XQuery in the Browser reloaded

Riding on the coat-tails of JavaScript

Thomas Etter
ETH Zurich
<etterth@student.ethz.ch>

Peter M. Fischer
ETH Zurich
<peter.fischer@inf.ethz.ch>

Dana Florescu
Oracle
<dana.florescu@oracle.com>

Ghislain Fourny
ETH Zurich
<gfourny@inf.ethz.ch>

Donald Kossmann
ETH Zurich
<donaldk@inf.ethz.ch>

Abstract

Over the years, the HTML-based Web has become a platform for providing applications and dynamic pages that have little resemblance to the collection of static documents it had been in the beginning. This was made possible by the introduction of client-side programmable browsers. Because XML and HTML are cousins, XML technologies can be almost readily adapted for client-side programming. In the past, we suggested to do so with XQuery and implemented it as a plugin. However, using a plugin was seen as an insurmountable obstacle to a wider adoption of client-side XQuery.

In this paper, we present a version of XQuery in the Browser without any plugin, needing only JavaScript to interpret XQuery code. This enables use even on mobile devices, where plugins are not available. Even though our current version is still considered to be at an alpha stage, we were able to deploy it successfully on most major desktop and mobile browsers. The size of the JS code is about 700KB. By activating compression on the web server (reducing the transfered data to less than 200 KB) as well caching on the client using the XQuery engine does not cause noticable overhead after the initial loading.

In addition, we are already reaching a large level of completeness and compliance, more than 98.6 percent correct tests at the 1.0.2 XQuery Test Suite. We have not yet done formal testing on Update and Full text, but plan to do so in the near future.

Keywords: XML, XQuery, Browser

1. Motivation

Currently, there is a growing perception that the Web and XML communities are drifting apart. One general concern is that up-to-date XML technologies (such as XSLT 2.0 or XQuery 1.0) are not seeing any support in the browsers, thus negating much of their potential.

Web pages are based on HTML. XML and HTML are both derived from SGML, and therefore have many similarities. This is why programming languages for XML can also be used for HTML with some adjustments. We have proposed the use of XQuery as a client-side programming language. XQuery seamlessly supports HTML navigation and updates, and as it is already used on the server (business logic, database querying), moving code between the layers is almost straightforward.

Last year, at XML Prague 2010 [8], the XQuery in the Browser plugin [7] was presented as a possible solution. It provides full XQuery support on the client side by embedding the Zorba XQuery engine [6] into a number of contemporary browsers. While the applications and usability were convincing, using a (binary) plugin was seen as insurmountable obstacle to a wider adoption, since even well-established plugins like Flash or Java are no longer available on major platforms, e.g. on the growing number of mobile devices.

Instead, browser vendors have been investing significantly into the quality of their JavaScript implementations [1] [3] [9], achieving orders of magnitude better performance. As a result, JavaScript has become a viable platform for implementing XQuery. Since writing an XQuery engine from scratch is a major effort, we opted for translating MXQuery [4], an existing Java-based engine, using Google's Web Toolkit [2].

A similar, albeit independent approach has been taken by Michael Kay [10]. The target language is XSLT 2.0 instead of XQuery, yet the overall approach, design and results are very similar to ours.

2. Current approaches for client-side programming

2.1. Container-based approaches: Java, Flash, Silverlight

For a long time, the most popular approach of programming complex applications in the browser has been to use a self-contained runtime environment like Java, Flash

or Silverlight. While such an approach provides high performance and effective developer support, it does not integrate well with HTML.

To make matters worse, the runtimes have to be downloaded, installed and updated separately on most platforms. On most mobile devices they are not available at all, on desktop system privileged user rights are often required for installation.

2.2. Javascript: DOM, Events, Frameworks

JavaScript is nowadays by far the most commonly used programming language for client-side website programming. Its greatest advantage is that it is available in all modern browsers. Because of the popularity of JavaScript, a lot of resources in browser development go into optimizing its execution speed [1] [3] [9] and in the last few years impressive performance improvements have been achieved.

Another advantage is being able to manipulate the homepage directly. Unlike in XSLT, where the transformation operates outside the current page, JavaScript allows editing the Web site directly through the DOM (Document Object Model).

Example 1. Javascript embedded in HTML page

```
<html>
  <head>
    <script type="text/javascript">
      window.onload = function(){
        var a = document.createElement('div');
        a.textContent = 'some text';
        document.body.appendChild(a);
      }
    </script>
  </head>
  <body>
  </body>
  </html>
```

This Web site will display "some text". The user interaction with the browser is made accessible by the so-called "DOM events", e.g. a button press, mouse movement or keyboard input. JavaScript can listen to these events and trigger actions based on them.

