Introduction to XQC - the C Language Binding for XQuery

Vinayak Borkar, Matthias Brantner, Christopher Hillery, John Snelson

Overview

XQC is the C language binding for XQuery. It is intended to provide a standard API allowing applications to make use of any one of several available XQuery implementations. It was developed as a cooperative effort between developers of the XQilla XQuery engine[1] and the Zorba XQuery engine from the FLWOR Foundation[2].

XQC offers a variety of features for programmatic control over all aspects of XQuery processing, including: compiling queries into re-usable expressions; executing compiled queries and navigating through the results; binding external variables and the context item for use during query execution; modifying many components of the static and dynamic query contexts; and receiving and handling errors which may occur at any point during the process. XQC also offers several convenience features and the option for extensions to the basic functionality.

The benefits to having a standard, cross-engine API are numerous. Several specific ones include:

- The ability for developers to migrate between different XQuery engines without needing major code rewrites. This allows developers to try out several engines to see which is best for them, and to change engines later as new features or licensing options become available.
- The ability for tools to be written without depending on a particular XQuery implementation. Tools developers can deliver their products to a wider range of potential customers if they can develop to a common API.
- Eventually, the possibility of standardized training for XQuery/C developers.

The Java Database Connectivity (JDBC) standard[3] is a good example of how much benefit can be derived from a robust cross-application API standard. The Java world also has a standard API for XQuery, known as XQJ[4]. XQJ is an inspiration for XQC, although XQC does differ from XQJ in a number of respects.

XQC is an object-oriented API, even though it is for the C language. The current release (version 1) of the API comprises seven C structs, each of which contain only function pointers. Instances of these structs can be treated as objects; internally they contain implementation-specific data, and each function pointer acts as a method call operating on that data. As such, this document will refer to the structs as "classes", and to the functions contained in these structs as "methods". It is intended that later versions of XQC will also include a C++ language binding which closely mirrors the C API.

The current version of the XQC specification and mailing list archives are available via Sourceforge[5].

---

1. http://xqilla.sourceforge.net/
2. http://www.zorba-xquery.com/
3. http://java.sun.com/products/jdbc/overview.html
4. http://jcp.org/en/jsr/detail?id=225
5. https://sourceforge.net/projects/xqc/

Basics of Query Compilation and Execution

The entry point for users of XQC is the `XQC_Implementation` class. Client code obtains a pointer to an `XQC_Implementation` in an implementation-specific manner. For example, when using the Zorba XQuery processor, the following code will initialize the backing data store (using Zorba's basic "simple store") and return an `XQC_Implementation*` :

```
#include <xqc.h>
#include <zorba/simplestorec.h>

XQC_Implementation* impl;
void* store = create_simple_store();
zorba_implementation(&impl, store);
```

> *Note:* `zorba_implementation()` returns `XQC_Error`, an enum defined by XQC for specifying the success or type of failure for all XQC methods. This is why the `XQC_Implementation*` is returned via an output argument. Error checking and handing will be discussed in more detail in a later section.

To compile a query, the method `prepare()` is called on the `XQC_Implementation`. The result of preparing a query is an `XQC_Expression`. This class represents a compiled query, and allows the query to be executed repeatedly.

```
XQC_Expression* expr;
impl->prepare(impl, "(1+2, 3, 4)", NULL, &expr);
```

> *Note:* because C is not object-oriented, it is necessary to pass the instance itself as the first argument to any method. This idiom is used for all methods in XQC.

To execute a prepared query, the method `execute()` is called on the `XQC_Expression`. The result of executing a prepared query is an `XQC_Sequence`. This object represents an instance of the XQuery Data Model (XDM)[6].

```
XQC_Sequence* sequence;
expr->execute(expr, NULL, &sequence);
```

Results of Query Execution

`XQC_Sequence` provides a cursor-like interface for exploring the query results. Recall that an instance of the XDM is defined to be a sequence of Items, where each *Item* is either a *Node* or an *Atomic Value*. The `next()` method of `XQC_Sequence` allows you to advance through the sequence one item at a time. Details about the current item of a sequence can then be read via accessor methods. For example, to retrieve the value of each item as a C string, the `string_value()` accessor method can be used as follows:

```
const char* string;
while (sequence->next(sequence) == XQC_NO_ERROR) {
  sequence->string_value(sequence, &string);
  printf("%s", string);
}
```

---

6. http://www.w3.org/TR/xpath-datamodel/

Other accessors on `XQC_Sequence` include:

- `integer_value()` and `double_value()` - return the current item as a C int or a C double, respectively.
- `item_type()` - return the basic type of the current item as an `XQC_ItemType`, which is an enum comprising all seven XDM node types and the 23 non-derived XML schema simple types.
- `node_name()` - return the name of the current node (an element or attribute) as a QName (a namespace URI / localname pair).
- `type_name()` - return the type of the current item as a QName.

