Qizx/open User's Guide

Table of Contents

1. Installing Qizx/open

The installation of Qizx/open is straightforward, provided that Java 1.4 is already installed (JRE 1.4.2 recommended).

- Unpack the qizxopen.zip or qizxopen.tar.gz archives in the desired directory.
- The following files should be found:
    - A short help in the file `README.txt`
    - `LICENSE.txt`: the applicable license.
    - A directory `docs` containing documentation.
    - `qizxopen.jar` : an executable jar that can be invoked directly as a command-line tool (see below).
    - directory `net` and subdirectories: the Java source code. See the `LICENSE.txt` file for information about use conditions.
    - a Ant file `build.xml` allows to rebuild a jar from the source code.

2. Command line application

Qizx comes with a simple demonstration application that can be used for running XQuery scripts from the command line, or for executing queries in interactive mode. This simple tool should not be confused with the XQuery engine itself, which is basically a class library integrable in a variety of Java applications.

This application is implemented by the class `net.xfra.qizxopen.app.XQuery` .

The command line demonstration can be invoked for execution of a file containing one query (if the filename is '-', the standard input is used) :

```
java -jar qizxopen.jar options query_filename
```

or it can be used in interactive mode:

```
java -jar qizxopen.jar
```

In this mode each line typed is interpreted as an independent query. It is possible (though not very convenient) to enter a multi-line query by typing a backslash as first character and ending by a line containing a single dot.

For example, on the prompt `XML Query >`, you type (in italics):

```
XML Query > for $i in 1 to 3 return element x { $i }
  <x>1</x>
  <x>2</x>
  <x>3</x>
-> 3 item(s)
```

To get help about command-line options:

```
java -jar qizxopen.jar -help
```

2.1. Command line options

Command line options can be specified before or after the path of file containing a query.

-input *inputURI*
: specifies the location of a XML document used as input(). Can be any URL supported by Java.

-baseURI *baseURI*
: default base URI for queries and for locating documents.

-modules *moduleBaseURI*
: base URI for locating modules.

-serial
: evaluates a XQuery expression directly into a XML serializer (i.e. without building nodes, in the style of a XSLT processor). The evaluation of the query must yield a document or a single node.

  XML serialization parameters can be specified with options of the form -Xparameter=value, for example -Xindent=yes.

-out *file*
: redirects display of results to a file.

-D*variable_name*=value
: initializes a global variable defined in the query.

-X*option*=value
: sets a XML serialization option. See below for the detail of serialization options.

-collation *collation*
: specifies the default collation.

-timezone *timezone*
: specifies the implicit timezone in duration format (eg -timezone P-5H). By defaut the local timezone is used.

-wrap
: Wrap each item of the results inside an element "item" bearing an attribute giving the type of the item. This is no more the default.

-jt
: Trace lookups of Java methods bound by the Java extension mechanism, and calls to these functions. This helps finding why a Java method cannot be bound.

-tex
: full trace of exceptions.

-help
: prints the help.

-- arguments...

The double dash indicates that all following arguments are passed to the XQuery script: they are accessible by the predefined variable **$arguments** of type `xs:string*`.

2.2. Display of results

Results are now (from version 0.3) displayed in a simpler form: atomic values are separated by a simple space, nodes are separated by a newline.

The former mode, in which each item was wrapped inside an element item with an attribute giving the item type, is still accessible through the option -wrap.

For example, by default the display is like below (the user's input is in italics):

```
XML Query> true(), 1, 0.5, "string", element a {attribute x {1}}
true 1 0.5 string <a x="1"/>
-> 5 item(s)
evaluation time: 0 ms, display time: 1 ms
```

With the option -wrap, the following display is obtained:

```
XML Query> true(), 1, 0.5, "string", element a {attribute x {1}}
<?xml version='1.0'?>
<query-results>
  <item type="xs:boolean">true</item>
  <item type="xs:integer">1</item>
  <item type="xs:decimal">0.5</item>
  <item type="xs:string">string</item>
  <item type="element()">
    <a x="1"/>
  </item>
</query-results>
-> 5 item(s)
evaluation time: 0 ms, display time: 1 ms
```

3. Features

3.1. Implementation-defined features

This section documents features or properties described in the XML Query specifications as "implementation-defined".

Implicit timezone

The implicit timezone is currently the default timezone of the Java locale. It can be set by the API and as an option of the command line utility.

Collations

Collations are supported through Java collators. A custom collation can be registered using the API. The URI of a collation follows the Java convention for locales: for example "en" or "fr-CH" can be used as collation URIs.

- A collation URI can be followed by a "fragment" or "reference" that has the value "primary", "secondary" or "tertiary", defining the "strength" of the collator (see the Java documentation for more details). For example, the expression `contains("The next café", "CAFE", "en#primary")` should return true, because the collation with strength `primary` ignores case and accents.

- The special URIs `codepoint` and `"http://www.w3.org/2003/05/xpath-functions/collation/codepoint"` and refer to the basic Unicode codepoint matching (or absence of collation).

Default collation
The default collation is Unicode codepoint. The default collation can be set by the API and by an option in the command line utility.