Example 2. Listening for an event

```
var button = document.getElementById('button1');
button.onclick = function (){
    window.alert('button 1 was pressed');
}
```

Furthermore, data can be downloaded, as long as it comes from the same domain.

Example 3. Retrieving a document using XMLHttpRequest

```
var req = new XMLHttpRequest();
//the last argument determines wheter the request asynchronously
req.open('GET', 'http://www.example.org/file.txt', false);
req.send(null);
if(req.status == 200)//HTTP OK
    window.alert(req.responseText);
```

While the combination of DOM manipulation, event handling and (background) downloads provides a powerful basis for rich Web applications, these APIs are at a quite low level of abstraction and not fully standardized across browsers. Therefore libraries which hide many of the compatibility issues and provide higher-level APIs as well as UI components have gained popularity, examples are jQuery or Dojo.

2.3. Cross-Compilation: Google Web Toolkit

The Google Web Toolkit (GWT) provides a full framework for creating web applications which allows a developer to write most of the code in Java. It offers widgets and RPC mechanisms, but the main innovation is a Java to JavaScript Compiler. GWT implements a subset of the Java standard library in JavaScript, thus allowing reuse of code on both the client and server side. Means for dealing with multiple browser versions are also integrated in GWT.

As our approach makes use of GWT to compile our code to JavaScript, we will describe GWT in more details in Section 4.1.2.

2.4. XML-based approaches: XSLT and XQuery in the Browser

2.4.1. XSLT

XSLT (eXtensible Stylesheet Language Transformations) is a declarative language to transform XML documents. While there is support for XSLT in many browsers, it suffers from several drawbacks, making it unsuitable as a general, complete solution for Web client developement. The main problem is that current browsers only support XSLT 1.0, often even in incompatible dialects. Since XSLT 1.0 is more than 10 years old, a lot of important functionality available in XSLT 2.0 is missing. In addition, XSLT in the browser runs completely independently from the Web site it belongs to. An XSL Transformation just receives an XML node and outputs an HTML/XML document or text, depending on the output type set, but does not interact with the Web site.

Example 4. An XSL transformation

```
<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
    version="1.0">
    <xsl:template match="/">
        <xsl:copy-of select="//b/text()"/>
    </xsl:template>
</xsl:stylesheet>
```

In the browser, an XML Transformation can be launched from a JavaScript program.

Example 5. A JavaScript sample which will execute an XSL transformation

```
//xsl holds a stylesheet document
//xml holds the XML document
xsltProcessor=new XSLTProcessor();
xsltProcessor.importStylesheet(xsl);
resultDocument = xsltProcessor.transformToDocument(xml);
```

Another variant to use XSLT in the browser is by including

```
<?xml-stylesheet href="stylesheet.xsl" type="text/xsl" ?>
```

at the beginning of the page, then the page itself is being used as the input for the transformation and it will take the output as the new page. This variant only runs the transformation once when the page is loaded and therefore offers no means for changing the page after it has loaded.

2.4.2. XQuery in the Browser plugin

Last year, we presented the latest release of our XQIB browser plugin [8]. It offers the functionality of XQuery inside the browser. Unlike the plugins presented before, it is not just a box inside a browser, but integrates seamlessly into the HTML DOM. It is possible to manipulate the website using all updating XQuery operations (insert, replace, remove, rename). It also allows subscribing event handlers to DOM events.

While XQIB combines XML technologies, declarative programming and browser/DOM interaction, it suffers from being a plugin: it needs to be manually installed and will only have limited availability.

3. XQuery in the Browser, JavaScript Edition: API and Functions

This section presents the API we suggest for programming browser applications. In general, most functions of XQuery 3.0, XQuery Update Facility and XQuery Scripting Extension are supported.

3.1. An example

Here is an example of a page containing XQuery code.

Example 6. Simple XQIB example

```
<html xmlns="http://www.w3.org/1999/xhtml">
  <head>
    <script type="text/javascript"
        src="mxqueryjs/mxqueryjs.nocache.js"></script>
    <script type="application/xquery">
      module namespace mod = "http://www.example.com/module";

      declare sequential function module:messagebox($evt, $loc) {
        b:alert(//div[1])
      };
    </script>
    <script type="application/xquery">
      import module namespace my = "http://www.example.com/module";

      b:addEventListener(b:dom()//input, "onclick", xs:QName("my:messagebox"))
    </script>
  </head>
  <body>
    <div>some text</div>
    <input type="button" value="Button"/>
  </body>
</html>
```

This will display a page containing a button. When pressing the button, it will display a message box with "some text".

