’ objects like folders. For example:

gn4:article[gn4:folderRef/@idref=

/gn4:folder[@id='obj7']/nav:descendant[@attrName='folderParent']/nav:refObject/@idref]

returns all the articles contained in the folder sub-tree rooted at the folder with id 7 or, in other words, all the articles referencing the folder with id 7 or any of its descendant folders. The query works because:

/gn4:folder[@id='obj7']/nav:descendant[@attrName='folderParent']/nav:refObject/@idref

returns the ids of all the folder in the sub-tree, and so the condition ‘[gn4:folderRef/@idref=...]’ is true for all the articles referencing any of those folders via their folderRef attribute.

Another example – using categories this time:

gn4:story[gn4:cats/gn4:ref/@idref=/gn4:category[@name='international']

/nav:descendant[@attrName='categories']/nav:refObject/@idref]

returns all the stories that have at least one category equal to ‘international’ or any of its descendants.

Another use of the nav:descendant element is to write queries returning trees of objects, for example

gn4:pubDest[@name='GNWeb']/nav:descendant[@attrName='pubDestParent']

/nav:refObject/gn4:pubDest

returns the publishing destination named ‘GNWeb’ and all its sub- publishing destinations.

## Full-text searches

Full-text searches use the special function fn:fullText(*attributeExpr*, *searchExpr*); for example:

gn4:story[fn:fullText(gn4:title, 'Obama')]

returns all stories containing the word ‘Obama’ anywhere in their title, and

gn4:story[fn:fullText(nav:text, 'Obama')]

returns all stories containing the word ‘Obama’ anywhere in the full-text indexed attributes.

fn:fullText() behaves like a Boolean function returning true when the content of the attribute specified by the first argument matches the full-text search expression specified by the second argument (e.g. similar to the contains() function), but ‘under the hood’ it is translated in a full-text query executed by the full-text search engine installed with the system.

Due to this translation, the first argument of fn:fullText()must be the path of a full-text searchable object attribute, or the special path ‘nav:text’ indicating all the full-text indexed data.

The second argument must be either a string literal or a string variable specifying a valid search expression.

This means that these expression are NOT valid:

fn:fullText(concat(gn4:title, gn4:authors), 'Obama') -- WRONG

the first argument is a function call and not just the path of an object attribute;

fn:fullText(gn4:title, gn4:authors) -- WRONG

the second argument is a path and not a string literal or string variable;

fn:fullText(gn4:title, 'Obama and') -- WRONG

the second argument is not a correct search expression (see below).

### Search expressions

The simplest possible search expression is a simple word:

Obama

matches any text containing ‘Obama’. Note that it does not match texts containing ‘Obamacare’ – i.e. the match must be with a whole word.

Any sequence of alphanumerical character is a valid search word, with the exception of the reserved words ‘and’, ‘or’, ‘not’, ‘before’, ‘near’ (see below). Non-alphanumeric characters like punctuation or symbols cannot be used directly in a search word, they need to be placed in a search phrase (see below).

#### Logical (Boolean) operators

Multiple words separated by spaces are interpreted as a logical ‘**and**’:

Biden Obama

matches any text containing both ‘Biden’ and ‘Obama’ – in any order and position. The ‘and’ can be made explicit:

Biden and Obama

is exactly the same as the ‘Biden Obama’ above.

The two other possible logical operators are ‘**or**’ and ‘**and not**’. For example

Biden or Obama

matches any text containing ‘Biden’ or containing ‘Obama’ or containing both; whereas

Biden and not Obama

matches any text that contains ‘Biden’ but does not contain ‘Obama’.

The operator names ‘and’, ‘or’, ‘and not’ are case-sensitive: they must be lower-case.

#### Phrases

Search for phrases (sequences of multiple words) enclosing them with ‘/’, e.g.:

/president Obama/

matches texts containing the phrase ‘president Obama’ as is.

Use the same delimiter to search for reserved words – e.g. to search for texts containing the word ‘and’ use:

/and/

Finally, use phrases to search for non-alphanumeric characters, e.g.:

/ASP.NET/

searches for texts containing the phrase ‘ASP.NET’. Note though that punctuation and other non-alphanumeric characters are typically considered equivalent to spaces by the full-text search engines – so the above search is equivalent to /ASP NET/. Similarly, to search for ‘OS/2’ use simply:

/OS 2/

because the slash is equivalent to a space.

#### Proximity searches

To search for words near each other use the ‘**near**’ and ‘**before**’ operators; e.g.:

president near Obama

matches all texts containing both ‘president’ and ‘Obama’ within 5 words – in any order;

president near/3 Obama

matches all texts containing both ‘president’ and ‘Obama’ within 3 words – again in any order;

president before Obama

matches all texts containing ‘president’ within 5 words before ‘Obama’, and finally:

president before/3 Obama

matches all texts containing ‘president’ within 3 words before ‘Obama’.

