Qizx/open Manual

Axyana Software, XMLmind `<qizx-support@xmlmind.com>`

Qizx/open Manual

by Axyana Software

Version 4.1

Published September 28, 2010

Part I. Installation

Chapter 1. Requirements

Hardware
: Qizx is designed for running on any standard computer supporting a Java™ Runtime Environment.

  The memory size required is widely dependent on applications:

  - It is quite possible to perform queries even on large databases with the default memory size (64 Mb).
  - Performing large transactions (tens of thousands of documents and collections or more) or handling very large documents can require more memory.

    It is in general reasonable to allow for 128 Mb or more. In the case of a server supporting many concurrent queries, it can be worth specifying a large memory size (e.g 512 Mb or more) to benefit from large caches (Qizx adapts the size of caches to the available memory).

Java Virtual Machine (JVM)
: Starting from version 4.0, Qizx requires a JVM version 1.5 or more.

Operating System
: Qizx is supported on the following OS:

  - Microsoft Windows XP, Vista and Seven.
  - Linux 2.6+.
  - Mac OS X 10.5+.
  - In general, any OS derived from Unix, where a Sun™ Hotspot JVM version 1.5 is supported, should be able to run Qizx. However no support can be provided for these platforms.

Additional libraries
: No additional library is required.

  The distribution includes the following utility jars:

  - `resolver.jar`, the XML entity resolver for XML parsing.
  - `jhall.jar`, the Java Help engine for Qizx Studio.

Chapter 2. Installation

Installation of Qizx/open simply consists in unpacking the zipped distribution:

1. Install on Linux

1. Check that the requirements of the previous chapter are met by your platform. In particular, you need a Java Runtime Environment (JRE) version 1.5+. For example:

```
$ java -version
java version "1.5.0_11"
Java(TM) 2 Runtime Environment, Standard Edition (build 1.5.0_11-b03)
Java HotSpot(TM) Client VM (build 1.5.0_11-b03, mixed mode, sharing)
```

2. Unzip the `qizx.zip` package. This will create a `qizx-`*`VVV`* directory where *`VVV`* is the version of Qizx.

   For example:

```
$ cd /usr/share
$ unzip -l /tmp/qizxopen-4.1.zip
$ ls qizxopen-4.1
bin  config  docs  legal  lib  server src.zip
```

   You can directly run the `qizx` or `qizxstudio` shell scripts from any location by giving the proper path:

```
$ qizxopen-4.1/bin/qizxstudio &
```

3. You may want to add the directory *`QIZX_HOME`*`/bin` to your `PATH` environment variable.

2. Manual install on Windows

1. Check that the requirements of the previous chapter are met by your platform. In particular, you need a Java Runtime Environment (JRE) version 1.5+. For example:

```
C:\Program Files> java -version
java version "1.5.0_11"
Java(TM) 2 Runtime Environment, Standard Edition (build 1.5.0_11-b03)
Java HotSpot(TM) Client VM (build 1.5.0_11-b03, mixed mode, sharing)
```

2. Unzip the `qizx.zip` package. This will create a `qizx-`*`VVV`* directory where *`VVV`* is the version of Qizx.

   For example:

```
C:\Program Files> unzip -l \temp\qizxopen-4.1.zip
C:\Program Files> dir qizxopen-4.1
... <DIR> bin
... <DIR> config
... <DIR> docs
... <DIR> legal
... <DIR> lib
... <DIR> server
... <DIR> src.zip
```

   You can directly run the `qizx.bat` or `qizxstudio.bat` batch files from any location by giving the proper path:

```
C:\Program Files> qizxopen-4.1\bin\qizxstudio
```

3. You may want to add the directory *`QIZX_ROOT`*`\bin` to your `PATH` environment variable.

Chapter 3. Content of the distribution

After installation, the following directories should be found in the installed Qizx/open directory:

`docs/`
: Root of the documentation and samples.

  `index.html`
  : Dispatches to the different parts of the documentation.

  `manual.pdf`
  : Qizx manual in PDF form.

  `manual/`
  : Qizx manual in browsable HTML form.

  `javadoc/`
  : Java documentation of the API and utility classes.

  `samples/`
  : Examples (documents, queries, Java code, DTD and catalogs) used by the chapters "Getting started" [6] and "Programming with the Qizx API" [11] of the manual.

`bin/`
: Contains executable scripts:

  `qizx`, `qizx.bat`
  : Scripts for running the command-line tool, respectively on Unix-like platforms (Linux, Mac OS X, others), and MS Windows.

  `qizxstudio`, `qizxstudio.bat`
  : Scripts for running the graphic interface Qizx Studio, respectively on Unix-like platforms (Linux, Mac OS X, others), and MS Windows.

`lib/`
: Contains the run-time jars used by Qizx:

  `qizx.jar`
  : Core Qizx engine.

  `qizxstudio.jar`, `qizxstudio_help.jar`
  : Qizx Studio application.

  `resolver.jar`
  : Apache XML Catalogs resolver for catalog-based entity resolution.

  `jhall.jar`
  : Standard Java Help engine.

`legal/`
: Contains licenses and information for Qizx/db and third-party components used in Qizx.

`src.zip`
: Complete source code.

Part II. User's Guide

Chapter 4. Getting Started with Qizx/open

This section is a simplified version of the tutorial found in Chapter 4, *Getting Started with Qizx/open* [6].

To help experimenting and developing with XML Query, Qizx/open comes with two tools which make it easy to write and execute XML Query scripts:

Qizx Studio [44]
: A graphic tool featuring a simple XML Query workbench with which you can write and execute XML Query scripts, and view the results.

qizx [37]
: A command-line tool which can be used to execute XML Query script files.

1. Writing and running queries with Qizx Studio

Qizx Studio currently provides a basic environment for editing and running XML Query queries.

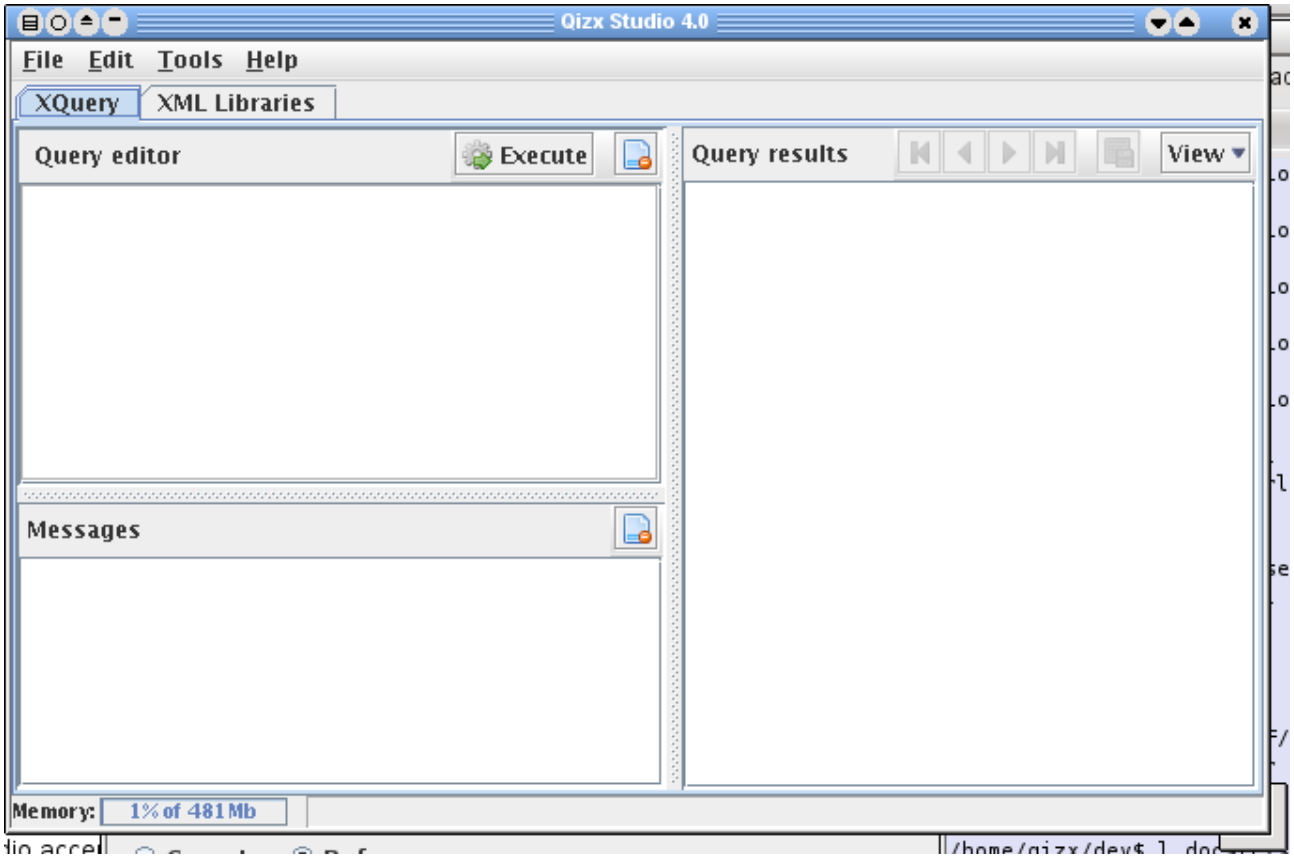
Starting Qizx Studio

- On Windows, the directory bin inside the Qizx distribution contains an executable **qizxstudio.exe** (or **qizxstudio.bat**), that can be started directly by a double-click,
- On Linux or Mac OS X or other Unix, the shell script **bin/qizxstudio** can be started from a terminal or from a graphical file manager.

  Note that when started from a console, Qizx Studio accepts command-line arguments, for example to directly load a XML Query script in the editor. See the reference documentation [44].

You should then see a window looking like this:

Figure 4.1. Qizx Studio first launch



Note

In contrast with Qizx, there is only one tab in Qizx/open Studio: "XQuery" for entering and running queries.

Let us try this query (which is the contents of the file `qizxopen/queries/4.xq`):

```
(: Find all books written by French authors. :)
declare namespace t = "http://www.qizx.com/namespace/Tutorial";

declare variable $authors := collection("../../book_data/Authors/*.xml");
declare variable $books := collection("../../book_data/Books/*.xml");

for $a in $authors//t:author[@nationality = "France"]
   for $b in $books//t:book[.//t:author = $a/t:fullName]
return
    $b/t:title
```

Note that the directory `docs/samples/programming/qizxopen/queries` contains the queries needed to illustrate this lesson.

- Use the menu File → Open XQuery to load the file mentioned above.

  Note that you can also save to a file a query that you have entered or edited in Qizx Studio.

  There is an history that allows running again former queries, so it is not necessary to save intermediary experiments.

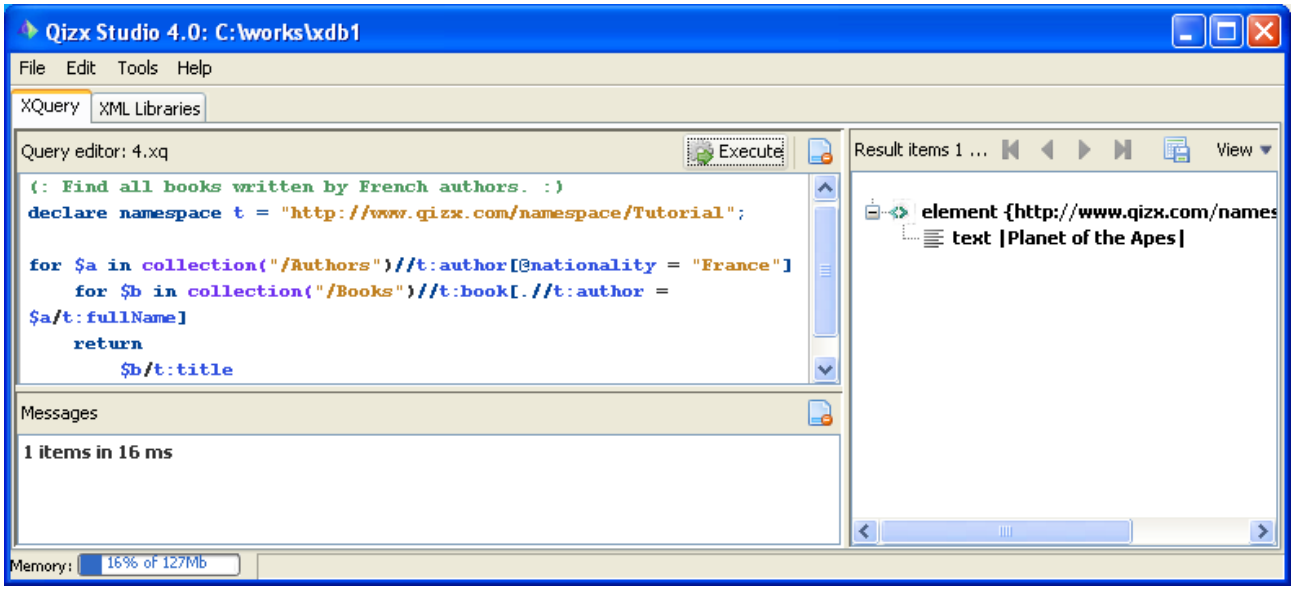
- Then you can use the button Execute to run the query.

Important

The query above uses relative paths for the `collection()` function. Depending how you launched Qizx Studio, you might have to replace the relative paths by absolute paths, otherwise you will get an error like "`empty collection ../../book_data/Authors/*.xml`".

After execution, we should obtain something similar to this:

Figure 4.2. Result of a query



- Notice that, in the picture above, the display mode of the right-side view has been changed to "Data Model", by using the View combo-box. This makes it easier to see the Data Model structure.

  The result sequence contains one item, which is a element `t:title` whose string value is "`Planet of the Apes`".

  If for example we change the value "`France`" to "`US`" in the query, then we get a sequence of 8 items.

  In the same directory there are a few other queries that you can also try.

- The result items in the right-side view can be exported into a file using a button in the header. Notice that the resulting file will not in general be a well-formed XML document.

- The Diagnostic view at bottom left contains messages, which can be simple information (execution times) or possible execution errors.