3.2. Program structure

XQuery code can be included in HTML script tags with a MIME type `application/xquery`. A page may contain several of these tags, and each of these may contain either a main module or a library module. The previous example is a page with a main module and a library module. The main module imports the library module.

The semantics of a library module is that it is made available in the context. Any other module on the page may import it. We also allow importing modules from an external location instead of declaring in a script tag. In this case, a location hint needs to be specified, and the same-origin policy will be obeyed:

```
import module namespace m = "http://www.xqib.org/module" at "module.xquery";
```

The semantics of a main module is that it will be evaluated as soon as the page has finished loading. The XDM returned by the query body is then inserted in the

page after the script tag. Of course a main module may define functions (local namespace), but their scope will only be this module itself, and they cannot be imported by other module.

For each module, the XHTML namespace is set as the default element namespace. A built-in module with browser-specific functionality, the browser module, is also imported.

3.3. Browser-specific functions

We have defined the following browser specific functions in the namespace `http://xqib.org/browserapi` with the prefix `b:`

Table 1. Functions in the browser namespace

Signature	Side-effecting?	Semantics
`b:dom() as document()`	no	returns the currently displayed document
`b:getStyle($what as element(),$stylename as xs:string) as xs:string`	yes	returns the value of the style with the name `$stylename` of element `$what`
`b:setStyle($what as element(), $stylename as xs:string, $newvalue as xs:string)`	yes	sets the style with the name `$stylename` of element `$what` to `$newvalue`
`b:addEventListener($where as element()+, $eventname as xs:string, $listener as xs:QName)`	yes	Adds an Eventhandler to the element(s) `$where`, which listens for the event with the name `$eventname`. When the event is fired, it will call the function with the QName `$listener` and arity 2
`b:removeEventListener($where as element()+, $eventname as xs:string, $listener as xs:QName)`	yes	Removes an event listener previously added through `b:addEventListener` from the element(s) `$where`
`b:alert($message as xs:string)`	yes	Displays a message box with the content `$message`

Additionally, we also support the EXPath HTTP library for data retrieval. It has the same limitations as `fn:doc` which are described in the next paragraph.

3.4. Functionality with different semantics

As there is no file system in the browser, the semantics of module import with a file location hint and of `fn:doc($uri as xs:string)` are defined as follows. If a relative URI is provided, the library will download the file automatically using an XMLHttpRequest.

If an absolute URI is given, the library will also try to retrieve it. This may fail due to security constraints, namely the same-origin policy, which only allows requests coming from one page to access the same host at the same port using the same protocol as was used on that page. This policy may be circumvented with HTTP headers.

As we use the JavaScript classes for regular expressions, not all options are supported. The option for dot-all matching is currently not available because it is not supported by the JavaScript RegEx classes, but an emulation will probably be provided in the future.

3.5. Not implemented functionality

We have chosen not to implement schema support to keep the code smaller, and because we were not aware of any client-side schema validator when the project was started.

Also missing is the function `fn:normalize-unicode`, because neither GWT nor any third-party libraries provide support for it with reasonable code footprint.

At the time where this paper is written, the newer main/library module API is not yet implemented, but we are working on it. Samples are available at [5].

4. Implementation

4.1. Background

4.1.1. MXQuery

MXQuery is an XQuery engine written in Java with the goal of good portability and a small footprint, targeting also mobile and embedded devices. It supports a broad range of XQuery-related standards, including XQuery 1.0, the Update Facility, Fulltext and parts of XQuery 3.0 and Scripting. It is implemented in Java and available as open source with an Apache 2.0 License.

The design of MXQuery is inspired by query processing/database architectures, using an iterator-style query plan generated by a parser/optimizer frontend. Therefore, it is well suited for streaming execution, keeping the runtime memory needs low unless blocking or nested operations require materialization.

4.1.1.1. Tokenstream-based Data Model

MXQuery uses a Token model for its XDM representation, similar to the BEA/XQRL XQuery engine. Tokens are conceptually very similar to parse events (SAX, StAX), but are objects which directly carry all relevant information instead of a parse type identifier.