Static and Dynamic Contexts

You may have noticed the `NULL` arguments passed to the `XQC_Implementation::prepare()` and `XQC_Expression::execute()` methods. Together, these allow the user to provide the *Expression Context*[7] for evaluating the query.

`prepare()` can be passed a *Static Context* in the form of an instance of `XQC_StaticContext`. Client code obtains an `XQC_StaticContext` via the `XQC_Implementation::create_context()` method. Methods on `XQC_StaticContext` allow the client to look up and configure a large number of components of the static context, such as: the *statically-known namespace* definitions; the *default element/type namespace* and the *default function namespace*; the *base URI*; the *construction mode* ("preserve" or "strip"); the *ordering mode* ("ordered" or "unordered"); the *default order for empty sequences* ("least" or "greatest"); the *boundary-space policy* ("preserve" or "strip"); the *copy namespace mode* ("preserve" or "no-preserve", and "inherit" or "no-inherit"); and the *XPath 1.0 compatibility mode* setting. For all components which have enumerated possible values, XQC provides a C enum matching those values which the corresponding methods on `XQC_StaticContext` accepts. For instance, `XQC_BoundarySpaceMode` has two values, `XQC_PRESERVE_SPACE` and `XQC_STRIP_SPACE`, which can be passed to `XQC_StaticContext::set_boundary_space_policy()`.

Analogously, `execute()` can be passed a *Dynamic Context* in the form of an instance of `XQC_DynamicContext`. Client code obtains an `XQC_DynamicContext` via the `XQC_Expression::create_context()` method. Methods on `XQC_DynamicContext` are primarily for binding values to external variables in the prepared query via the `set_variable()` method, and for binding a value to the context item for the expression via the `set_context_item()` method. Both of these methods take an `XQC_Sequence` for the value, which allows chaining queries. The client can also specify the implicit timezone with `set_implicit_timezone()`.

Here is a complete example, which demonstrates: setting a static context component (base URI); executing a query with that static context; and chaining queries by binding the result of the first query to an external variable in a second query.

```
XQC_StaticContext* stat = NULL;
XQC_DynamicContext* dyn = NULL;
XQC_Expression* query1 = NULL, query2  = NULL;
XQC_Sequence* result1 = NULL, result2 = NULL;
```

---

7. http://www.w3.org/TR/xquery/#context

```
// prepare and execute query1
impl->create_context(impl, &stat);
stat->set_base_uri(stat, "http://www.example.com/");
impl->prepare(impl, "fn:resolve-uri(\"index.html\")", stat, &query1);
query1->execute(query1, NULL, &result1);

// prepare, bind variable, and execute query2
impl->prepare(impl,
  "declare variable $uri external; fn:concat($uri, \"#anchor\")",
  NULL, &query2);
query2->create_context(query2, &dyn);
dyn->set_variable(dyn, "", "uri", result1);
query2->execute(query2, dyn, &result2);

// retrieve the string-value of the first item in result2
const char* str;
result2->next(result2);
result2->string_value(result2, &str);
/* prints "http://www.example.com/index.html#anchor" */
printf("%s", str);
```

Error Handling

As mentioned earlier, virtually all methods in XQC return `XQC_Error`. This enum has values describing various error conditions that can occur when calling XQC methods, such as "invalid argument", "not a node" (returned by `XQC_Sequence::node_name()`), and "no current node" (returned by accessors on `XQC_Sequence` if the sequence has been exhausted or not yet begun). It further has values for some non-error conditions, such as "end of sequence" (returned by `XQC_Sequence::next()` when the sequence is complete) and the successful "no error" condition. It also has values for implementation failures such as "internal error" and "not implemented".

Finally, it has values for describing various classes of errors that can occur during query compilation and processing, such as *static errors*, *dynamic errors*, and *type errors*. However, frequently these will be insufficiently specific. The XQuery specification details a large number of explicit error conditions, each identified by a QName, that an implementation can raise to describe a wide range of processing errors. XQC client code can receive these values via an `XQC_ErrorHandler`. This class is slightly different than the five XQC classes we have seen so far in that the client code itself is responsible for allocating the object and implementing its single method. For client code to inform the XQC implementation about an `XQC_ErrorHandler`, it may be passed to the `set_error_handler()` method on either `XQC_StaticContext` or `XQC_DynamicContext`.