  Compilation and execution errors have generally a link to the location in the source code. By clicking the link, the cursor moves to the location of the error in the editor view.

- For more information about the editor and the query history, please see the documentation of Qizx Studio [44].

2. Running queries with the qizx command line tool

The shell script `qizx` (`qizx.bat` on Windows) is also located in the `bin/` directory in the Qizx distribution. In the following we assume that this `bin/` directory is in the `PATH` environment variable.

In a terminal window, type the following command (the current directory is assumed to be `docs/samples/programming/qizxopen`):

```
qizx queries/4.xq
```

Results are displayed on the console (or standard output) by default. The option `-out` specifies an output file. Serialization options can be used to specify the output format.

The details of option switches can be found in the tool reference qizx(1) [37].

Part III. Developer's Guide

Chapter 5. Programming with the Qizx/open API

The API in Qizx/open is much simpler than in Qizx because it has not to deal with the management of XML libraries. This section is a simplified version of the tutorial found in Chapter 5, *Programming with the Qizx/open API* [11].

1. Compiling and running the code samples

The code samples used to illustrate this chapter (class `OpenQuery.java`) are found in the `docs/samples/programming/qizxopen` directory. Files containing XQuery scripts are found in the `docs/samples/programming/qizxopen/queries` sub-directory. These scripts are almost the same as those used in the Qizx tutorial, except that access to documents is performed through file paths instead of locations within an XML Library.

You'll need a recent version of ant, a Java-based build tool to compile and run the codes samples.

2. Creating a work session

To create a XML Query session, we need a factory, which is an instance of the class `XQuerySessionManager`. This class manages documents and XML Query modules, so it is recommended to use a single instance from which all sessions are created. The argument of the constructor is an URL used to locate XQuery modules: here it points to the current directory, but we could also use an HTTP URL pointing to a network server.

```
File currentDir = new File(System.getProperty("user.dir"));
XQuerySessionManager sm = new XQuerySessionManager(currentDir.toURL());

XQuerySession session = sm.createSession();
```

Tuning the document cache:

If you have to handle large documents or many documents in a Qizx/open application, it can be useful to tune the size of the document cache. This cache keeps the last documents parsed, so it avoids reloading documents in different sessions. However the cache detects a modification on a document in a file and reloads it.

Reminder: in Qizx/open, documents are always parsed into memory before processing. The functions that load documents are `doc()` [xv] and `collection()` [xv]. They are documented in the last section of this appendix: Differences with Qizx database engine [xv].

Use `XQuerySessionManager.setTransientDocumentCacheSize(int size)` to specify a size in bytes for this cache. You can also use the system property `com.qizx.docpool.maxsize` (For example you would specify `-Dcom.qizx.docpool.maxsize=100000000` on the command line).

3. Compiling and executing queries

Compiling and running a XML Query script is fairly easy:

```
Expression expr = session.compileExpression(script);1
ItemSequence results = expr.evaluate();2
while (results.moveToNextItem()) {3
    Item result = results.getCurrentItem();

    /*Do something with result.*/
}
```

1 First compile an XQuery expression using `XQuerySession.compileExpression`. The argument *`script`* contains a XQuery query expression as a string. If no compilation errors (`CompilationException`) are found, this returns an `Expression` object.

2 Then evaluate the expression using `Expression.evaluate`. If no evaluation errors (`EvaluationException`) are found, this returns the results of the evaluation in the form of an `ItemSequence`.

3 An `ItemSequence` allows to iterate over a sequence of `Items` (see ???). A `Item` is either an atomic value or an XML `Node`.

Example (`1.xq`):

```
(: Compute and return 2 + 3 :)
2 + 3
```

evaluates to an `ItemSequence` containing a single atomic value (5).

Example (`3.xq`):

```
(: List all books by their titles. :)
declare namespace t = "http://www.qizx.com/namespace/Tutorial";

collection("../../book_data/Books/*.xml")//t:book/t:title
```

evaluates to an `ItemSequence` containing several `t:title` element `Nodes`.

The `OpenQuery` class implements a simple command-line tool allowing to run queries.

Excerpts of `OpenQuery.java`:

```
    private static Expression compileExpression(XQuerySession session,
                                                String script,
                                                QName[] varNames,
                                                String[] varValues)
        throws IOException, QizxException
    {
        Expression expr;
        try {
            expr = session.compileExpression(script);
        }
        catch (CompilationException e) {
            Message[] messages = e.getMessages();
            for (int i = 0; i < messages.length; ++i) {
                error(messages[i].toString());
            }
            throw e;
        }

        if (varNames != null) {
            for (int i = 0; i < varNames.length; ++i) {
                expr.bindVariable(varNames[i], varValues[i], /*type*/ null);1
            }
        }

        return expr;
    }
```

1 An XQuery expression can be further parametrized by the use of variables. Example (`101.xq`):

```
(: List all books containing the value of variable $searched
   in their titles. :)
declare namespace t = "http://www.qizx.com/namespace/Tutorial";

declare variable $searched external;

collection("/Books")//t:book/t:title[contains(., $searched)]
```

`Expression.bindVariable` allows to give a variable its value, prior to evaluating the expression.

Some queries may return thousands of results. Therefore, displaying just a range of results (e.g from result #100 to result #199 inclusive) is a very common need.

```
    private static void evaluateExpression(Expression expr,
                                           int from, int limit)
        throws QizxException {
        ItemSequence results = expr.evaluate();
        if (from > 0) {
            results.skip(from);1
        }

        XMLSerializer serializer = new XMLSerializer();
        serializer.setIndent(2);

        int count = 0;
        while (results.moveToNextItem()) {
            Item result = results.getCurrentItem();

            System.out.print("[" + (from+1+count) + "] ");
            showResult(serializer, result);
            System.out.println();

            ++count;
            if (count >= limit)2
                break;
        }
        System.out.flush();
    }
```

1 `ItemSequence.skip` allows to quickly skip the specified number of `Items`.
2 This being done, you still need to limit the number of `Items` you are going to display.

In this lesson, we'll just show how to print the string representation of an `Item`.

```
    private static void showResult(XMLSerializer serializer,
                                   Item result)
        throws QizxException {
        if (!result.isNode()) {1
            System.out.println(result.getString());2
            return;
        }
        Node node = result.getNode();3

        serializer.reset();
        String xmlForm = serializer.serializeToString(node);4
        System.out.println(xmlForm);
    }
```

1 3 `Item.isNode` returns `true` for a `Node` and `false` for an atomic value. Similarly, `Item.getNode` returns a `Node` when the `Item` actually is a Node and `null` when the `Item` is an atomic value.
2 `Item.getString` returns the *string value* of an `Item` (whether `Node` or atomic value). What precisely is the string value of an `Item` is specified in the XQuery standard.
4 The `XMLSerializer.serializeToString` convenience method is used to obtain the string representation of a `Node`.

3.1. Compiling and running the code of this lesson

- Compile class `Query` by executing **ant** (see `build.xml`) in the `docs/samples/programming/query/` directory.
- Run **ant run1** in the `docs/samples/programming/qizxopen/` directory to perform this query:

```
(: Find all books written by French authors. :)
declare namespace t = "http://www.qizx.com/namespace/Tutorial";

for $a in collection("/Authors")//t:author[@nationality = "France"]
    for $b in collection("/Books")//t:book[.//t:author = $a/t:fullName]
    return
        $b/t:title
```

You can execute all the queries by running **ant run_all** in `docs/samples/programming/qizxopen/`.

Part IV. Reference

Differences with Qizx database engine

1. Java API

In Qizx/open, the following elements are absent:

- in package com.qizx.api: all classes whose name begins with '`Library`': `Library`, `LibraryMember`, etc,
- in package com.qizx.api: interface `AccessControl` and `User`, class `AccessControlException`,
- package com.qizx.api.util.accesscontrol .

2. Function fn:doc()

```
fn:doc ($path-or-URL as xs:string)
   as node()
```

The standard `doc()` function of XQuery.

Parameter *`$path-or-URL`*: This argument can be a simple file path or any URL supported by the Java runtime.

Returned value: a *document node*.

Example:

```
doc("../book_data/Authors/iasimov.xml")
doc("http://www.axyana.com/qizx_tests/doc.cml")
```

3. Function fn:collection()

```
fn:collection ($path as xs:string)
   as node()*
```

This is the standard `collection()` function of XQuery, slightly extended.

Parameter *`$path`*: This argument is a list of documents paths, separated by commas or semicolons.

- A normal path (without wildcard characters) is treated as per the function `fn:doc()`. So it can either be part of a XML Library, or be an external document (file or URL) parsed on the fly.
- If a path contains the wildcard characters `*` or `?`, it is treated as a file pattern and expanded. Attention: wildcard characters are currently accepted only in the file name, not in the path of the parent directory.

  For example `collection("/home/data1/*.xml;/home/data2/*.xml")` can be expanded, while `collection("/home/*/*.xml;")` currently cannot be expanded (generates an error).
- All of the documents must be accessible, or an error is raised.

  If the expanded sequence of documents is empty, an error is raised.

Returned value: a sequence of *document nodes*.

Example:

```
collection("../book_data/Authors/*.xml;../book_data/Author blurbs/*.xhtml")
```

Chapter 6. General XQuery extension functions

These general purpose functions belong to the namespace denoted by the predefined "`x:`" prefix. The `x:` prefix refers to namespace "`com.qizx.functions.ext`".

1. Serialization

Serialization — the process of converting XML nodes into a stream of characters — is defined in the W3C specifications, however there is no standard function for performing serialization.

`x:serialize` can output a document or a node into XML, HTML, XHTML or plain text, to a file or to the default output stream.

```
x:serialize( $node as node(), $options as element(option) )
  as xs:string?
```

Description: Serializes the element and all its content into text. The output can be a file (see options below).

Parameter *`$tree`*: a XML tree to be serialized to text.

Parameter *`$options`*: an element bearing options in the form of attributes: see below.

Returned value: The path of the output file if specified, otherwise the serialized result.

The options argument (which may be absent) has the form of an element of name "options" whose attributes are used to specify different options. For example:

```
x:serialize( $doc,
             <options output="out\doc.xml"
                      encoding="ISO-8859-1" indent="yes"/>)
```

This mechanism is similar to XSLT's xsl:output specification and is very convenient since the options can be computed or extracted from a XML document.

Table 6.1. Implemented serialization options

option name	values	description
method	XML (default) XHTML, HTML, or TEXT	output method
output / file	a file path	output file. If this option is not specified, the generated text is returned as a string.
version	default "1.0"	version generated in the XML declaration. No validity check.
standalone	"yes" or "no".	No check is performed.
encoding	must be the name of an encoding supported by the JRE.	The name supplied is generated in the XML declaration. If different than UTF-8, it forces the output of the XML declaration.
indent	"yes" or "no" (default "no").	output indented.
indent-value *(extension)*	integer value	specifies the number of space characters used for indentation.
omit-xml-declaration	"yes" or "no" (default "no").	controls the output of a XML declaration.
include-content-type	"yes" or "no" (default "no").	for XHTML and HTML methods, if the value is "yes", a META element specifying the content type is added at the beginning of element HEAD.
escape-uri-attributes	"yes" or "no" (default "yes").	for XHTML and HTML methods, escapes *URI attributes* (i.e specific HTML attributes whose value is an URI).
doctype-public	the public ID in the DOCTYPE declaration.	Triggers the output of the DOCTYPE declaration. Must be used together with the `doctype-system` option.
doctype-system	the system ID in the DOCTYPE declaration.	Triggers the output of the DOCTYPE declaration.
auto-dtd *(extension)*	"yes" or "no" (default "yes").	If the node is a document node and if this document has DTD information, then output a DOCTYPE declaration. • A Document stored in an XML Library may have properties storing this information (dtd-system-id and dtd-public-id) initially set by import. • a parsed document gets DTD information from the XML parser. • a constructed node has no DTD information.

2. XSL Transformation

The `x:transform` function invokes a XSLT style-sheet on a node and can retrieve the results of the transformation as a tree, or let the style-sheet output the results.

This is a useful feature when one wants to transform a document (for example extracted from the XML Libraries) or a computed fragment of XML into different output formats like HTML, XSL-FO etc.

This example generates the transformed document `$doc` into a file `out\doc.xml`:

```
x:transform( $doc, "ssheet1.xsl",
             <parameters param1="one" param2="two"/>,
             <options output-file="out\doc.xml" indent="yes"/>)
```

The next example returns a new document tree. Suppose we have this very simple stylesheet which renames the element "doc" into "newdoc":

```
<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
                version ="1.0" >
  <xsl:template match="doc">
     <newdoc><xsl:apply-templates/></newdoc>
  </xsl:template>
</xsl:stylesheet>
```

The following XQuery expression:

```
x:transform( <doc>text</doc>, "ssheet1.xsl", <parameters/> )
```

returns:

```
<newdoc>text</newdoc>
```

```
x:transform( $source as node(),
             $stylesheet-URI as xs:string,
             $xslt-parameters as element(parameters)
             [, $options as element(options)] )
  as node()?
```

Transforms the source tree through a XSLT stylesheet. If no output file is explicitly specified in the options, the function returns a new tree.

Parameter *`$source`*: a XML tree to be transformed. It does not need to be a complete document.

Parameter *`$stylesheet-URI`*: the URI of a XSLT stylesheet. Stylesheets are cached and reused for consecutive transformations.

Parameter *`$xslt-parameters`*: an element holding parameter values to pass to the XSLT engine. The parameters are specified in the form of attributes. The name of an attribute matches the name of a `xsl:param` declaration in the stylesheet (namespaces can be used). The value of the attribute is passed to the XSLT transformer.

Parameter *`$options`*: [optional argument] an element holding options in the form of attributes: see below.

Returned value: if the path of an output file is not specified in the options, the function returns a new document tree which is the result of the transformation of the source tree. Otherwise, it returns the empty sequence.

Table 6.2. XSLT transform options

option name	values	description
output-file	An absolute file path.	Output file. If this option is not specified, the generated tree is returned by the function, otherwise the function returns an empty sequence.
XSLT output properties (instruction xsl:output): version, standalone, encoding, indent, omit-xml-declaration etc.		These options are used by the style-sheet for outputting the transformed document. They are ignored if no output-file option is specified.
Specific options of the XSLT engine (Saxon or default XSLT engine)		An invalid option may cause an error.