4.1.1.2. Iterators

All functions, XPath selectors and other operators in MXQuery are implemented through iterators. An iterator takes as input zero (an iterator can return a constant value) or more iterators and outputs a token stream. Iterators are then combined into a tree/DAG to represent the query.

4.1.1.3. Stores

While MXQuery tries to stream tokens as much as possible, in many cases additional functionality is needed, such updates, full text or stream storage. A store provides an interface for getting an iterator for an XDM instance and for additional operations like indexed path access, applying updates or fulltext retrieval. XDM tokens in MXQuery carry a link to their store in their node IDs (see also Section 4.2.3).

4.1.1.4. Customizability and Platform Abstractions

In order to easily adapt MXQuery for environments with restricted processing resources, several components can be detached: Most of the functions and extension modules are loaded dynamically. Stores for particular functionality (streams, fulltext, updates) are instantiated on demand and can be omitted when this functionality is not needed. Furthermore, even the parser/compiler frontend can be removed if not interactive compilation is needed. Within the scope of this prototype, we have not exercised these options. If an additional code/functionality reduction is required, some of these options will become useful.

Since the Java language space has become fragemented over various version of the Micro Edition (J2ME), the Standard Edition and newer Google proposals such as Android/Dalvik, Google App Engine and GWT, MXQuery aims to use a subset of language expressions and library classes that is broadly available. At the beginning of the XQIB-JS project, MXQuery had an abstraction layer that hid the differences between J2ME, J2SE 1.4 and J2SE 1.5/1.6. As it turned out, Google has chosen to make GWT a different Java subset than J2ME, so several of the abstractions were not sufficient.

4.1.2. Google Web Toolkit (GWT) Fundamentals

GWT provides a Java API (as a subset of J2SE) which exposes several browser features, most prominently DOM access to the browser. The application and the required libraries need to be available as Java source code. This Java code is then translated into Javascript, creating by default a monolithic code file. Since the available functionality varies from browser to browser, several different versions of the code are generated. At runtime, a small Javascript file `projectname.nocache.js` has to be included in the Web site which will detect the browser type and download the actual application Javascript code file, encapsulated in an HTML file. This file is loaded into an IFrame using an XMLHttpRequest and inserted into the current page using the JavaScript function `document.write`. Therefore the same-origin policy applies, preventing central hosting of an GWT-based library. Furthermore, this function is only available in HTML and therefore GWT and our library can currently not be used on XHTML pages.

4.2. Selected Implementation Aspects

When translating MXQuery using, we initially encountered various compiler errors (see Section 4.2.6) due to the library differences to either J2ME or J2SE. We solved them by removing functionality and gradually reintroducing it until we could run some basic queries. Once we had a version that compiled, we added the integration with the browser rewrite/added code to re-enable the remaining missing features.

4.2.1. Solution architecture

Similar to the original XQIB plugin, we use the *Store* abstraction as a means to encapsulate browser-specific aspects, as shown in Figure 1. From an MXQuery point of view, the DOM is yet another store that supports full path navigation and updates. This store is created when the `b:dom()` function is present in the code. If other XDM instances are needed, normal MXQuery stores are generated.

4.2.2. Mapping the DOM to MXQuery-tokens

The DOM (Document Object Model) is the fastest way of accessing the data in a browser. A website is mapped into a DOM tree when it is loaded, which can then be accessed through JavaScript.

The XHTML DOM and the HTML DOM have some small differences. XHTML is naturally namespace-aware because it is derived from XML, whereas HTML is not. To make the DOM implementations compatible again, the W3C has specified that all nodes in a HTML document should be in the XHTML namespace `http://www.w3.org/1999/xhtml/`. While node names in XHTML are defined to be lowercase, the standard implementation in HTML for `node.nodeName` has always

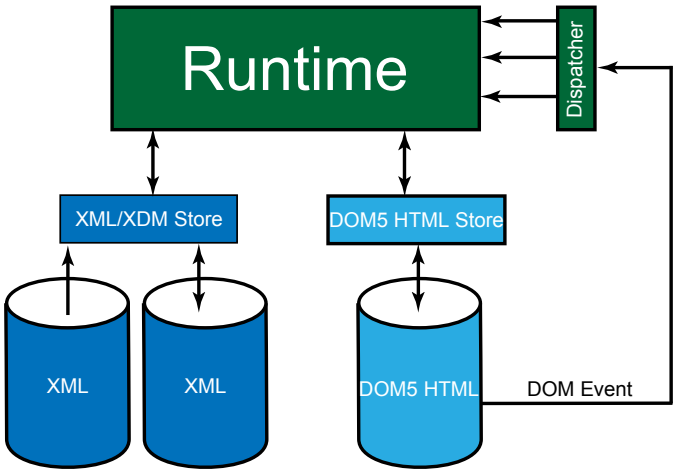

Figure 1. XQIB Architecture Overview

been to return an uppercase value. To avoid breaking existing code, this is how the attributes are defined in HTML5.

