Ten Reasons Why Saxon XQuery is Fast

Michael Kay
Saxonica Limited
Reading, Berks, UK
mike@saxonica.com

Abstract

This paper describes the internal features of the Saxon XQuery processor that make the most significant contribution to its speed of execution. For each of the features, an attempt is made to quantify the contribution, in most cases by comparing performance achieved when the feature is enabled or disabled.

1 Introduction

Saxon [1, 2] is an implementation of XQuery written in Java. It implements the XQuery 1.0 specification [3] in full, with the exception of the static typing feature (see [3], section 5.2.3), but including support for schema-aware processing. It also implements the XQuery Update specification [4], which is currently a W3C Candidate Recommendation.

Saxon also implements XSLT 2.0 [5], XPath 2.0 [6], XML Schema 1.0 [7], and a significant subset of the new features in the draft XML Schema 1.1 specification [8]. In fact Saxon started life as an XSLT processor, and was later adapted to handle XQuery as well. The two languages are implemented as different syntax front-ends to the same run-time engine; both compilers generate the same code and at run-time there is essentially no knowledge of whether the code originated as XSLT or XQuery.

Saxon is available in several versions. The open-source product, Saxon-B, implements all the mandatory features of the W3C specifications. The commercial version of the product, Saxon-SA, provides additional optional features, including schema processing, schema-aware XSLT and XQuery processing, and XQuery Update, as well as a number of performance-oriented features including a more advanced query optimizer, support for streamed query execution, document projection [9], and Java code generation.

Saxon is released on both the Java and .NET platforms. The code is written in 100% pure Java. The .NET version is created by cross-compiling the Java bytecode into .NET IL code, using the open-source IKVMC cross-compiler [10]. The version described in this paper is Saxon-SA 9.1 on the Java platform, unless otherwise specified.

Saxon has been under development for over ten years, and the size of the code base is now some 180,000 non-comment lines, excluding test material and tooling. The development objectives for Saxon are, in order of priority: **(1)** Rigorous standards conformance; **(2)** Reliability; **(3)** Usability (primarily of interfaces and error messages); and **(4)** Performance.

Bulletin of the IEEE Computer Society Technical Committee on Data Engineering

While this paper is concerned with performance, it is important to note at the outset that performance goals are never achieved by sacrificing the higher-priority objectives. In practice, while the objectives are sometimes in conflict, it has in nearly all cases proved possible to achieve the required performance without compromising other goals. For an example see [11].

It is not the intention of this paper to compare the performance of Saxon with other XQuery processors. It is impossible to do this objectively when one knows one product much better than the others. A number of papers have been published describing comparative benchmarking of different XQuery processors [12, 13, 14]. Independent benchmarks can be frustrating for a vendor because they exhibit a lack of specialized knowledge on how to get the best possible results from one's own product; also in the case of Saxon, they often use the open-source version rather than the higher-performance commercial version. Nevertheless, the overall conclusion from these independent studies is that Saxon performance, while not always in pole position, is comfortably near the front of the field.

Another problem with benchmarks is that performance is not a one-dimensional objective. Some users are interested in the throughput of a transaction processing workload that handles thousands of small messages per second using the same queries. Others are interested in the elapsed time for processing very large documents. Some users generate queries on-the-fly, in which case query compile time can be as important as execution time. Some workloads are dominated by the cost of parsing source documents, some by serialization of results, others by the computational cost of the query itself. A well-rounded product needs to satisfy all its users, not just to optimize its score in a synthetic benchmark.

2 The Architecture of Saxon

There is no space in this paper to give a detailed account of the internal architecture of the Saxon product. An article [15] was published some years ago, and although it describes the product from an XSLT rather than XQuery perspective, the broad picture remains valid today.

It should be noted that Saxon is not a database product. Its raw material is XML held in unparsed form in filestore, or sent over the wire. This means that Saxon does not have the luxury of maintaining persistent indexes or collecting statistical data for use by its optimizer; it has to take the data as it comes. When a query is schema-aware, Saxon is able to take schema information into account when compiling a query, but the general rule is that queries are compiled with no knowledge of what will be found in instance documents.

Like every other implementation, the Saxon XQuery processor has compile-time and run-time processing phases. Broadly, the compiler works by creating an expression tree as the output of the parsing phase. It then performs type checking, which labels nodes in the tree with the results of static type inferencing, and adds additional operators to the tree to perform run-time type checking or conversion where required. Saxon works on the principle of *optimistic static type checking*, which means that a compile-time error is reported only if the inferred static type of an expression is disjoint with the required type; if the static type overlaps but is not subsumed by the required type, then additional code is generated to perform run-time type checking. Following type-checking, the next phase is optimization; this examines the tree for constructs that can be rewritten and replaced by alternative, hopefully more efficient equivalents. The optimization phase is optional, and where compile-time performance is critical it can safely be omitted or performed less aggressively.