About the efficiency of the connection with XSLT

The connection with an XSLT engine uses generic JAXP interfaces, and thus must copy XML trees passed in both directions. This is not as efficient as it could be and can even cause memory problems if the size of processed documents is larger then a few dozen megabytes, depending on the available memory size.

3. Dynamic evaluation

The following functions allow dynamically compiling and executing XQuery expressions.

```
function x:eval( $expression as xs:string )
  as xs:any
```

Compiles and evaluates a simple expression provided as a string.

The expression is executed in the context of the current query: it can use global variables, functions and namespaces of the current static context. It can also use the current item '.' if defined in the evaluation context.

However there is no access to the local context (for example if `x:eval` is invoked inside a function, the arguments or the local variables of the function are not visible.)

Parameter *`$expression`*: a simple expression (cannot contain prologue declarations).

Returned value: evaluated value of the expression.

Example:

```
declare variable $x := 1;
declare function local:fun($p as xs:integer) { $p * 2 };

let $expr := "1 + $x, local:fun(3)"
return x:eval($expr)
```

This should return the sequence `(2, 6)`.

4. Pattern-matching

The following functions match the string-value of nodes (elements and attributes) with a pattern.

Example 1: this expression returns true if the value of the attribute @lang matches the SQL-style pattern:

```
x:like( "en%", $node/@lang )
```

Example 2: this expression returns true if the content of the element 'NAME' matches the pattern:

```
$p/NAME[ x:like( "Theo%" ) ]
```

```
function x:like( $pattern as xs:string [, $context-nodes as node()* ])
  as xs:boolean
```

Returns true if the pattern matches the string-value of at least one node in the node sequence argument.

Parameter *`$pattern`*: a SQL-style pattern: the wildcard '_' matches any single character, the wildcard '%' matches any sequence of characters.

Parameter *`$context-nodes`*: optional sequence of nodes. The function checks sequentially the string-value of each node against the pattern. If absent, the argument default to '.', the current item. This makes sense inside a predicate, like in the example 2 above.

Returned value: a boolean.

```
function x:ulike( $pattern as xs:string [, $context-nodes as node()* ])
  as xs:boolean
```

This function is very similar to `x:like`, except that the pattern has syntax à la Unix ("glob pattern"). The character '?' is used instead of '_' (single character match), and '*' instead of '%' (multi-character match).

Note: these functions — as well as the standard `fn:matches` function, and the full-text functions — are automatically recognized by the query optimizer which uses library indexes to boost their execution whenever possible.

5. Date and Time

5.1. Differences with W3C specifications

Qizx is compliant with the W3C Recommendation. The only differences at present are extensions of the cast operation: Qizx can directly cast date, time, dateTime and durations to and from double values representing seconds, and keeps the extended "constructors" that build date, dateTime, etc, from numeric components like days, hours, minutes, etc.

5.2. Cast Extensions

In order to make computations easier, Qizx can:

- Cast `xdt:yearMonthDuration` to numeric values: this yields the number of months. The following expression returns 13:

  ```
  xdt:yearMonthDuration("P1Y1M") cast as xs:integer
  ```

- Conversely, cast numeric value representing months to `xdt:yearMonthDuration`. The following expression holds true:

  ```
  xdt:yearMonthDuration(13) = xdt:yearMonthDuration("P1Y1M")
  ```

- Cast `xdt:daytimeDuration` to double: this yields the number of seconds. The following expression returns 7201:

  ```
  xdt:dayTimeDuration("PT2H1S") cast as xs:double
  ```

- Conversely, cast a numeric value representing seconds to `xdt:daytimeDuration`.

- Cast `xs:dateTime` to double. This returns the number of seconds elapsed since ``the Epoch", i.e. 1970-01-01T00:00:00Z. If the timezone is not specified, it is considered to be UTC (GMT).

- Conversely, cast a numeric value representing seconds from the origin to a dateTime with GMT timezone.
- cast from/to the `xs:date` type in a similar way (like a dateTime with time equal to 00:00:00).

```
xdt:date("1970-01-02") cast as xs:double = 86400
```

- cast from/to the `xs:time` type in a similar way (seconds from 00:00:00).

```
xdt:time("01:00:00") cast as xs:double = 3600
```

5.3. Additional constructors

These constructors allow date, time, dateTime objects to be built from numeric components (this is quite useful in practice).

```
function xs:date( $year as xs:integer,
                  $month as xs:integer,
                  $day as xs:integer )
  as xs:date
```

Builds a `xs:date` from a year, a month, and a day in integer form. The implicit timezone is used.

For example `xs:date(1999, 12, 31)` returns the same value as `xs:date("1999-12-31")`.

```
function xs:time( $hour as xs:integer,
                  $minute as xs:integer,
                  $second as xs:double )
  as xs:time
```

Builds a `xs:time` from an hour, a minute as integer, and seconds as double. The implicit timezone is used.

```
function xs:dateTime( $year as xs:integer, $month as xs:integer, $day as xs:integer,
                      $hour as xs:integer, $minute as xs:integer, $second as xs:double
                      [, $timezone as xs:double] )
  as xs:dateTime
```

Builds a `xs:dateTime` from the six components that constitute date and time.

A timezone can be specified: it is expressed as a signed number of hours (ranging from -14 to 14), otherwise the implicit timezone is used.

5.4. Additional accessors

These functions are kept for compatibility. They are slightly different than the standard functions:

- they accept several date/time and durations types for the argument (so for example we have get-minutes instead of get-minutes-from-time, get-minutes-from-dateTime etc.),
- but they do not accept untypedAtomic (node contents): such an argument should be cast to the proper type before being used. So the standard function might be as convenient here.

```
function get-seconds( $moment )
  as xs:double?
```

Returns the "second" component from a xs:time, xs:dateTime, and xs:duration.

Can replace fn:seconds-from-dateTime, fn:seconds-from-time, fn:seconds-from-duration, except that the returned type is double instead of decimal, and an argument of type xdt:untypedAtomic is not valid.

```
function get-all-seconds( $duration )
  as xs:double?
```

Returns the total number of seconds from a xs:duration. This does not take into account months and years, as explained above.

For example `get-all-seconds(xs:duration("P1YT1H"))` returns 3600.

```
function get-minutes( $moment )
  as xs:integer?
```

Returns the "minute" component from a `xs:time`, `xs:dateTime`, and `xs:duration`.

```
function get-hours( $moment )
  as xs:integer?
```

Returns the "hour" component from a `xs:time`, `xs:dateTime`, and `xs:duration`.

```
function get-days( $moment )
  as xs:integer?
```

Returns the "day" component from a `xs:date`, `xs:dateTime`, `xs:day`, `xs:monthDay` and `xs:duration`.

```
function get-months( $moment )
  as xs:integer?
```

Returns the "month" component from a `xs:date`, `xs:dateTime`, `xs:yearMonth`, `xs:month`, `xs:monthDay` and `xs:duration`.

```
function get-years( $moment )
  as xs:integer?
```

Returns the "year" component from a `xs:date`, `xs:dateTime`, `xs:year`, `xs:yearMonth` and `xs:duration`.

```
function get-timezone( $moment )
  as xs:duration?
```

Returns the "timezone" component from any date/time type and `xs:duration`.

The returned value is like `timezone-from-*` except that the returned type is `xs:duration`, not `xdt:dayTimeDuration`.

6. Error handling

XQuery has currently no mechanism to handle run-time errors.

Actually the language is such that an error handling is not absolutely mandatory: many errors need not be recovered (for example type errors); the `doc()` function which, can generate a dynamic error, is now protected by a new function `doc-available()`.

However, extensions (namely the Java binding mechanism) can generate errors. It is not possible to provide a protection auxiliary like `doc-available()` for every functionality.

Qizx provides a try/catch construct, which is a syntax extension. This construct has several purposes.

```
try { expr } catch($error) { fallback-expr }
```

The try/catch extended language construct first evaluates the body *expr*. If no error occurs, then the result of the try/catch is the return value of this expression.

If an error occurs, the local variable *$error* receives a string value which is the error message, and *fallback-expr* is evaluated (with possible access to the error message). The resulting value of the try/catch is in this case the value of this fallback expression. An error in the evaluation of the fallback-expression is not caught.

The type of this expression is the type that encompasses the types of both arguments.

Important

The body (first expression) is guaranteed to be evaluated completely before exiting the try/catch - unless an error occurs. In other terms, lazy evaluation, which is used in most Qizx expressions, does not apply here.

This is specially important when functions with side-effects are called in the body. If such functions generate errors, these errors are caught by the try/catch, as one can expect. Otherwise lazy evaluation could produce strange effects.

Example: tries to open a document, returns an element `error` with an attribute `msg` containing the error message if the document cannot be opened.

```
try {
    doc("unreachable.xml")
}
catch($err) {
    <error msg="{$err}"/>
}
```

7. Miscellaneous

```
function x:parse($xml-text)
  as node()?
```

Parses a string representing an XML document and returns a node built from that parsing. This can be useful for converting to a node a string from any origin.

Note that function x:eval could be used too (and it is more powerful, since any kind of node can be built with it), but there are some syntax differences: for example in x:eval, the curly braces `{` and `}` have to be escaped by duplicating them.

Parameter *$xml-text*: A well-formed XML document as a string.

Returned value: A node of the Data Model if the string could be correctly parsed; the empty sequence if the argument was the empty sequence. An error is raised if there is a parsing error.

```
function x:in-range( $value, $low-bound as item(), $high-bound as item() )
  as xs:boolean

function x:in-range( $value, $low-bound as item(), $high-bound as item(),
                     $low-included as xs:boolean,
                     $high-included as xs:boolean )
  as xs:boolean
```

Returns true if at least one item from the sequence *$value* belongs to the range defined by other parameters.

This function is used typically to optimize a predicate in a Library query, for example `//object[ x:in-range(@weight, 1, 10) ]` which is equivalent to `//object[@weight >= 1 and @weight <= 10]`.

The reason for this function is that the query optimizer is not able to detect such a double test in all situations. The function could become useless in later versions of Qizx, after improvement of the query optimizer.

Parameter *$value*: Any sequence of items. Items must be comparable to the bounds, otherwise a type error is raised.

Parameters *$low-bound*, *$high-bound*: Lower and upper bounds of the range. They must be of compatible types.

Parameters *$low-included*: If *$low-included* is equal to `true()`, the comparison used is `$low-bound <= $value`, otherwise `$low-bound < $value`. If absent, <= is assumed.

Parameters *$high-included*: If *$high-included* is equal to `true()`, the comparison used is `$value <= $high-bound`, otherwise `$value < $high-bound`. If absent, <= is assumed.

Returned value: True if at least one item from the sequence *$value* belongs to the range defined by *$low-bound*, *$high-bound*.

Chapter 7. Full-text XQuery extension functions

Starting from version 3.0, Qizx implements the standard XQuery Full-Text from the W3C (abbreviated XQFT hereafter).

Please see chapter ??? for more information about standard full-text support. That chapter contains a section explaining how to migrate your Qizx 2.2 applications from the former full-text functionalities.

This current chapter introduces new full-text extension functions from version 3.1:

- **A simplified search function** that uses a simpler and more usual query syntax than the XQuery Full-Text standard.

  Note: it is actually similar to the former full-text function (in Qizx 2.2 and before), but beware that the syntax is somewhat different.

- **Utility functions** for highlighting full-text terms, generating summary snippets, looking up indexes and finding spell-checking suggestions.

1. Simplified full-text search

The justification for a simplified full-text search facility is the following:

- A standard XQFT query is not an object than can be manipulated by an XQuery script. This makes it more difficult for an XQuery application to synthesize a full-text query and then execute it, unless one resorts to a dynamic evaluation function like Qizx x:eval() [?].

- The standard XQuery Full-Text from the W3C is not yet a completely stable specification (in July 2009, it reached the stage of *Candidate Recommendation*, and it can take up to one year before it becomes a definitive standard).

- The standard W3C full-text syntax is a bit complex and unusual, even for advanced users (those users who would otherwise have no difficulty with a query like: `title:product +"beta quality" -alpha`).

1.1. Definition of the simple full-text syntax

This syntax is very simple and resembles the one found in most full-text engines. Notice that there is no notion of Fields, since XQuery itself provides all the means of searching specific parts of XML documents.

Search Capability	Examples	Remarks
Simple word (without quotes)	Hello	Tokenized according to the language and configuration. Note than a composed word like *never-ending* can actually be tokenized into 2 words, equivalent to phrase "never ending".
Wildcard	?ello *ell*	Can be used in place of a simple word inside a phrase.
Phrase (single or double quotes)	"Hello world" 'Hello, world!'	Tokenized according to the language and configuration.
Phrase with proximity	"hello world"~3	Same meaning as in "window 3 words" of the standard syntax:

Search Capability	Examples	Remarks
		matches "hello new world", but not "hello brand new world".
Required term	+world +'Hello world'	Acts like a ftand, while plain terms act like a ftor.
Negated term	-hello -"old world"	Such terms must not be found in the searched document or fragment.
Juxtaposition	hello "brave new world" +me -you	Terms without + are ORed. Terms with + are ANDed. The example on the left is equivalent to: `"me" ftand ("hello" ftor "brave new world") ftand ftnot "you"`

1.2. Search function

```
function ft:contains ($query, [$options])
```

```
function ft:contains ($query, $context, $options)
```

returns true if the search context matches the full-text query.
Note: this function is similar to the former ft:contains function of Qizx up to version 2.2, but beware that the query syntax is not quite the same.

This function is typically used as a *predicate* in a *Path Expression*. Examples:

```
//SPEECH[ ft:contains("+romeo +juliet") ],
//SPEECH[ ft:contains(" 'to be or not to be' ", LINE, <options/>) ]
```

Returned value: true if the context matches the query, false otherwise.

Parameter $query: A query using the simple full-text syntax.

Parameter $context (optional): A node, or sequence of nodes, inside which the full-text expression is searched for. Note: this is the equivalent of a Field in classical full-text engines.