Table 2. DOM5 HTML attributes and their use for traversal

Function	Semantics	Used for
`String node.namespaceURI`	hardcoded to `http://www.w3.org/1999/xhtml/`	retrieving the namespace of an attribute/element
`String node.localName`	node's name without namespace in lowercase	retrieving the name of an attribute/element
`String node.nodeName`, `Stringelement.tagName`	returns node.localName in uppercase	not used
`NamedNodeMap node.attributes`		retrieving the attributes of a Node
`Node node.firstChildNode node.nextSiblingNode node.parentNode`		axis navigation

Therefore, we can just use `node.localName` and `node.namespaceURI` to get the same behavior without difference between an XHTML and an HTML document.

An important decision in the design of XQIB was the linking between the DOM as a materialized, updateable tree and and the stream of immutable tokens. Instead of building a "shadow" structure of tokens that are generated when traversing the DOM, but is no longer connected, each token links explicitly back to its orginating DOM node, and also draws as much information as possible from there instead of copying it, as shown in Section 4.1.1.1. By doing so, the tokens stay in sync with the

DOM, and we only create them when there is a specific access to the DOM, e.g. by navigating a specific axis. We changed to Token implementation of MXQuery to support this new kind of Token alongside to the existing, "eager"/"standalone" tokens.

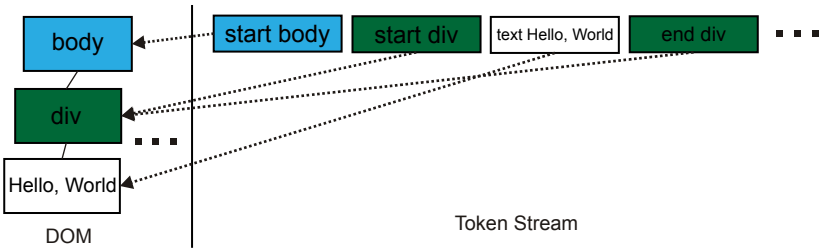

Figure 2. Mapping HTML DOM to a lazy Token Stream

4.2.3. Node IDs

XDM mandates that every node has a unique identifier, which may also encode the document order or provide structural informations, e.g. parent-child relationships. Node IDs may not change during the execution of a query. In the standalone application, MXQuery generates IDs for the nodes while generating XDM from input or constructing new elements, typically using ORDPath Dewey IDs.

We also considered this approach for the DOM5 HTML wrapping , but then decided to utilize existing means from the DOM in order to avoid the overhead of generating new identifiers and the effort of keeping the DOM and the XDM identifiers synchronized

DOM nodes in JavaScript are uniquely identifiable using the `Node` interface. We therefore can compare two nodes for equality using the Java == operator. It will correctly tell us whether they are referencing the same node no matter how we retrieved the references. To compare by document order, the DOM nodes can also be used. The DOM Level 3 Core offers the function `compareDocumentPosition` which allows to compare two nodes directly. Where this interface is not available we can fall back to an alternative method, based on the Lowest Common Ancestor (LCA). The LCA can be found by walking through the tree up to the root while memoizing intermediate nodes. From the root node, we then compare the intermediate node lists of the two nodes. The LCA is the last node which we can find in both lists. If one of the nodes is the LCA, we know the document order. Otherwise we can look at the children of the LCA and determine which path comes first.

Using the nodes themselves as IDs, we can reduce runtime overhead and when `compareDocumentPosition` is available, we also have a very efficient way of ordering nodes.

4.2.4. Applying Updates

When we have an updating query, we get as result of the query a PUL (Pending Update List), containing all updates that need to be applied. Performing these updates on DOM5 HTML in a correct and efficient way needs some special considerations: Deletion is the most straightforward primitive, since we can just call the node's `removeFromParent()` function. For insertion, the token stream needs to be converted into DOM nodes. It is very important not to insert an element into the DOM before all its children have been generated, since changes to the DOM cause a browser to trigger a time-consuming and possibly user-visible repaint. Replaces are done by inserting the source before the target, followed by removing the target. This way, the implementation for replace is very short and we avoid code duplication. Renaming of elements is a feature which is not natively supported by the browser's DOM, so we use a workaround: We first create a new node with the desired name. Second, we set all attributes from the old node on the new node. Third, we set the parent of all children of the old node to the new node. Finally, we replace the old node with the new node.