Input
Can be defined as a parsed XML document. See the documentation of the command line tool and of the API.

Serialization
An extension function is provided for serialization to files from within XQuery (see below). The command line utility can also directly serialize the result of an evaluation (which must be a whole document).

The supported options are described in the documentation of the `x:serialize` extension function (see the Programmer's Guide[1]).

Pragmas
No pragma is recognized by default. The servlet extensions uses pragmas to specify serialization and XSLT transformations.

Must-understand-extensions
No extension is recognized: an error is always raised when an extension is encountered.

Stable sort
The sort in FLWOR expressions (*order by*) is always stable, with or without the *stable* keyword.

empty least
empty least is the default In *order by* clauses of the FLWOR expression.

Precision of type xs:integer
This type is implemented with Java long (64 bits). This is compatible with the XML Schema standard which specifies that "minimally conforming· processors ·must· support [decimal] numbers with a minimum of 18 decimal digits".

Note: xs:decimal is implemented with unlimited precision (Java BigDecimal). For most applications, there is no performance issue here, but for intensive numeric computations, xs:double is preferable because much more efficient.

3.2. Other features

Static Type-checking
Currently, Qizx enforces a strict control of types (more precisely: of basic types, because schema import is not yet implemented). This policy allows to optimize the execution, especially of arithmetic operators and functions.

It means that static analysis errors will be detected if for example one writes the Fibonacci function like this:

```
declare function local:fibo( $n ) {
   if( $n lt 2 ) then $n
   else local:fibo($n - 1) + local:fibo($n - 2)
}
```

This is because the variable $n and the function have no defined type, so the operators + and - have no matching signature.

To eliminate the errors, the function must be rewritten like follows:

```
declare function local:fibo( $n as xs:integer ) as xs:integer {
   if( $n lt 2 ) then $n
   else local:fibo($n - 1) + local:fibo($n - 2)
}
```

[1] devguide.html

This constraint might be relaxed in future versions of Qizx, however using sloppy typing will be at the cost of execution speed.

Automatic optimization of join queries

Joins are frequently used when processing tabular data. Consider this example:

```
for $client in doc("clients.xml")//client
 return <tr>
    <td>{ $client/id }</td>
    <td>{ sum( for $inv in doc("invoices.xml")//INVOICE
                where $inv/@client-id = $client/id
               return $inv/amount) }</td>
  </tr>
```

It joins `client` elements and `INVOICE` elements through the value of the client `id`, and returns the total amount of invoices related to each client.

If executed naively on a database containing 10,000 clients and 30,000 invoices, the expression might imply 300 millions iterations and require more than one hour to be computed on a recent PC. With the join optimization implemented by Qizx 0.3, the computation time on a Pentium4 2.5GHz is around 2 seconds! (including parsing and output).

A join is detected when the optimizer sees a **for** loop with a **where** clause which is a relation (operators = <= < >= >) between an expression involving the loop variable ($inv in the example above) and an expression involving the control variable of an enclosing loop ($client in the example). The two expressions can have any type, though numeric and string values are specially optimized.

The optimizer has some limitations: currently a where clause with a **or** prevents optimization. It does not treat the possible case where the join relation is a predicate in a path, like in the equivalent query:

```
for $client in doc("clients.xml")//client
 return <tr>
  <td>{ $client/id }</td>
  <td>{
   sum( doc("invoices.xml")//INVOICE[ @client-id = $client/id ]/amount )
  }</td>
 </tr>
```

This latter case will be treated in future versions.

4. XML Catalogs

To access DTD specified in XML documents, Qizx implements the OASIS Open Catalog using Suns's implementation. The `resolver.jar` class library must be present in the classpath.

Catalogs are specified as described in the reference documentation: through `CatalogManager.properties` or through system properties, for example `-Dxml.catalog.files="mycatalog1;mycatalog2".`

For more details, see the Oasis documentation.

5. Extensions

All these extensions are documented in the Programmer's Guide[2]. They are just mentioned here:

XQuery functions

Additional predefined functions. They belong to a private namespace "qizx.extensions" referenced by the **x:** or **qizx:** prefixes.

They implement in particular serialization, error handling, text searching and highlighting.

[2] devguide.html

Extensions to standard XQuery functions

Some standard functions (manipulation of date, time, duration) have been extended or modified to become more powerful or more convenient.

Binding Java methods as supplementary functions

This mechanism (similar to those found in other XSLT and XQuery engines, for example Saxon) provides an easy way to extend XQuery by binding methods of any Java class, making this methods appear as XQuery functions. Arguments and results are automatically converted if possible (number, string, boolean) or can be manipulated as opaque wrapped objects with type `xdt:object`. Qizx also converts Java arrays, vectors and enumerations to XQuery sequences and conversely.

Web applications

This extension uses XQuery as a powerful and convenient Web page template language: the results of an expression evaluation are serialized to the HTTP output stream, or alternately can be piped to a XSLT transformation. The whole Java Servlet API is available through the Java extension mechanism mentioned above, or through convenience functions. This feature is described in a separate document: XML Query Server Pages[3].

[3] doc_xqsp.html