When *`context`* parameter is not specified, the current context node '.' is used implicitly like in the example above. Note that when the function is called with 2 arguments, the last argument represents the options, not the context.

When *`context`* parameter is present, it specifies a smaller search domain (in general inside to the current context node) . The 2nd example above finds SPEECH elements which contain at least one LINE element which in turn contains the phrase 'to be or not to be'.

Parameter $options (optional): An element (conventionally named "options") bearing attributes:

- attribute `case`: value is "sensitive" or "insensitive" (using only first characters, e.g "sens", is allowed)
- attribute `diacritics`: value is "sensitive" or "insensitive"
- attribute `language`: value is a legal language name, used for tokenizing words and phrases, and stemming. This option must precede stemming and thesaurus options if used (see below).
- attribute `stemming`: value is a boolean "true" or "false". Assumes that the application provides a Stemmer implementation (see the Java API documentation).

- attribute `thesaurus`: value is a thesaurus URI. Assumes that the application provides a Thesaurus implementation (see the API documentation).

Example:

```
<options language="fr" diacritics="sensitive"/>
```

2. Other full-text extension functions

```
function ft:score ($sequence, [$length], [$start])
```

returns the sequence sorted by decreasing full-text score. Optionally, the result sequence can be 'sliced' in pages by specifying the first element and the length of a page.

The input sequence is typically a full-text search expression using either `ft:contains()` or the standard operator '`contains text`'.

The purpose of this function is to simplify the use of scoring, but also to make it more efficient than the `` `for score ... order by $score descending` '' pattern of XQFT standard. Further versions of Qizx could enhance this function to make it even more efficient by allowing fast heuristic scoring strategies.

When $length and $start are used, this function is an optimized equivalent of:

```
fn:subsequence( for $hit score $score in $sequence
                  order by $score descending
                  return $hit,
                $start, $length )
```

Example:

```
ft:score( //SPEECH[ ft:contains("hello +world") ], 10 )
```

Returned value: The input sequence ordered by descending score, possibly sliced.

Parameter *$sequence*: A query using the simple full-text syntax (function ft:contains), or the standard `'contains text'` operator.

Parameter $length (optional): Number of results to be returned. Used for slicing results. If not specified, the value is 10.

Parameter $start (optional): rank of the first hit to be returned. Used for slicing results.

```
function ft:highlight ($node, $query, [$options]) as node()
```

Transforms an XML fragment (document or node) by replacing each occurrence of the words of a full-text query by a XML template that contains the word. This is called highlighting because typically it can be used with a formatting language (HTML) to render the word with some styling, using for example CSS.

Words within a ftnot clause are not highlighted.

Word occurrences are highlighted individually. For example if the query specifies a phrase, all occurrences of the words of this phrase will be highlighted, whether they belong to an occurrence of the phrase or not.

Example:

```
let $doc := <P>this is some text searched by a query.</P>
return ft:highlight( $doc, "query text", <options word-wrap="B"/> )
```

returns:

```
<P>this is some <B>text</B> searched by a <B>query</B>.</P>
```

Returned value: A copy of the node in which all occurrences of the full-text query words are replaced by the specified pattern.

Parameter *$node*: an XML fragment (document or node) to be highlighted.

Parameter *$query*: An expression which is either of:

- The operator `contains text`. Example:

```
ft:highlight($node, . contains text "hello world" any word)
```

  Note: the expression must be exactly '`contains text`', a boolean combination is not allowed. The context part (here '.') is ignored. Full-text options following `contains text` are taken into account.

- the function ft:contains(). The optional *`context`* argument is ignored. Full-text options are taken into account.

```
ft:highlight($node, ft:contains(" 'hello world' "))
```

  Note: in this example, although the query requires a phrase, all individual occurrences of the words 'hello' and 'world' will be highlighted, not the phrase only.

- a string (using the simple full-text syntax). In that case it is not possible to specify options.

```
ft:highlight($node, "hello world")
```

Parameter $options (optional): An element (conventionally named "options") with attributes containing the options. There are two ways of specifying how a word is "highlighted":

The first way uses a simple element bearing an attribute, similar to the SPAN element of HTML with a class attribute:

- attribute *`word-wrap`*: its value is the name of an element used to wrap the word. Default is "B".
- optional attribute *`word-style`*: value is the name of an attribute placed on the word-wrapper element. It is not present by default.
- optional attribute *`word-pattern`*: value is a pattern that is used to give a value to attribute *`word-style`*. If it contains the character %, this character is replaced by the rank of the word in the query.

  Example:

```
let $doc := <P>this is some text searched by a query.</P>
return ft:highlight( $doc, "xquery +text",
               <options word-wrap="SPAN" word-style="class"
                        word-pattern="hilite%"/> )
```

  produces:

```
<P>this is some <SPAN class="hilite1">text</SPAN
 > returned by a <SPAN class="hilite0">XQuery</SPAN> expression.</P>
```

The second way uses a function called by name (XQuery cannot pass a function as a parameter of another function):

- attribute *`word-function`*: value is the name of a function that is called for each occurrence of a word to highlight. The value returned must be a Node which replaces the word.

  The called function must be compatible with this signature:

```
 function($word as xs:string, $word-rank as xs:int, $node as text()):
```

  - $word receives a string which receives the value of the word

- $word-rank is an integer which receives the rank of the word in the query.
- $node is the text node that contains the word. This allows to test arbitrarily complex conditions.

Example that highlights a word with bold if it is inside a TITLE, otherwise with a span/class:

```
declare function local:hilite($word, $word-rank, $node) {
  if($node/parent::TITLE)
  then <B>{$word}</B>
  else <span class="hilite{$word-rank}">{$word}</span>
}

let $doc := <P>this is some text searched by a query.</P>
return ft:highlight( $doc, . contains text "query text" all words,
            <options word-function="local:hilite"/> )
```

```
function ft:snippet ($node, $query, [$options]) as element()
```

Extracts a representative snippet from a document. words from a full-text query are "highlighted" in the same way as the ft:highlight [27] function. This allows getting a result similar to the snippets produced by most major web search engines.

A snippet is an element that contains text fragments and highlighted words.

Example:

```
for $doc in //SPEECH[ ft:contains("hello +world") ]
return ft:snippet($doc)
```

Returned value: An element node containing the snippet.

Parameter *$node*: an XML document or node to be represented.

Parameter *$query*: A string (simple syntax query) or an expression using contains text, for example `. contains text "hello world"`.

Parameter $options (optional): An element (conventionally named "options") with attributes.

Options similar to ft:highlight [27]:

- attribute *`word-wrap`*: its value is the name of an element used to wrap the word. Default is "B".
- optional attribute *`word-style`*: value is the name of an attribute placed on the word-wrapper element. It is not present by default.
- optional attribute *`word-pattern`*: value is a pattern that is used to give a value to attribute *`word-style`*. If it contains the character %, this character is replaced by the rank of the word in the query.
- attribute *`word-function`*: value is the name of a function that is called for each occurrence of a word to highlight. The value returned must be a Node which replaces the word.

  The called function must be compatible with this signature: function($word as xs:string, $word-rank as xs:int, $node as text()):

Specific options:

- attribute *`snippet`*: its value is the name of an element used to wrap the snippet. Default is "snippet".
- optional attribute *`length`*: the maximum number of words in the snippet. Default value is 20.
- optional attribute *`work-size`*: the maximum number of words from start examined to find the best parts of the document. Default value is 500.

```
function ft:word-count($word as xs:string) as xs:integer?
```

returns the total count of occurrences of this word in the current XML Library.

Example:

```
ft:word-count("hamlet") (: counts occurrence of Hamlet, HAMLET etc. :)
```

Parameter *$word*: A string containing a single word. Character case and diacritics are not taken into account.

Returned value: An positive integer item, or the null sequence if the word is not found, or if not connected to an XML Library.

```
function ft:word-doc-count($word as xs:string) as xs:integer?
```

returns the total count of documents in the current XML Library that contain at least one occurrence of this word.

Parameter *$word*: A string containing a single word. Character case and diacritics are not taken into account.

Returned value: An positive integer item, or the null sequence if the word is not found.

```
function ft:word-lookup([$word-pattern as xs:string?]) as xs:string*
```

returns a list of words indexed in the current XML Library that match the pattern. If no pattern is passed, then all the words indexed in the Library are returned.

Attention: words are sorted ignoring character case and diacritics, and the different forms in which a word occurs are not returned. For example `ft:word-lookup("cafe")` does not return a sequence `like ("CAFE", "CAFÉ", "Cafe", "Café", "cafe", "café")` even if these forms occur in the XML Library. This situation is likely to change in later versions, which will optimize case-sensitive and diacritics-sensitive searches, but that will require to change the representation of indexes.

Parameter *$word-pattern*: A string containing a wildcard pattern (standard syntax, case and diacritics insensitive). If absent, then all the words indexed in the Library are listed.

Returned value: A sorted list of strings, or the null sequence if the word is not found. Sorting is done ignoring character case and diacritics.

```
function ft:suggest($word as xs:string) as xs:string*
```

returns a list of words that are "close to" the specified word, sorted by increasing distance. The distance used is a simple Levenshtein algorithm, where differences in case or diacritics have a lesser weight than deletion or insertions. The function also tries space insertion (e.g "myword" can yield "my word").

Note: this function is not a spell-checking facility, it can only return words that actually appear in a document of the Library.

Parameter *$word*: A string containing a single word. Character case and diacritics are taken into account for distance calculation.

Returned value: A string sequence containing at most 20 suggestions. Best effort is done for returning at least one suggestion.

3. Examples

This section is a short tutorial showing how to use Qizx full-text functionalities.

Query a collection of documents:

The most classical way of doing full-text queries is to look for whole documents matching a full-text expression anywhere in their contents. For example, using standard XQuery Full-Text:

```
/*[ . contains text "printing press" ] (: uses implicit collection :)
```

or the same using the simplified syntax:

```
/*[ ft:contains(" 'printing press' ") ]  (: notice the quotes :)
```

The 2 examples above return a sequence of the root elements of the matching documents. If you want to retrieve the Document objects themselves, use xlib:document():

```
for $doc in /*[ . contains text "printing press" ]
 return xlib:document($doc)
```

"Advanced Search" a la Google™:

The Advanced Search by Google offers the possibility to search for pages that match "all these words", "this exact wording or phrase", "one or more of these words", but not pages that have "any of these unwanted words" (words are specified in form fields).

This is easy to implement with XQFT and Qizx, assuming that you have the field values in 4 variables named $all, $exact, $any, $unwanted:

```
/*[ . contains text
      { $all } all words ftand
      { $any } any word ftand
      { $exact } phrase ftand
      ftnot { $unwanted } any word
  ]
```

Note that if all fields are empty, no error is detected but no document is returned.

Find best scoring documents:

The function ft:score is designed to make easier to finding best scoring documents and list them in pages. To display the first 10 documents matching a query:

```
ft:score( /*[ . contains text "printing press" ] , 10)
```

To display the following 10 documents by descending score, just increment a variable $start (initialized to 0) by 10 and use it as third argument of

```
ft:score( /*[ . contains text "printing press" ] , 10, $start)
```

Display summary snippets of documents:

Popular web search engines display a short abstract of each document showing highlighted terms of the full-text query. The function ft:snippet allows to do this easily in Qizx:

```
let $query := "printing press"
for $doc in /*[ . contains text { $query } ]
 return ft:snippet($doc, $query)
```

The output of ft:snippet and ft:highlight functions can be controlled finely (see the reference documentation).

Summary: a simple "Advanced Search"

This query finds the 10 best matching documents, and for each document returns a snippet where the query terms are in bold:

```
for $doc in
   ft:score(/*[ . contains text
                { $all } all words ftand
                { $any } any word ftand
                { $exact } phrase ftand
                ftnot { $unwanted } any word ], 10)
return
  <div><h4>{ xlib:document($doc) }</h4>
       { ft:snippet($doc,
                    . contains text { $all } all words ftand
                                    { $any } any word ftand
                                    { $exact } phrase,
                    <options word-wrap="b"/>)
       }</div>
```

Chapter 8. Java™ Binding

The *Java binding* feature is a powerful extensibility mechanism which allows direct calling of Java methods bound as XQuery functions and manipulation of wrapped Java objects.

Java Binding opens a tremendous range of possibilities since nearly all the Java APIs become accessible. The implementation performs many automatic conversions, including Java arrays and some Java collections.

The Java binding mechanism is widely used in several XQuery extension modules such as *XML Library handling functions* and *SQL Connectivity*.

Qizx Java Binding is similar to the mechanism introduced by several other XQuery or XSLT engines like XT or Saxon: a qualified function name where the namespace URI starts with "`java:`" is automatically treated as a call to a Java method.

- The namespace URI must be of the form `java:`*`fullyQualifiedClassName`*. The designated class will be searched for a method matching the name and arguments of the XQuery function call.
- The XQuery name of the function is modified as follows: hyphens are removed while the character following an hyphen is upper-cased (producing 'camelCasing'). So "`get-instance`" becomes "`getInstance`".

In the following example the `getInstance()` method of the class `java.util.Calendar` is called:

```
declare namespace cal = "java:java.util.Calendar"
cal:get-instance()    (:  or cal:getInstance()  :)
```

The mechanism is actually a bit more flexible: a namespace can also refer to a package instead of a class name. The class name is passed as a prefix of the function name, separated by a dot. For example:

```
declare namespace util = "java:java.util"
util:Calendar.get-instance()
```

The following example invokes a constructor, gets a wrapped File in variable `$f`, then invokes the non-static method `mkdir()`:

```
declare namespace file = "java:java.io.File"

let $f := file:new("mynewdir")
return file:mkdir($f)
```

In this example we list the files of the current directory with their sizes and convert the results into XML :

```
declare namespace file = "java:java.io.File"

for $f in file:listFiles( file:new(".") )    (: or list-files() :)
return
    <file name="{ $f }" size="{ file:length($f) }"/>
```

Security:
: The use of Java Binding in a server environment is a potential security vulnerability. Therefore Java Binding is not allowed by default in the API (applications Qizx Studio and command-line tool enable it).

  Binding can be enabled on a class by class basis. To allow binding of a specific class, use the method `enableJavaBinding` in interface `XQuerySession`.