4.2.5. Handling Events

In order to be able to use XQuery in the Browser to implement interesting web applications, we need access to the DOM events. These include keypresses, mouseclicks and similar functionality. For this, we provide functions to add and remove event handlers in our browser namespace `b:` (more about that in Section 3.3). To add a function to handle certain events on an element, we need three arguments: the elements on which the event should be observed, the event name and the identifier/name of a function which will be called when the event is triggered. Since MXQuery does not support Function Items yet, we have opted to take QNames as function identifers.

For a complete sample see Section 3.2. We will now take a look at what happens when we register an event handler.

```
b:addEventListener(b:dom()//input, "onclick", xs:QName("local:somehandler"))
```

Inside XQuery in the Browser, there is one DOM event handler function which handles all incoming DOM events and dispatches them to the relevant XQuery functions. It also aggregates the subscriptions from the XQuery code and pushes the relevant subscriptions into the DOM, so that only the necessary events are being generated. To keep track of which event handlers are currently registered, there is a Hashmap, which has the following declaration:

```
static HashMap<NodeAndEventName, List<String/*a QName*/>> handlers;
```

NodeAndEventName is a helper class to provide means to use a node reference and an event name as a key. So when we get an event in our universal event handler,

we get the target node and the event name from the DOM event object. Then all functions in the list are invoked.

An advantage of having only one function is that it is easy to unsubscribe events. When e.g.

```
b:removeEventListener(b:dom()//input,"onclick","local:somehandler")
```

is called, the universal event handler gets the list of handlers for the pair (element, "onclick"). It removes "local:somehandler" from the list. If the list is empty now, we can remove the event handler in the browser using the JavaScript function `element.removeEventListener`.

To resolve the function name and to call a function, we utilize the the runtime function call mechanism of MXQuery, treating the pre-compiled query like a module.

4.2.6. Compatibility issues

4.2.6.1. Missing functionality in GWT

For many browser functions, GWT only supports the functionality available on all browsers. This reduces many functions for DOM access to the level of Internet Explorer 6. This forced us to rewrite or extend DOM classes like Node or Element with functionality such as `element.setAttributeNS` or `element.localName`.

4.2.6.1.1. Missing Java Functions

In GWT, we do not have a Java VM (Virtual Machine), thus the class-loading ability is missing. The mainline MXQuery relies heavily on on-demand loading of classes for functions and operators, in particular for extensibility. We needed to hardcode all functions as a big Java file, transformed from the XML metadata files used in MXQuery.

Another missing area are I/O classes. There is no implementation of `OutputStream`, `PrintStream`, etc. available, so we had to include these classses from the Apache Harmony project.

Furthermore, no`Calendar` class is included in GWT.This class backs most date and time operations in MXQuery. The Calendar classes from the Apache Harmony project depend on IBM's internationalization library (ICU) which is nearly impossible to port to GWT. Therefore we first used a third party `Calendar` emulation, `gwt-calendar-class`, which uses the GWT internationalization features to provide the full functionality provided by Java's `Calendar`. This class fixed a lot of tests, but included information for all time zones, which increased the file size by 300 KB and increased the loading time by more than a second.

Fortunately, the XPath Data Model (XDM) does not require daylight savings time awareness. So the exact time zone does not matter to an XQuery program, only

the time offset to the UTC. We therefore kept the API of `gwt-calendar-class`, but re-implemented the relevant functionality, significantly reducing the code footprint and the initialization delay.The current implementation is very close to the XDM specification and has good test conformance (see Section 5.1.2).

This will need to be further improved to support conversion to the end user's time zone. Also, the implementation is leap second-unaware and will therefore have small errors when computing durations.

4.2.6.2. Deviating from the standard

In some cases, fully conforming to the standard would increase the download size a lot and the performance would decrease drastically. One of these cases are regular expressions. GWT does not provide the `java.util.regex` package because it would be difficult to implement it correctly. As JavaScript already provides regular expression functionality, we just used that one. It may not offer all functions (e.g. the dot-all option is missing), but its performance is much faster than an implementation in JavaScript because it can be optimized by the browser supplier.

The syntax of a conforming regular expression implementation could also be simulated. If we take the missing dot-all option, this could be emulated by replacing all dots in the search string with `[\s\S]`.