The single method on `XQC_ErrorHandler` is named `error()`. It will be called by the XQC implementation when an error is about to be raised. It will be passed the QName of the error, as well as a string description. If the error occurred because the XQuery called the `fn:error()` function, the argument to `fn:error()` will also be passed to the `XQC_ErrorHandler` as an `XQC_Sequence`.

Memory Management and Thread Safety

All of the XQC classes, except `XQC_ErrorHandler`, have a method `free()`. Client code is required to call this method when it is done with a given XQC object. There are documented correct orders for freeing objects; generally, any object which was initially created by a method on another XQC object must be freed before the object which created it. For example, users must free any `XQC_Expression` objects created by calls to `XQC_Implementation::prepare()` prior to freeing the corresponding `XQC_Implementation`.

There are two exceptions to this general rule:

1. If an `XQC_Sequence` is bound to an external variable via `XQC_DynamicContext::set_variable()`, the implementation takes ownership of that `XQC_Sequence` and now has the responsibility for freeing it (presumably when the `XQC_DynamicContext` is freed). Somewhat oddly, the same is not true for an `XQC_Sequence` passed to `XQC_DynamicContext::set_context_item()`; the client code remains responsible for freeing that sequence.
2. The `free()` method of `XQC_InputStream` (discussed in the next section) is actually called by the XQC implementation, not client code. This will be explained below.

Three XQC classes have documented rules regarding thread safety. `XQC_Implementation` and `XQC_Expression` are explicitly thread-safe, such that instances may be shared between multiple threads. `XQC_StaticContext` is explicitly *not* thread-safe; each thread should create its own when required.

Other Features

Additional XQC_Implementation factories

In addition to the `prepare()` and `create_context()` methods discussed earlier, `XQC_Implementation` has several other methods which create instances of XQC classes.

1. `prepare_file()` and `prepare_stream()` allow compilation of queries from a C `FILE*` and from an `XQC_InputStream` (discussed below), respectively.
2. `create_empty_sequence()`, `create_singleton_sequence()`, `create_integer_sequence()`, and `create_double_sequence()` create instances of `XQC_Sequence` with the corresponding contents.
3. `parse_document()`, `parse_document_file()`, and `parse_document_stream()` parse XML from a C `char*`, a C `FILE*`, or an `XQC_InputStream`, respectively, returning the contents as an `XQC_Sequence`.

XQC_InputStream

The seventh and final XQC class is `XQC_InputStream`. Like `XQC_ErrorHandler`, this is intended to be allocated and implemented by client code. It is used as a way for client code to pass data into the XQC implementation, specifically for compiling a query or parsing an XML document via the `prepare_stream()` and `parse_document_stream()` methods discussed in the last section. The main method the user must implement is `read()`, which the implementation will call to read a number of bytes from the client code. Also, as mentioned in the section on

memory management, there is a `free()` method which the client code must also implement. This is called by the XQC implementation when it has completed reading the data from the stream.

Extension interfaces

Each implementation of XQC will likely have some custom functionality it would like to provide. XQC allows for implementation-specific extension functionality via the method `get_interface()`, which is defined on the five XQC classes that the implementation creates (that is, all except `XQC_ErrorHandler` and `XQC_InputStream`). These methods allow client code to request implementation-specific interfaces by name.

Conclusions and Future Outlook

XQC is a young specification. The authors would like to encourage adoption of the API among a larger number of XQuery engines. Further, we would very much like to have additional contributors to the XQC specification itself and to any reference implementations of XQC. There are several outstanding projects which need additional insight and implementors. For example:

1. As mentioned in the overview, it is intended that XQC will comprise both a standard C API and a parallel standard C++ API. Version 1 defines only the C API.
2. To allow truly cross-engine compatibility, especially for tools developers, it would be ideal if the current requirement for an implementation-dependent entry point to `XQC_Implementation` could be dropped.
3. `XQC_Sequence` has no methods for most of the *node accessor* methods defined by the XQuery Data Model specification. Critically, there are no `children()` and `attributes()` accessors for navigating into the node hierarchy.
4. XQC does not provide a way to specify collations or the default collation. (Zorba, for example, provides this functionality via an extension interface.)
5. XQC version 1 provides no mechanism for implementing external XQuery functions. (Again, Zorba also provides this functionality via an extension interface.)

We believe that XQC in its current form represents a useful and usable standard, and look forward to significant expansion and refinement in the future.