Static and instance methods:
: A static Java method must be called with the exact number of parameters of its declaration.

  A non-static method is treated like a static method with an additional first argument ('`this`'). The additional first actual argument must of course match the class of the method.

Constructors:

A constructor of a class is invoked by using the special function name "`new`". A wrapped instance of the class is returned and can be handled in XQuery and passed to other Java functions or to user-defined XQuery functions. For example:

```
declare namespace file = "java:java.io.File";
 file:new("afile.txt")
```

Overloading on constructors is possible in the same way as on other methods.

Wrapped Java objects

Bound Java functions can return objects of arbitrary classes which can then be passed as arguments to other functions or stored in variables. The type of such objects is `xdt:object` (formerly `xs:wrappedObject`). It is always possible to get the string value of such an object (invokes the Java method `toString()`).

Type conversions:

Parameters are automatically converted from XQuery types to Java types. Conversely, the return value is converted from Java type to a XQuery type.

Basic Java types are converted to/from corresponding XQuery basic types.

Since the XQuery language handles *sequences* of items, special care is given to Java arrays which are mapped to and from XQuery sequences. In addition, a `Vector`, `ArrayList` or `Enumeration` returned by a Java method is converted to a XQuery sequence (each element is converted individually to a XQuery object).

The type conversion chart below details type conversions.

Overloading

Overloaded Java methods are partially supported:

- When two Java methods differ by the number of arguments, there is no difficulty. XQuery allows functions with the same name and different numbers of arguments.

- When two Java methods have the same name and the same number of arguments, there is no absolute guaranty which method will be called, because XQuery is a weakly typed language, so it is not always possible to resolve the method based on static XQuery types (Resolution at run-time would be possible but much more complex and possibly fairly inefficient).

  However, static argument types can be used to find the best matching Java method. For example, assume you bind the following class:

  ```
  class MyClass {
      String myMethod(String sarg) ...
      int myMethod(double darg) ...
  }
  ```

  Then you can call the `myMethod` (or `my-method`) function in XQuery with arguments of known static type and be sure which Java method is actually called:

  ```
  declare namespace myc = "java:MyClass"
  myc:my-method(1)   (: second Java method is called :)
  myc:my-method("string")   (: first Java method is called :)
  ```

  1. in the first call, the argument type is `xs:integer` for which the closest match is Java double, so the second method is called.

  2. In the second call, the argument type is xs:string which matches `String` perfectly, so the first method is called.

  Of course it is possible to use XQuery type declarations, or constructs like `cast as` or `treat as` to statically specify the type of arguments:

```
declare function local:fun($s as xs:string) {
   myc:my-method($s)   (: first Java method is called :)
}
```

or:

```
 myc:my-method($s treat as xs:string)   (: first Java method is called :)
```

Limitations

There are still some limitations when in both methods the argument types is any non-mappable Java class (`xdt:object` in XQuery):

```
class MyClass {
    Object myMethod2(ClassA arg) ...
    int    myMethod2(ClassB arg) ...
}
```

In that case there is currently no way in Qizx to specify the static type of the actual argument, so the result is unpredictable and may result in a run-time error.

Table 8.1. Types conversions

Java type	XML Query type
void (return type)	empty()
String	xs:string
boolean, Boolean	xs:boolean
double, Double	xs:double
float, Float	xs:float
java.math.BigDecimal, java.math.BigInteger	xs:decimal
long, Long	xs:integer
int, Integer	xs:int
short, Short	xs:short
byte, Byte	xs:byte
char, Character	xs:integer
com.qizx.api.Node	node() ?
org.w3c.dom.Node	node() ?
java.util.Date, java.util.Calendar	xs:dateTime ?
other class	xdt:object ?
String[ ]	xs:string *
double[ ], float[ ]	xs:double *
long[ ], int[ ], short[ ], byte[ ], char[ ]	xs:integer *
com.qizx.api.Node[ ]	node()*
other array	xdt:object *
java.util.Enumeration, java.util.Vector, java.util.ArrayList (*return value only*)	xdt:object *

Part V. Tools

Note

This part is a copy of the documentation of the Qizx database engine product.

Operations on XML Libraries are available in Qizx/open only in client mode (see documentation of Qizx Server in full Qizx product).

Name

qizx — Qizx command line tool

Synopsis

`qizx` argument...

Description

qizx is a simple command-line interface for administrative and development use.

It provides basic operations on XML libraries, in particular:

- Creating XML Libraries, and performing administrative tasks (like re-indexing, backup).
- Importing XML documents into a Library, by parsing files or URL's.
- Executing XQuery expressions from files.
- Outputting the results of a XQuery execution to a file, with a number of options.
- Exporting a XML document or a collection from a Library.

The command-line option switches allow all these basic operations. For more complex problems, it is still possible to benefit of the full power of the XQuery language (and of extension functions provided by Qizx) by executing a script.

Options

Option switches always start with a minus sign. They can be followed by an argument. The letter case is generally significant.

An argument not starting with '`-`' is considered a XQuery source file to be executed.

Attention: processing of options has changed in version 4.0. The order of options is no more relevant. This makes it simpler to use, but induces some limitations: some operations (import, backup) can be performed only once in a launch.

General

`-group` *`path`*, `-g` *`path`*
: Specifies the location of a group of XML Libraries - or the address of a remote server.

  - **Local Library group on disk**: the path points to the root directory of the Group.
  - **Server**: the path is an HTTP URL like http://somehost:8080/qizx/api.

    A default installation of a Qizx Server would end with "/qizx/api" which corresponds to the Qizx REST API connector. But this path - of course the host and the port too - depend on the configuration of the server. See the Server installation documentation for more details.

`-library` *`library_name`*, `-l` *`library_name`*
: Specifies the name of a XML Library inside the selected group. The library must exist, unless the option `-create` is used (see below), otherwise the tool will stop in error.

  Most operations, like executing queries, require an XML Library.

  In local group mode, the XML Library will be locked for exclusive access.

-login *username:password*
: Used when connecting to a Qizx server that requires authentication. Since the password may appear on the command-line, this is not recommended for the best security. You may want to use the following switch -auth:

-auth *secret-file*
: Specify login credentials read from a file for better security. If authentication is required, credentials will be read from this file. The file should contain the following values:

```
login=admin
password=xxxx
```

: Of course the file should be protected from reading by other users.

Administration operations

`-create`
: Using this option, the group specified with option `-group` is created if it does not exist, and the library specified with option `-library` is created if it does not exist.

: The option has no effect if both exist.

: In client/server mode, only a Library can be created, this would not create a group since it is defined by the server.

`-import` *`collection XML-file-or-directory...XML-file-or-directory`*
: Import one or several XML documents into a collection (the collection is created if it does not yet exist): documents are parsed (they must be valid), stored and automatically indexed.

: Several XML file paths or directory paths can follow this option switch. Directories are scanned recursively, plain files encountered are considered XML and tentatively stored (the `-include` and `-exclude` options below can be used to filter files). A parsing error does not stop the load process.

: The path of the collection can be a document pattern containing a character '`*`': this character is replaced by an integer incremented on each document stored, providing an automatic naming mechanism for new documents. For example:

```
qizx -g mygroup -l mylib -import /a/collection/doc*.xml files...
```

: would create documents `/a/collection/doc1027.xml`, `/a/collection/doc1028.xml`, `/a/collection/doc1030.xml`, etc. The numbers are of course guaranteed to be unique, and always increasing, but no other assumption can be made about their values.

`-include` *`suffix`*
: Used together with `-import`: restricts the importing operation to files ending with this suffix (case insensitive). Can be used before or after `-import`, but must precede the XML file list. For example `-include .xml` would select only the files whose name ends with `.xml` or `.XML` or `.Xml`, etc.

: This option can be repeated: `-include .xml -include .xsd` would select both files ending with `.xml` and `.xsd`. By default all plain files are taken, unless a `-exclude` option is present (see below)

`-exclude` *`suffix`*
: Used together with `-import`: eliminates form a storing operation files ending with this suffix (case insensitive). Can be used before or after `-import`, but must precede the XML file list. For example `-exclude .txt` would eliminate the files whose name ends with `.txt` or `.TXT`, etc.

: This option can be repeated: `-exclude .jpg -exclude .png` would eliminate files ending with either `.jpg` or `.png`.

`-export` *`member_path`*
: Exports a member of the selected library (Collection or Document) using its path.

`-indexing` *`path`*
Defines an Indexing specification for the Library. An Indexing specification is used for customizing the way XML documents are indexed in the Library. For more information, see ???.

If documents were already imported with a different indexing specification, it is strongly recommended to use the option `-reindex` (see below) to rebuild indexes.

`-reindex`
Rebuilds all the indexes from scratch, without altering documents.

`-optimize`
Forces the compaction of indexes, without altering the contents of the Library.

`-delete-library`
Using this option, a library is specified with `-library` will be removed from the group and physically deleted.

`-delete` *member_path*
Deletes a member of the selected library (Collection or Document) using its path.

`-backup` *backup_group_path*
complete backup of the specified Library to the location specified: this location must correspond to a directory on a file-system, where a group will be created if necessary, and in which a XML Library with the same name will be created (if it already exists, it is first erased).

Example:

```
qizx -c mygroup -library mylib -backup /backups/my-group
```

This creates a backup group at `/backups/my-group` (if necessary), then copies the Library `mylib` into the backup group.

`-check` *`log_file`*
Performs a structural check of all the Libraries of the group and report errors to the log file. This is intended for debugging purpose.

Note

This operation is currently not able to repair a damaged XML Library.

XQuery execution and settings

`-q` *`query-file`*, *`query-file`*
Executes the XQuery expression contained in that file(s). The `-q` option switch is optional (it has to be used only if the file path starts with a dash, which is rarely the case). Several query files can be executed in order. Notice that if you specify the value of XQuery variables (option -D), this applies to all scripts, whatever their respective order.

`-base-uri` *`URI`*
Define the base URI for locating parsed XML documents. Unless a query redefines this base-URI, it will be used for resolving relative document locations as in the function `doc()`.

This option has no effect when connecting to a server.

`-module-base-uri` *`URI`*
Define the base URI for locating XQuery modules.

This option has no effect when connecting to a server.

`-i` *`collection`*
Defines the ``Implicit Collection'' i.e the set of documents or Nodes used as search roots when a XQuery Path Expression has no explicit root.

For example the expression `//ELEM` has no explicit root, while `collection("/mycollec")//ELEM` has the explicit root `collection("/mycollec")`, a Collection of the current XML Library.
Using this option is quite useful, as it allows writing XQuery scripts which are independent on the actual data used as input. Furthermore it makes scripts more concise.

If the Implicit Collection is defined through this option, for example `-i /mycollec` , the expression `//ELEM` is equivalent to `collection("/mycollec")//ELEM`.

Values: acceptable values for this option are the same as the argument of function fn:collection (for Qizx/open, see here [xv]):

- the path of a Document or a Collection inside an XML Library ,
- a file path or an URL: for example `dir1/doc1.xml` or `http://foobar.com/docs/summary.xml`
- a file pattern: `dir/*.xml`
- a semicolon-separated list of the above elements: `dir1/*.xml;dir2/doc2.xsl`

Note

If you want to use a single document, append a comma or semicolon after its path or URL.

Note

The function `collection()` without argument, or the deprecated XQuery function `input()` can also be used instead of an implicit root.

`-domain` *`collection`*
Alias for option `-i` above.

`-D`*`variable_name=value`*
Defines the value of a global variable. For example if the variable is declared like this:

```
declare variable $output external;
```

then the option `-Doutput=foo` initializes `$output` with the string value "`foo`".

If the variable is declared with a type, an attempt to cast the string value to the declared type is made.

If the variable is declared with an initial value, this value is overridden.

`--` *`argument`* ... *`argument`*
The double dash switch is used to pass command-line arguments to a XQuery script. It stores all following command-line tokens into the predefined variable `$arguments`. For example:

```
qizx myscript.xq -- arg1 arg2 arg3
```

runs the script `myscript.xq` after putting the sequence of 3 string items ("`arg1`", "`arg2`", "`arg3`") into variable `$arguments`.

Note

Because of this option, the scripts are always executed after interpretation of all other options.

`-timezone` *`duration`*
Defines the *implicit timezone* in the dynamic XQuery context. The value must be in `xs:duration` form, for example `-timezone -PT5H`.

This option has no effect when connecting to a server.

`-collation` *`uri`*
Defines the default collation for string comparisons.

Collations are supported through Java collators based on a locale name, for example "`en`" or "`fr-CH`". There is currently no support for plugging user-defined collators.

Syntax of the URI of a collation:

- Leading slash (so that the URI is absolute, otherwise it would be dereferenced relatively to the base-uri property of the static context).
- Name of a locale following the Java conventions.
- An optional URI fragment (beginning with a '#') whose value is "primary", "secondary" or "tertiary", defining the "strength" of the collator (see the Java documentation for more details). The value "primary" is less specific than "tertiary".

  If the strength is absent, it is in general equivalent to "tertiary".

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`" refer to the basic Unicode codepoint matching (or absence of collation). This is the default collation, unless redefined in the static context.

Output options

`-X`*`option=value`*
Defines a serialization option for result output. For example `-Xmethod=html` produces results in HTML markup.

For details of serialization options, see the documentation of the `x:serialize()` XQuery extension function.

`-out` *`file`*
output the result of a XQuery expression to a file (defaults to standard output).

`-wrap`
wraps the displayed results in description tags. For example with `-wrap` the expression `1`, "`a`" would display:

```
Query ? 1, "a"
<?xml version='1.0' encoding='UTF-8'?>
<query-results>
  <item type="xs:integer">1</item>
  <item type="xs:string">a</item>
</query-results>
```

instead of:

```
Query ? 1, "a"
1 a
-> 2 item(s)
```

`-jt`
trace use of Java extension functions (for debugging).

This option has no effect when connecting to a server.

`-tex`
verbose display of run-time exceptions (for debugging).

Note

In Qizx/open, only the query execution options and output options are available

Examples

This section is an How-To for some common operations with the qizx command line tool:

Create a group with a single XML Library

```
qizx -group D:\xmldb\group1 -library orders -create
```

This creates a group in the directory `D:\xmldb\group1` (which must be non-existent or empty), containing a single XML Library named '`orders`'.