5. Evaluation

5.1. XQuery Standard Compliance

For testing an XQuery implementation, the W3C provides the XQTS (XQuery TestSuite), an extensive testing framework. We used the version 1.0.2 for our tests, because it was the newest one available when we started the project.

5.1.1. Testing with the XQTS

The XQTS 1.0.2 consists of 15133 tests, covering by mandatory and optional features of XQuery 1.0 Each test is in its own file and there is one or more valid outputs for each test. There are three main categories of tests: standard, parse-errors, runtime-errors. Standard tests have to return a result, which can then be compared to the expected result. The error tests expect a certain error code which indicates what triggered the error.

These tests are described in an XML file. Because we wanted continuous testing, we had to find a way to automate it. GWT supports JUnit tests and our build server (Hudson) also offers good JUnit support. For these reasons we converted the XML file to JUnit test cases using an XSLT stylesheet.

Another problem remained: GWT runs tests in a windowless browser environment. It is not possible to load any data from outside due to the same origin policy which should prevent cross-site scripting. Therefore we had to integrate all data into the the tests and could not load anything from outside.

5.1.1.1. Test performance optimizations

For each test, the whole browser was restarted, what took several seconds per test class. So to optimize it, instead of running a lot of test classes, we combined them using the `TestSuite` class provided by JUnit. This way, we were able to run the whole XQTS in under an hour.

Because we wanted to have it still faster, we saw that parsing the XML from strings was a bottleneck. Therefore we cached the token stream. This reduced the combined build/test time to under 20 minutes. This is a good result considering that also the "native" Java version of MXQuery uses 3 to 4 minutes for the test suite.

5.1.2. Testing results

Our goal was of course to be 100% compatible. In this section, we will evaluate how far we came to achieving this goal and the limitations of the platform or our design choices.

Among the minimal compliance test of XQTS 1.0.2, we currently pass 14433, giving us 98.6 percent conformance. In addition, we pass all Use Case and Full Axis tests.

Of the 468 total cases which we do not pass, we skip the following:

- The unicode character assemble and disassemble tests (89) do not compile because the GWT compiler has trouble finding the correct encoding
- We have implemented fn:doc() to load documents from the url given using XMLHttpRequest, but while testing, we cannot load anything due to the same origin policy as mentioned in Section 3.4.
- Similarly, we cannot test module import as it also depends on file access.
- We do not support `normalize-unicode()`, which accounts for 34 tests. Similarly, we do not support 197 test cases are schema related and 46 on static typing.

The tests cases that are actually failing are distributed across diverse test groups. Some problems arise from the fact that the catalog XML file is over 10 megabytes in size, giving us an XML parser error when executed in the browser. There are some remaining problems with date arithmetics and time zones.

5.2. Supported Platforms

XQuery in the Browser runs on all modern and standards-compliant browsers. These are namely (later versions should always work): Firefox 3.6 and 4 (also Mobile), Google Chrome 7, Internet Explorer 9, Safari 5, Opera 11 (also mobile), the Android browser (mobile Chrome), the iPhone browser (mobile Safari)

We do not yet support Internet Explorer 8 or lower because it has a much lower compliance to W3C standards.

Due to the limitations of GWT (using `document.write`), XQIB currently handles only HTML and HTML5 DOMs but not XHTML DOMs - which is somewhat ironic, given that XHTML is conceptually conceptually much closer to the XML world.

5.3. Performance

5.3.1. Runtime

Please note that these numbers are all approximated as there are extreme variations when testing. The dominant numbers for a certain browser have been chosen. They where taken on a system with average performance for the year 2010 (Phenom II X6 2.8 GHz, Windows 7 x64). In order to eliminate network overhead, the page was served on localhost using Apache.

First, some performance numbers from a simple test page:

```
<!DOCTYPE html>
<html xmlns="http://www.w3.org/1999/xhtml">
  <head>
    <title>XQIB: Sample page</title>
    <meta charset="UTF-8"/>
    <link href="style.css" rel="stylesheet" type="text/css">
    <script type="text/javascript">
        var time_start = Number(new Date());
    </script>
    <script type="text/javascript"
        src="mxqueryjs/mxqueryjs.nocache.js"></script>
    <script type="application/xquery">
        insert node
          <div>{let $x := b:dom()//h1 return xs:string($x)}</div>
        as last into b:dom()//body
    </script>
  </head>
  <body>
  </body>
</html>
```

The time is measured from the first script tag to the end of the query execution. Therefore some final rendering might not be measured.