As for the logical operators above, the operator names ‘before’ and ‘near’ are case-sensitive: they must be lower-case.

#### Stemming, thesaurus and prefix searches

To search for syntactical variations of a word (stemming) use the suffix ‘**^**’ , e.g.:

president^

matches all texts containing ‘president’, ‘presidential’, ‘presidentially’ etc.

To search for synonyms of a word use the suffix ‘**+**’, e.g.:

Obama+

matches all texts containing any of the synonyms of ‘Obama’ defined in the thesaurus of the full-text search engine.

To search for words staring with a prefix use the suffix ‘**\***’, e.g.:

Obama\*

matches all texts containing a word starting with ‘Obama’ – i.e. it will match a text containing ‘Obamacare’.

Caveats:

* Exalead does not support thesaurus searches, so the suffix ‘+’ is ignored when using Exalead as the full-text search engine
* Full-text search engines implement prefix searches first looking for words starting with the specified prefix in the index, and then doing a full-text search using the resulting list of words. Usually the list generated in the first step is limited in length for performance reason; this means that searches with a prefix that matches a lot of words won’t return all the possible results. E..g a search for ‘a\*’ won’t typically return all texts containing a word starting with ‘a’, it returns texts containing the first N words in the index that starts with ‘a’ – where N depends on the full-text search engine used and on its setup.

#### SpellsLike and SoundsLike

To search for words that sound like or spells like another word use the **spellslike** and **soundslike** operators.

Spellslike works only on Exalead, SoundLike works on Exalead and Oracle database.

soundslike Shakespeare

spellslike Shakespeare

#### Operators’ precedence and parenthesis

The operators’ precedence is:

* phrase (/. . ./) and suffixes (^, +, \*)
* near, before, spellslike, soundslike
* and, and not
* or

i.e. operators higher in the list are executed before operators in a lower position.

This means that in

Biden or Obama and president

‘Obama and president’ is evaluated first, and the result is combined with ‘Biden’ – i.e. the expression matches texts containing either ‘Biden’ or both ‘Obama’ and ‘president’.

Use parenthesis to alter the evaluation order, so

(Biden or Obama) and president

evaluates first ‘Biden or Obama’ and the result is combined with ‘president’ – i.e. the expression matches texts containing both ‘president’ and ‘Biden’ or ‘Obama’.

### Rank

The ‘virtual’ attribute @nav:rank contains the rank (relevance) of an object in the context of the full-text search being executed. It is a number, with bigger values indicating higher = better relevance.

The rank can be used to sort and filter the results of a full-text search, e.g.:

gn4:story[fn:fullText(nav:text, 'Obama')] order by @nav:rank descending

returns all the stories containing ‘Obama’ with the more relevant first; and:

gn4:story[fn:fullText(nav:text, 'Obama') and @nav:rank > 50]

returns all the stories containing ‘Obama’ with a rank (relevance) greater than fifty.

@nav:rank has meaning – and can be used - only in a full-text search, doing something like:

gn4:story[@nav:rank > 50] -- WRONG

causes an error.

Caveat: filtering using the rank is not supported by Exalead.

### Using Exalead – under the hood

When using Exalead as the full-text search engine the system first tries to convert a full-text query into a single Exalead one, if this fails it then try to convert the original query into a combination of an Exalead and SQL queries, if this fail as well the query cannot be executed and the system generates an error.

For example:

gn4:story[fn:fullText(nav:text, 'Obama')]

searches by object type and the entire full-text indexed content, this is translated in the single Exalead query:

corporate/tree=="Top/Source/story" AND ("Obama")

Another example – full text within a specific folder:

gn4:story[fn:fullText(nav:text,'Obama') and gn4:folderRef/@idref='obj8']

searches by object type, full-text indexed content and folder reference – all conditions supported by Exalead with the standard schema, so also this query becomes a pure Exalead one:

corporate/tree=="Top/Source/story" AND ("Obama") AND folderRef==8

Third example – full text within a specific folder and a minimum word count:

gn4:story[fn:fullText(nav:text,'Obama') and gn4:folderRef/@idref='obj8' and @nWords>500]

the number of words is not full-text indexed in the standard schema, so this query cannot be executed entirely by Exalead. The system splits it into two parts: pure full-text:

gn4:story[fn:fullText(nav:text,'Obama')]

and remaining conditions:

gn4:story[gn4:folderRef/@idref='obj8' and @nWords>500]

and converts the first part to an Exalead query and the second to a SQL one. The system then executes the Exlead query and filter the result using the SQL one. Note that the test on the folder is now in the second – SQL – query.

The sort must be executed always by the Exalead query, so all the sort conditions must use attributes that are full-text indexed and sortable by Exalead.