Create an empty group

```
qizx -group D:\xmldb\group1 -create
```

This creates a group in the directory `D:\xmldb\group1`, without any XML Library inside.

Connect to a server

```
qizx -login me:mypassword -group http://localhost:8080/qizx/api script.xq
```

This connects to a Qizx server and executes the script on this server. Most other commands and options can be used in this mode.

Authentication, if required, can be provided by option -login, or by -auth (use of a secret file), or would be read on the console.

Import XML documents into an existing XML Library

```
qizx -group D:\xmldb\group1 -library orders -import /2007/june c:\data\orders\june2007\*.xml
```

This assumes that the group at `D:\xmldb\group1` already exists and contains a Library named '`orders`'. Then the specified XML documents are stored into the Collection `/2007/june`. For example the document `c:\data\orders\june2007\A.xml` will be stored in the library at `/2007/june/A.xml`.

Create a group with a single XML Library and store XML documents

```
qizx -group D:\xmldb\group1 -library orders -create -import /2007/june c:\data\orders\june2007\*.xml
```

This is a combination of the previous commands: the group and library are created and immediately after the documents are stored into the Library '`orders`'.

Import XML documents into an existing XML Library with filters

```
qizx -group D:\xmldb\group1 -library data -import /2007/june -include .xml -include .xsl \
    c:\data\orders\june2007
```

Assuming that the group at `D:\xmldb\group1` already exists and contains a Library named '`orders`', then all XML documents contained within directory `c:\data\orders\june2007` (at any depth), and whose name ends with `.xml` or `.xsl` are stored into the Collection `/2007/june`.

```
qizx -group D:\xmldb\group1 -library data -import /2007/june -exclude .jpg \
    c:\data\orders\june2007
```

Assuming that the group at `D:\xmldb\group1` already exists and contains a Library named '`orders`', then all XML documents contained within directory `c:\data\orders\june2007` (at any depth), and whose name does not end with `.jpg` are stored into the Collection `/2007/june`.

Delete an XML Library within a group

```
qizx -group D:\xmldb\group1 -library dataLib -delete-library
```

Deletes the library '`dataLib`' (selected by `-library dataLib`) and all its contents. Beware, this operation is irreversible.

Delete a Document or a Collection within a Library

```
qizx -group D:\xmldb\group1 -library dataLib -delete /2007/june
```

Deletes the collection `/2007/june` in library '`dataLib`' and all its contents (documents and sub-collections). Beware, this operation is irreversible.

```
qizx -group D:\xmldb\group1 -library dataLib -delete /2007/june/order1.xml
```

Deletes the document `/2007/june/order1.xml` in library '`dataLib`'. Beware, this operation is irreversible.

Qizx Studio Help

XF, XMLmind <qizx-support@xmlmind.com>

Version 4.0

May 10, 2010

Abstract

Online help of Qizx Studio, a graphic user interface for Qizx.

Qizx Studio is a graphic user interface built on top of the API provided by the Qizx XML indexing and query engine.

Qizx Studio has several purposes:

- Offer an interactive tool to edit, execute and debug XQuery queries (there is no debugger yet, but that is planned for a future version such as 4.1 or 4.2).
- Offer an easy-to-use interface for administering XML Libraries.
- Demonstrate most of Qizx functionalities through menus and dialogs.

This documentation assumes that you have at least basic notions about XQuery and Qizx (XML Library, Collection, Document).

1. Starting Qizx Studio

Qizx Studio can be started:

- From a graphic environment
- From the command line: it supports a few option switches similar to the command-line too qizx. Here are the main ones:

  `-group` *`path`*, `-g` *`path`*
  : Specifies the location of a group of XML Libraries - or the address of a remote server.

    - **Local Library group on disk**: the path points to the root directory of the Group.
    - **Server**: the path is an HTTP URL like http://somehost:8080/qizx/api.

      A default installation of a Qizx Server would end with "/qizx/api" which corresponds to the Qizx REST API connector. But this path - of course the host and the port too - depend on the configuration of the server. See the Server installation documentation for more details.

  -login *`username:password`*
  : Used when connecting to a Qizx server that requires authentication. Since the password may appear on the command-line, this is not recommended for the best security. You may want to use the following switch -auth:

  -auth *`secret-file`*
  : Specify login credentials read from a file for better security. If authentication is required, credentials will be read from this file. The file should contain the following values:

    ```
    login=admin
    password=xxxx
    ```

    Of course the file should be protected from reading by other users.

2. The 'XML Libraries' tab

This tab is used to manage XML Libraries: creation, browsing, maintenance.

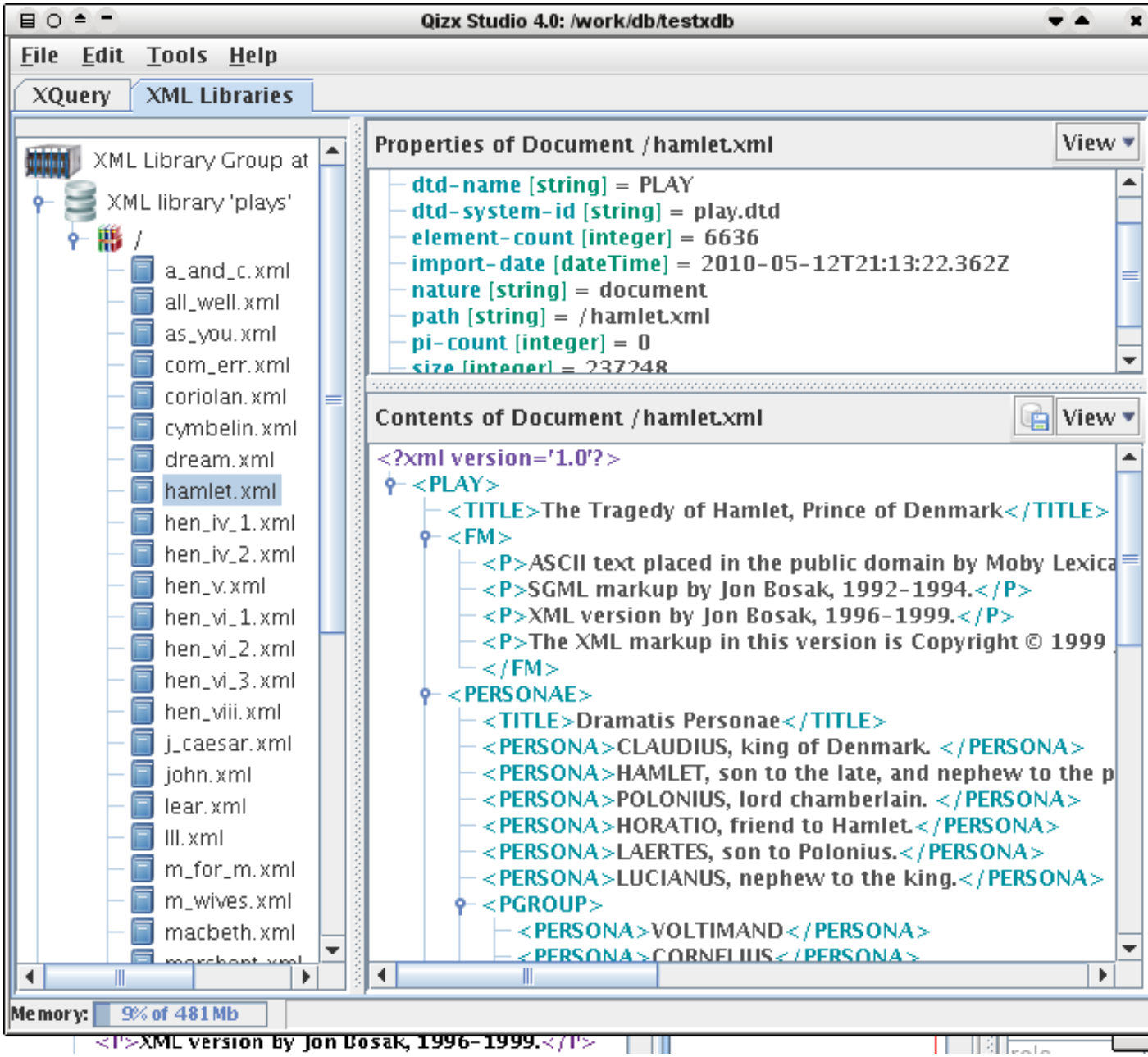
Note

In Qizx/open, this tab is absent.

It is divided in three views:

1. Library Browser (left side): a tree view to browse XML Libraries and contained Collections
2. Metadata Properties view (top right): displays the properties of a selected Document or Collection
3. Contents of Document view (bottom right): displays the contents of a selected XML Document.

Figure 1. XML Libraries tab



2.1. Library browser

This view displays the contents of a XML Library Group as a tree.

The view displays the following objects:

- XML Library Group  : the currently opened group of Libraries.
- XML Library  : a library belonging to the group
- Collection  : a Collection inside a Library. May contain other collections.
- Document  : a well-formed XML document stored and indexed in a Collection.

Each kind of object has an associated right-click menu, which gives access to a number of operations:

- **XML Library Group right-click menu**:
  - **Open Library Group**: opens a group of XML Libraries located in a directory. A file chooser appears to select that directory.

    In this mode, Qizx Studio has exclusive access to the XML Libraries. If another application or server already has locked the Libraries, an error panel will appear.

    This command also allows opening a *single* XML Library by selecting its root directory: It is a special case where the Library Group has no defined location, and therefore creating other libraries is not possible.
  - **Connect to Server**: opens a client connection to a Qizx Server.

    This will likely present an authentication dialog asking for a user name and a password (depending on the configuration of the server).
  - **Close Library Group**: closes the current group of Libraries. If currently connected to

    Passes in a mode where no Library is available. Note that it is still possible to run XQuery expressions, as long as they don't perform queries on a Library. This is equivalent to using Qizx/open.
  - **Create Library Group**: creates a group of Libraries in a directory. This directory is first selected by a file chooser. It must be empty or non-existent. The Library Group is created in the directory, then a dialog asks for the name of the first Library to be created inside the group.
  - **Create Library**: allows creating more XML Libraries inside the current group.
- **XML Library right-click menu**:
  - **Import Documents**: to store XML documents into the selected Library (in the root Collection); see the Import Documents dialog [53].
  - **Use Library as Query domain**: *query domain* means the default root of a XQuery/XPath path expression. For example, assume that you have a Library containing the plays of Shakespeare (as marked up by Jon Bosak), that you select the Library as the query domain, then the query `//SCENE` will return all `SCENE` elements in the Library. Note the particular query `//SCENE` has no explicit root or start-point. It uses here the default query domain.

    This feature is not supported in client-server mode (because it does not make much sense).

  - **Indexing**: a sub-menu that deals with indexing specifications.
    - **Indexing Specification**: load a new specification written in XML. See details here [55].
    - **Rebuild all indexes**: this operation is normally required after changing the indexing specifications.
    - **Optimize Library**: this a compaction operation that can slightly improve the performance of queries on the Library. It is normally performed automatically after a certain number of transactions.
  - **Backup Library**: this command makes a backup copy of a Library to an external directory.
  - **Delete Library**: this command physically destroys the selected Library.
  - **Refresh**: useful in client mode to see the latest state of the Library: another client may have modified it.

    Notice there is currently no notification mechanism that would allow an automatic refresh on update of an XML Library.

- **Collection right-click menu:**
  - **Use as Query domain**: query domain means the default root of a XQuery/XPath path expression (See here [46] for more details). If a Collection is used as query domain, the query is restricted to all documents contained within the Collection at any level.
  - **Import Documents**: command used to store XML documents into the Collection. Invokes the Import Documents dialog [53].
  - **Create Sub-Collection**: asks for the name of a Collection which will be child of the selected collection..
  - **Copy Collection**: this command allows copying the selected Collection and all its contents (sub-collections and documents) to another location in the same Library.
  - **Rename Collection**: this command allows changing the name or the location of the collection.
  - **Delete Collection**: this command destroys the selected Collection and all its contents (sub-collections and documents).
  - **Refresh**: useful in client mode to see the latest state of the Collection: another client may have modified it.
- **Document right-click menu:**
  - **Use as Query domain**: *query domain* means the default root of a XQuery/XPath *path expression* (See here [46] for more details). If a Document is used as query domain, the query is restricted to this particular document. For example if the query domain is the document `/coll/doc1.xml`, the query `//TITLE` is equivalent to `doc("/coll/doc1.xml")//TITLE`.
  - **Export Document**: command used to extract the XML contents of the document into a local file. Invokes the Export Document dialog [55] which allows choosing serialization options.
  - **Copy Document**: this command allows copying the selected Document to another location in the same Library.
  - **Rename Document**: this command allows changing the name or the location of the document.
  - **Delete Document**: this command destroys the selected Document.
  - **Refresh**: useful in client mode to see the latest state of the document: another client may have modified it.

2.2. Metadata Properties view

This view displays the properties (also called *metadata*) of the currently selected *Library member*, i.e Document or Collection.

The name (in blue) and the value of the property are displayed.

Modifying properties

By right-clicking on a property, its value and type can be edited.

Note: though the 'path' property can be edited, it is in fact built-in and will not change. It can be changed through the 'Rename' operation, by right-clicking on the corresponding Collection or Document.

2.3. Document display

This view displays the XML contents of a selected document. It should be empty if no document is selected.

2.3.1. Export document to file

With the button  , you can save the document to a file. This invokes the Document Export Dialog [55].

2.3.2. View mode

This drop-down selector selects one of two display modes for a XML *Data Model* (i.e the contents of a document):

- Markup: this is a XML-like display
- Data Model: shows each individual node constituting the data model.