Table 3. Load times

Browser:	**Firefox 3.6**	**Firefox 4**	**Chrome 8**	**Internet Explorer 9**
Time (ms):	230	200	120	140

These values seem very high, but when considering that there are many optimizations which can be done, they are quite good. We have to take into consideration that we are analysing load times, which are usually dominated by bandwith/latency constraints. While the script is running, a browser can continue to download images included on the page.

To test events and dynamic pages, we have also tested a modified version which does about the same as the first one, but triggered by a mouse click.

```
<!DOCTYPE html>
<html xmlns="http://www.w3.org/1999/xhtml">
  <head>
    <title>XQIB: Sample page</title>
    <meta charset="UTF-8"/>
    <link href="style.css" rel="stylesheet" type="text/css">
    <script type="text/javascript"
        src="mxqueryjs/mxqueryjs.nocache.js"></script>
    <script type="application/xquery">
      declare updating function local:handler($evt, $loc) {
        insert node
          <div>{let $x := b:dom()//h1 return xs:string($x)}</div>
        as last into b:dom()//body
      };
       b:addEventListener(b:dom()//input,"onclick",xs:QName("local:handler"))
    </script>
  </head>
  <body>
    <input type='button' value='Button'/>
  </body>
</html>
```

The time is measured from the beginning of the event handler until the event handler returns.

Table 4. Script execution times

Browser:	Firefox 3.6	Firefox 4	Chrome 8	Internet Explorer 9
Time for first run(ms):	15	15	30	30
Time for subsequent runs(ms):	15	15	5	10

When executing the same script as triggered by an event, we get different results. The execution is now much faster on all browsers. This demonstrates that the performance, even at this early stage, is already sufficient for dynamic websites.

5.3.2. Download Size

For loading a page, two files have to be loaded: First the dispatcher file "mxqueryjs.nocache.js" which is under 6 kB in size. This will then select the file with the actual code depending on the browser version. This file is about 700 kB in size. By enabling gzip compression on the server, the transferred data can be reduced to 200 kB. In addition, this code can be cached, making subsequent access (almost) instantaneous.

6. Conclusion

We have shown that it is possible to build an XQuery engine on top of JavaScript without major performance or functionality penalties. This allows to use XQuery for browser programming. XQuery already has a large user base which comes mainly from the database and XML communities. This enables them to write web applications in a language which is familiar to them.

7. Future work

We consider the following directions for future work

- Integration of the Javascript branch back to MXQuery mainline for better long-term maintenance.
- Improved browser support, investigating if Internet Explorer 8 might be a feasible target given its high market share.
- Performance optimizations, in particular fully using indexed access to the DOM.
- Integration of JSON (e.g. like the upcoming `parse-json()` function in XSLT 3.0 and the ability to call Javascript and be called by Javascript.

- Truly asynchronous HTTP and the ability to access more browser state such as headers, cookies or indexed storage.
- Further streamlining, modularization and dynamic loading as well as the ability to centrally host the library.

Bibliography

[1] Google Chrome's Need For Speed http://blog.chromium.org/2008/09/google-chromes-need-for-speed_02.html

[2] Google Web Toolkit http://code.google.com/webtoolkit/

[3] The New JavaScript Engine in Internet Explorer 9 http://blogs.msdn.com/b/ie/archive/2010/03/18/ the-new-javascript-engine-in-internet-explorer-9.aspx

[4] MXQuery XQuery Engine http://www.mxquery.org

[5] XQuery in the Browser Web site, JavaScript Edition samples http://www.xqib.org/js

[6] Zorba XQuery Processor https://www.zorba-xquery.com/

[7] G. Fourny, M. Pilman, D. Florescu, D. Kossmann, T. Kraska, D. McBeath: XQuery in the Browser, Proceedings of the 18th International Conference on World Wide Web, WWW 2009, Madrid, Spain, April 20-24, 2009 2009.

[8] G. Fourny, M. Pilman, D. Florescu, D. Kossmann, T. Kraska, D. McBeath: XQuery in the Browser (talk), XML Prague 2010, March 12-13, 2010, Czech Republic.

[9] A. Gal, M. Franz, Incremental Dynamic Code Generation with Trace Trees, Technical Report No. 06-16, Donald Bren School of Information and Computer Science, University of California, Irvine; November 2006.

[10] M. Kay, Compiling Saxon using GWT http://saxonica.blogharbor.com/blog/archives/2010/11/16/4681337.html