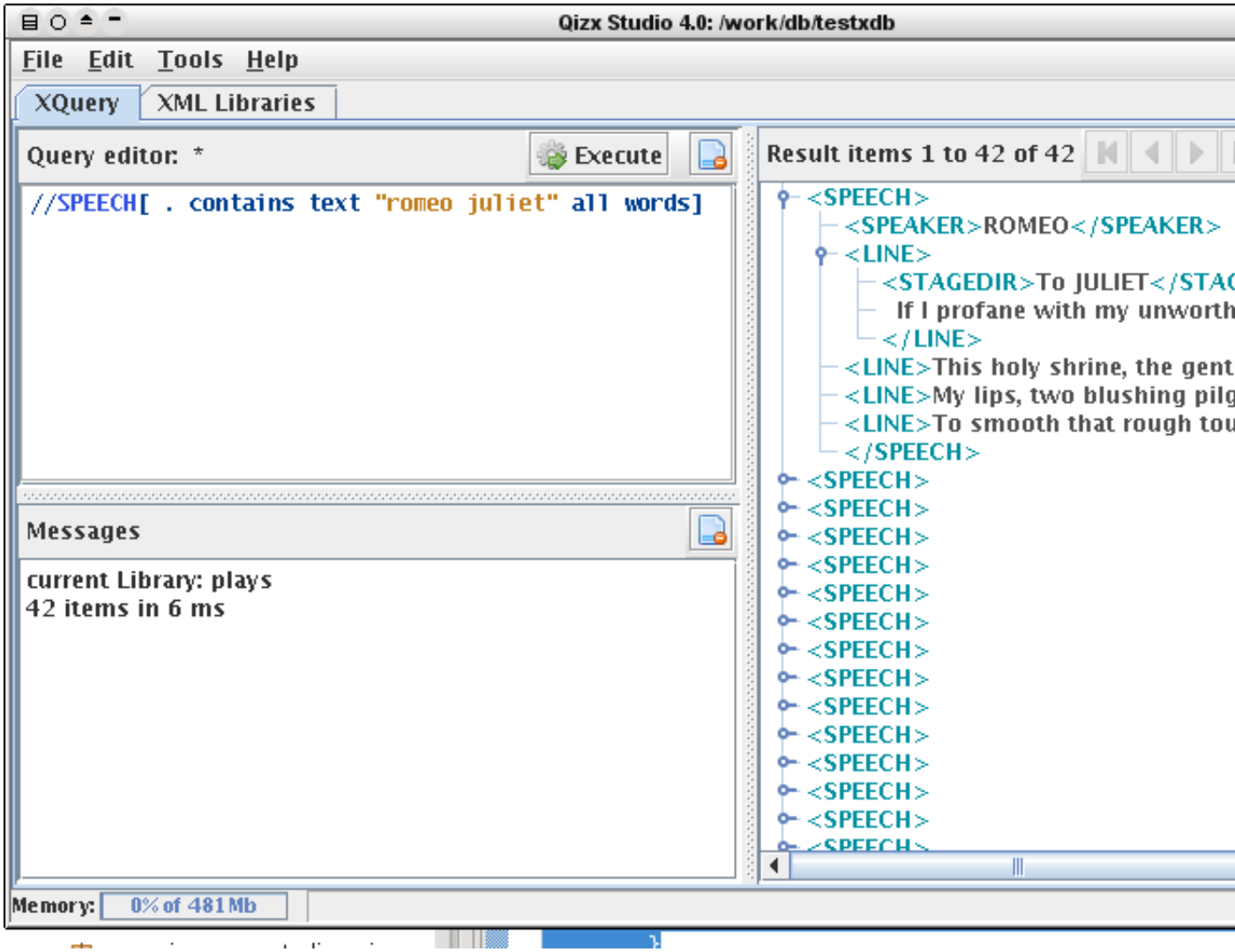
3. The 'XQuery' tab

This tab is used to edit and execute XQuery queries.

It is divided in three views:

1. Query editor (top left): a text editor with execution button and query history.
2. Messages view (bottom left): displays compilation and execution messages.
3. Query Results (right): displays the items of the result sequence.

Figure 2. XQuery tab



3.1. XQuery Editor

This area is a basic text editor of XQuery source code, performing syntax coloring.

A file can be loaded in the editor through the menu File → Open XQuery, and conversely the source code can be saved with menu File → Save XQuery.

Caution

No check is performed when saving or when exiting the application, however the *query history* (see below) keeps trace of all queries entered.

Specifying the path of a XQuery source file in the command-line of Qizx Studio will automatically load the file. This happens if the file extension (in principle `.xq`) has been associated with the Qizx Studio application, depending on the Operating System used.

The editor has several buttons and controls in the tool-bar above:

3.1.1. Query Execution

The button **Execute** compiles the current query and evaluates it. During execution the button changes to **Stop**.

The result sequence is displayed in the Result View. A message in the Message View below tells the number of items in the result sequence, and the time in milliseconds taken by the evaluation.

3.1.2. Stopping Query execution

A lengthy evaluation can be canceled with the button Stop. No results are displayed.

3.1.3. Clear editor text

The button clears the editor, to type a new expression.

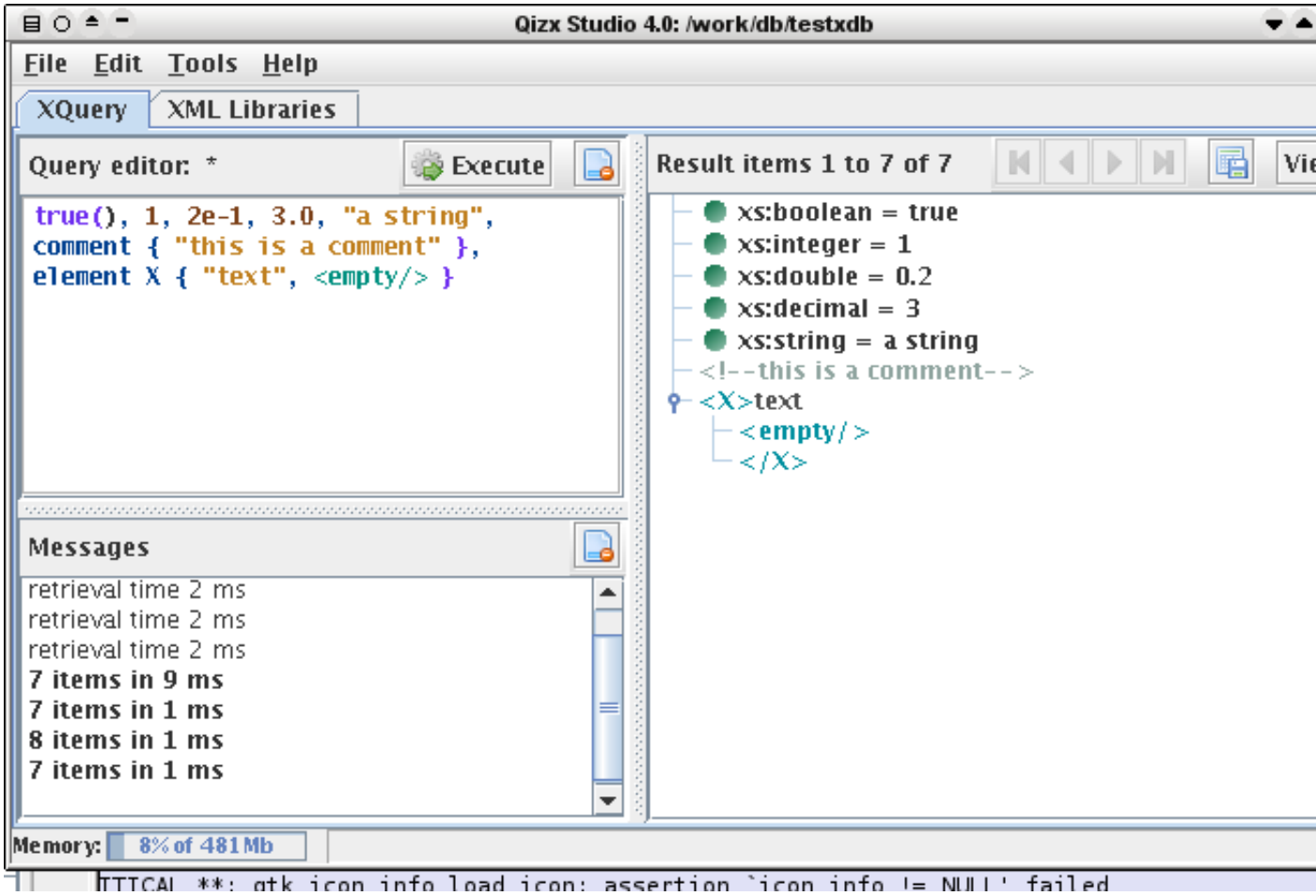
3.2. Result View

This view displays the *result sequence* produced by the evaluation of a XQuery expression.

- Simple items (integer, string, etc) are displayed with their type.
- Node items are displayed either in "markup" style, looking like XML.

Results are displayed by pages of 100 items. A set of buttons can be used to traverse the result sequence if it is long.

Figure 3. XQuery tab with miscellaneous results



3.2.1. Move forward and backward in result sequence

The two vertical arrows move the position of the displayed page by 100 items forward or backward.

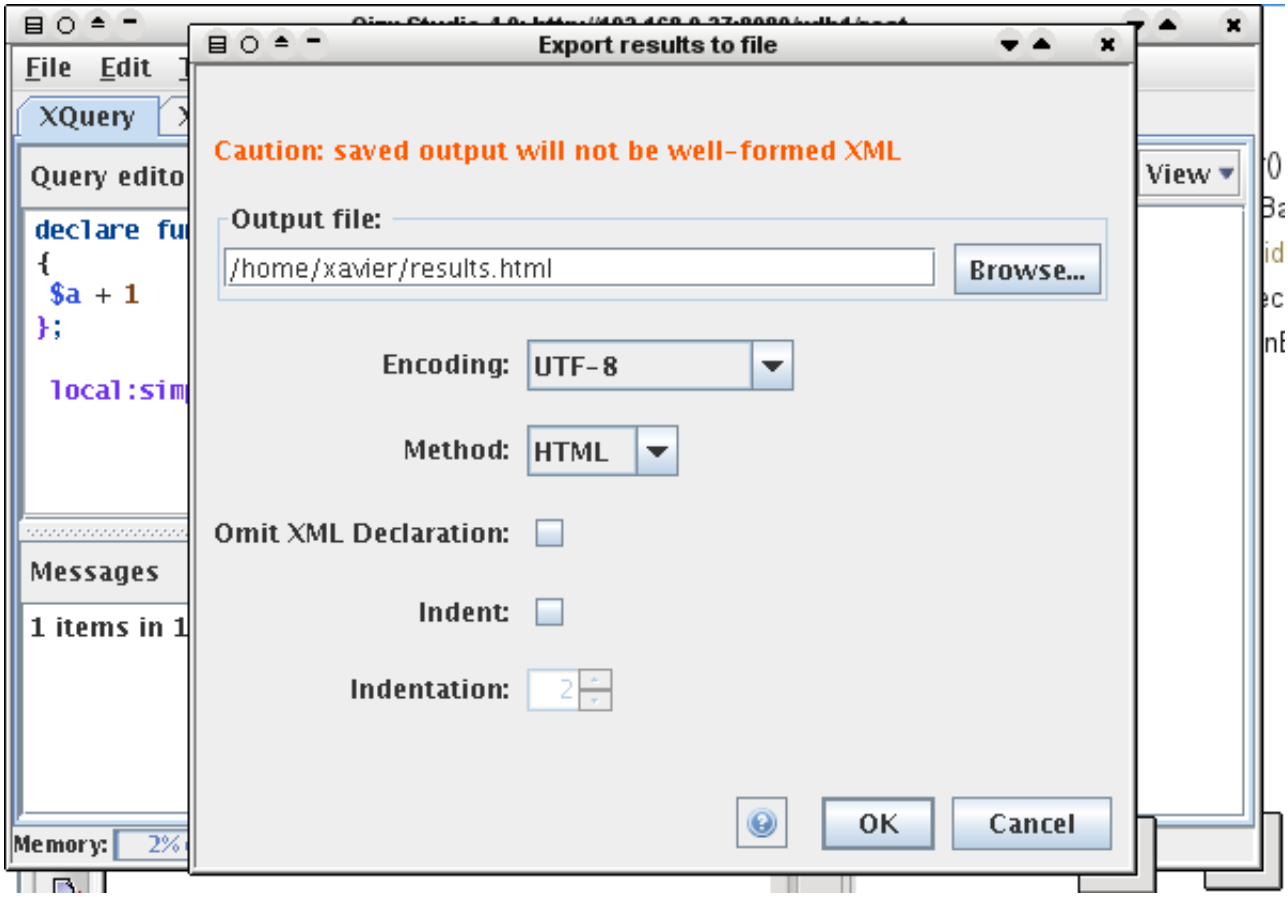
3.2.2. Export result sequence to a file

This button saves the whole result sequence onto a file.

Note that the resulting file will not in general represent a well-formed XML document, unless the result sequence contains a single Node. A message signals when the result is not well-formed.

The invoked dialog allows choosing serialization options. Some of these options (HTML) do not always make sense, depending on the actual results.

Figure 4. Export results dialog



3.2.3. Change the display style of results

This command changes the display style for Nodes only.

- The Markup style mimics XML markup.
- The Data Model style is a tree view of Node structures.

3.3. Message View

This view displays compilation and execution messages.

When an error is displayed, the location is underlined: by clicking on it, the cursor of the Query Editor is placed on the error location.

4. Dialogs

Note

In Qizx/open, only the XML Catalogs and Error Log dialogs are available.

4.1. Open local Library Group dialog

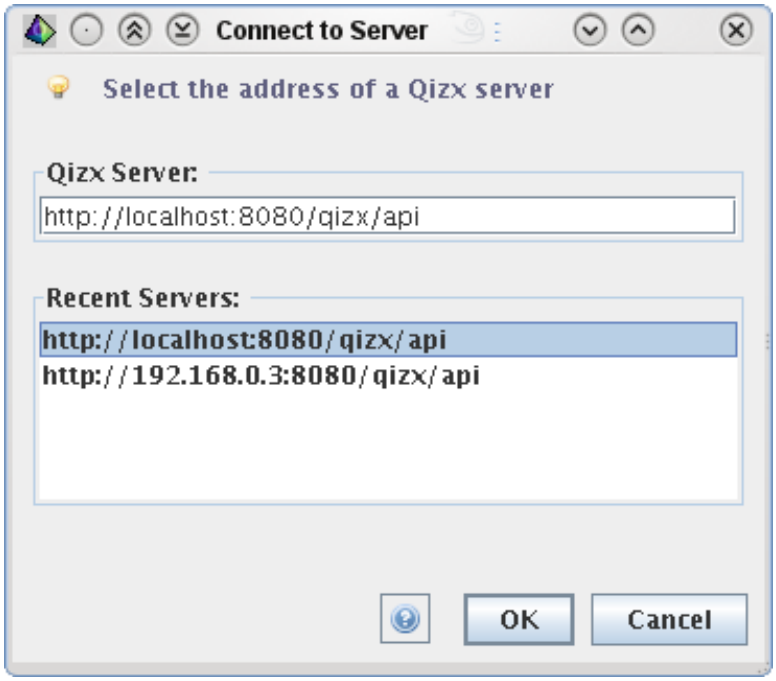
Used for opening an XML Library group located on a local disk.

- File browser: selects a directory on a local file-system.
- History of recently opened groups: double clicking on this list selects the clicked entry.

4.2. Connect to Server dialog

Used for opening an XML Library group managed by a remote server.

Figure 5. Connect to Server dialog



- Field text for the URL of the server:

  this URL is of the form http://*host:port/webapp*/api, where host, port and webapp depend on the installation.

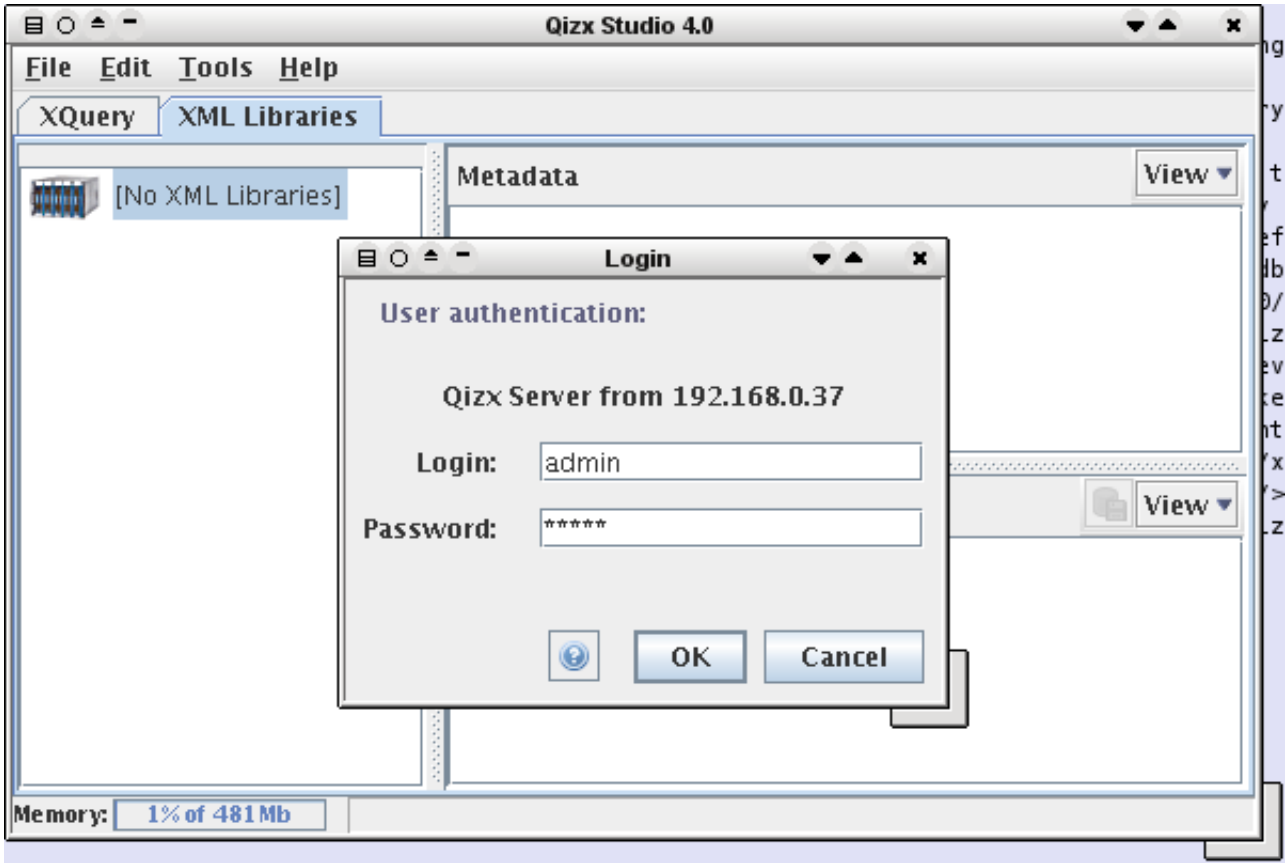
  The path 'qizx/api' is the default for the REST API of Qizx, but it can be changed in the configuration of the server.

- History of recently opened groups: double clicking on this list selects the clicked entry.

Authentication

Generally, a server will require a login and password on connection. This is configured in the installation of the server.

Figure 6. Connect to Server authentication dialog



4.3. 'XML Catalogs' dialog

A dialog used to define XML Catalogs used when documents are imported.

Qizx supports the OASIS XML Catalogs specifications.

The dialog edits the value of the system property "`xml.catalog.files`" which can contain a semicolon-separated list of catalog files.

It is possible to add file paths and URLs to the list.

4.4. 'Create Collection' dialog

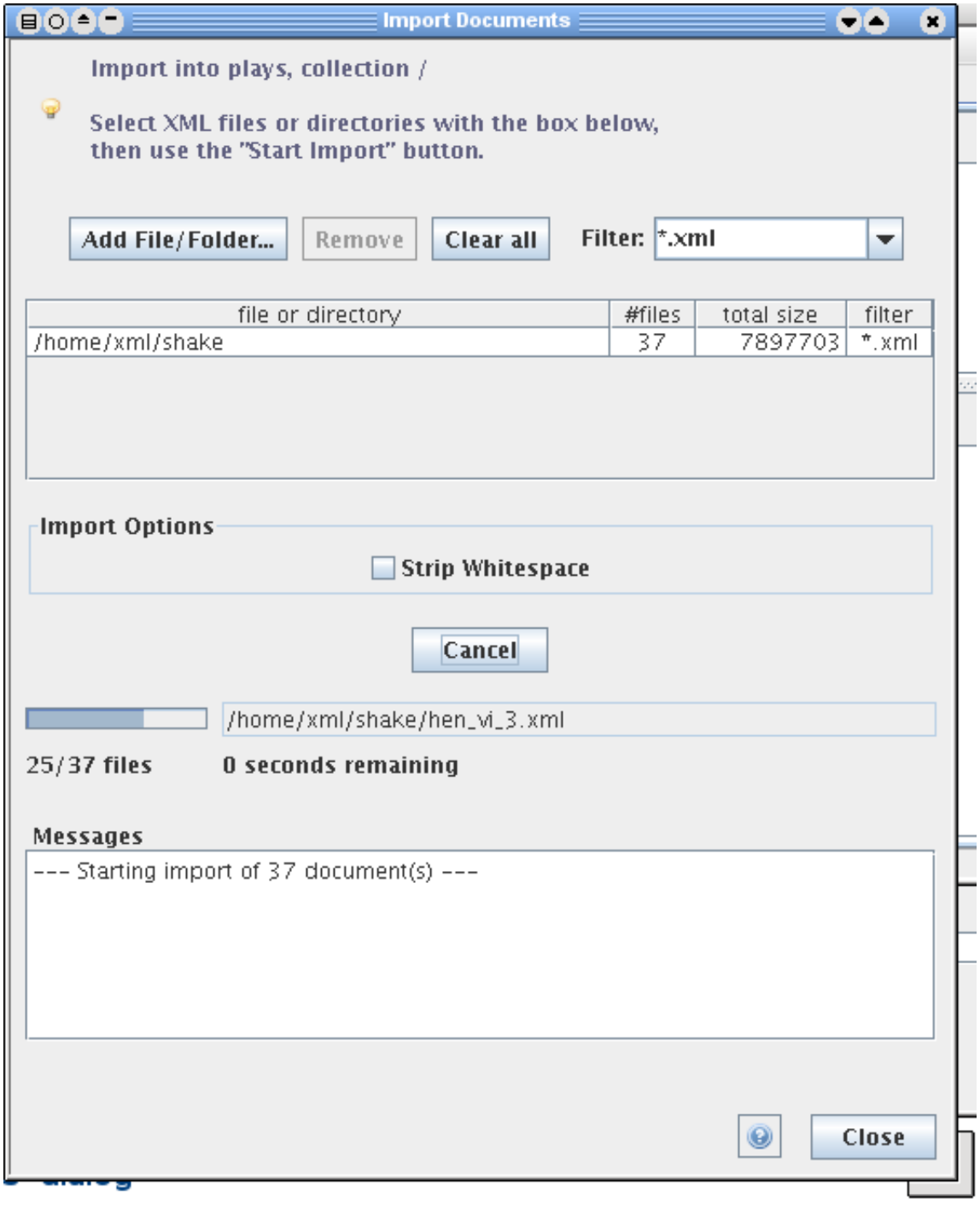
This dialog allows creating a child Collection of the selected Collection.

It simply prompts for the name of a child collection. This name must not contain the slash '/' character.

4.5. 'Import Documents' dialog

This dialog allows parsing, storing and indexing one or several XML documents into a Collection inside a XML Library.

Figure 7. Import dialog



An import operation is performed in two steps:

1. Create an import list of XML files or of directories. This list displays the path, the number of files (for directories), the total size in bytes, the filter used (for directories).
2. Push the button "Start Import".

To Add a file or directory (or several) to the list, use the button "Add File/Folder" and select the file(s) or directory.

For directories, you may first want to choose a filter for contained files. a new filter can be typed in the combo-box.

Items can be removed from the list by selecting them and using the button Remove, or the button "Clear all".

DTD and Schema are resolved through XML catalogs. The XML Catalogs menu [53] allows editing the catalogs.

Parsing errors are reported in the Messages area at bottom.

4.6. 'Export Document' dialog

This dialog is used for extracting a selected Document from a Library and write its contents back to a file.

The dialog allows choosing the file and Serialization options.

It is also used for exporting the results of a XQuery evaluation.

4.7. Metadata Property Editor dialog

This dialog allows adding a new property or editing an existing property. It is invoked by right-clicking on the name of a property.

The value is edited in string form. The type selected with the Type combo-box is then used to parse the value accordingly.

Possible types are currently:

- String.
- long integer (`xs:integer`).
- double (`xs:double`).
- Date (`java.util.Date`) : a value is edited in ISO standard form, for example 2010-05-01T14:54 .
- boolean (`xs:boolean`).
- node(): a single node (generally an element).
- expression: any executable XQuery expression can be entered. Only the first item will stored as property value.

4.8. 'Change Indexing Specification' dialog

This dialog allows defining and changing the Indexing Specifications. See ??? for more information about Indexing Specifications.

There are three ways of modifying the Indexing Specifications:

- Directly edit basic specifications using the simple editor presented in the dialog. This editor allows editing the most common indexing properties, but is too limited to handle all the Indexing capabilities.
- Load a specification file (XML format described in the documentation), using the button Load From File...
- Reset Indexing specifications to the default value (button "Restore To Default").

After any change (when using the button Apply), the user is suggested to rebuild the indexes entirely. This is strongly recommended for avoidance of inconsistencies in query results.

4.8.1. Reindexing Dialog

This is a simple dialog through which the indexes can be rebuilt entirely, using the current Indexing Specifications.

It is invoked automatically after a change in the Indexing Specification Dialog.

Please note that since Qizx 2.1, re-indexing is a synchronous operation. A progress bar is displayed by the dialog.

4.8.2. Optimize Library Dialog

This is a simple dialog through which the XML Library can be put into an "optimal" state. This operation involves compacting the document storage and the indexes, if necessary.

Please note that since Qizx 2.1, optimizing a Library is a synchronous operation. A progress bar is displayed by the dialog.

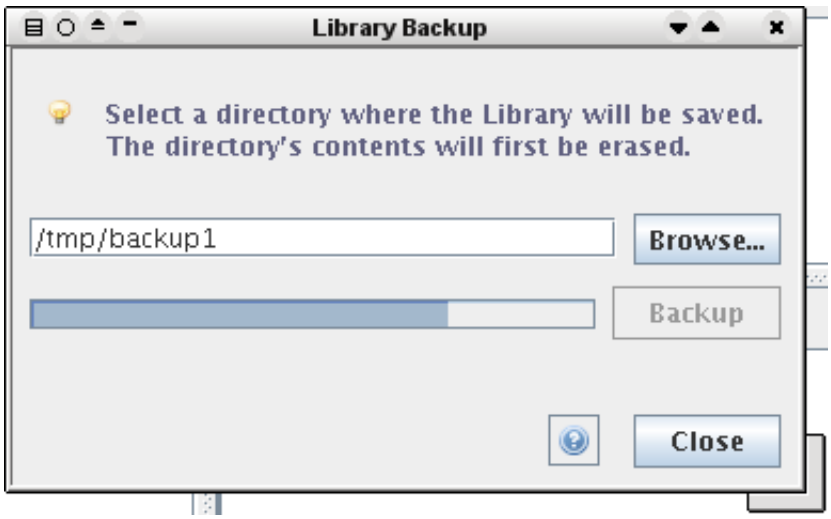
4.9. 'Backup Library' dialog

This dialog prompts you for a directory in the file system where the Library will be saved. The directory contents will be erased before backup.

This is a ``hot backup'' which saves a snapshot of the database: any modification made by another connection during the backup will be ignored. This is meaningful and useful in a multi-user environment, like a Web Application running in a servlet container.

The Restore operation consists simply of moving or copying the directory of the backup Library to the place of the original Library (see Administrator section of the manual for details).

Figure 8. Backup dialog



4.10. 'Error Log' dialog

This non-modal dialog appears when a serious error is detected by the Library manager.

Typically it appears if you try to use a XML Library that is already locked by another instance of the Qizx engine (an instance of Qizx Studio, of the qizx command-line tool, or one of your applications).