The final optimized expression tree can then be used in two ways: it can be interpreted by the run-time execution engine, or it can be used as input to the Java code-generator. This generates Java byte code to execute the query directly (currently via Java source code as an intermediate form), and the byte code is then executed by the Java VM in the normal way. The byte code, of course, still makes many calls on a precompiled Saxon run-time library.

Saxon does not include its own XML parser; it can work with a variety of third-party parsers (both push and pull). It does however include its own schema processor and validator: close integration between the schema

processor and the XQuery engine was considered essential for high performance.

3 Performance Features

In this central section of the paper we examine a number of features implemented in Saxon whose aim is to improve query performance, and we attempt to quantify the impact of each feature.

3.1 The TinyTree and the NamePool

The XML document used as input to a query may be stored in a variety of ways; what these have in common is that they all implement the abstract Java interface `NodeInfo`. `NodeInfo` is essentially at the same level as the abstract XDM model described by W3C [16]; it differs however in that it offers direct support for the thirteen XPath axes (child, descendant, ancestor, following-sibling, etc). This allows each `NodeInfo` implementation to optimize the way it navigates each axis; and in the case of models that create node objects on demand, it also means that nodes are created only where the caller actually requires them, and not for intermediate nodes that end up being skipped.

There are two native implementations of the `NodeInfo` interface in Saxon: the linked tree, which is a conventional "object-per-node" tree structure in which parent nodes contain a list of their children, and the TinyTree, which we will describe in this section. There are also a number of implementations of `NodeInfo` that wrap external object models including DOM [17] (both Java and Microsoft versions), JDOM [18], DOM4J[19], and XOM [20]. A number of vendors integrating Saxon into other applications have written `NodeInfo` implementations to access other data sources.

The TinyTree structure is unashamedly inspired by the DTM model in Xalan [21], though it does not mimic the design at a detailed level. There are also some similarities with Intel's "record representation" [22], though a significant difference is that Saxon's structure represents nodes in the tree, whereas Intel's represents events in the parse stream.

The TinyTree represents a document using six principal arrays of integers. These arrays contain one entry for each node (other than attribute and namespace nodes), and are indexed by node number. They contain respectively: the node kind (for example element, text, comment), the *name code* (see below), the depth in the hierarchy, a next sibling pointer (which for the last sibling points back to the parent node), and two overlaid values which in the case of elements point to the first attribute and the first namespace node, and for other kinds of nodes are pointers to the textual content in a text buffer (or in the case of a whitespace-only text node, a representation of the actual whitespace compressed using run-length encoding). The total size of these six integers is 19 bytes per node. Attributes and namespaces are represented in separate but similar sets of arrays.

Additional arrays are allocated when needed. The first time a reverse axis such as preceding-sibling is used on a particular document, an array containing prior sibling pointers is created and populated. If the document is schema validated, an additional array is allocated to hold the type annotations produced as a result of the validation process (again as integers, using the name code of the type name).

The TinyTree is designed to be compact without sacrificing speed of access. In particular, it avoids the heavy overhead of using one Java object for each node in the tree; instead, `NodeInfo` instances are allocated as transient objects on demand, and are garbage collected when no longer needed. Modern Java VMs make garbage collection of short-lived objects a highly efficient operation. The TinyTree is also optimized for read-only access. It makes it very efficient to compare nodes for document order, a common operation in XPath. This structure does not support XQuery Update; for that, a mutable linked tree must be used.

Names of elements and other nodes are represented using an integer name code, which can be translated into a fully qualified name by reference to the `NamePool` holding all the names, which is used to allocated new codes. The name code contains a unique identifier for the URI/local-name pair in 20 bits, with a further 10 bits

used to represent the prefix; this imposes a limit of a million or so URI/local-name pairs, which as far as I know has never caused a problem, and a limit of 1024 distinct prefixes for each URI, which does occasionally cause problems for pathological applications; but we can live with that. The essence of the approach is that the same `NamePool` is used at compile time and at document parsing time, which means that the compiler can generate code that searches for named nodes using an integer comparison rather than a string comparison.

The primary motivation for the TinyTree is to reduce memory occupancy and building time for large documents without sacrificing access speed, while the main driver for the use of integer name codes is to improve the speed of matching nodes by name. We can evaluate both effects by comparing the TinyTree with both the Saxon linked tree (which uses an object per node, but with integer name codes) and with the DOM (which uses an object per node, and string comparison for names.) To do this I took the 100Mb version of the XMark dataset [23], modified to use a namespace to make it more typical, and ran the query `count(/ns:site//ns:from)` against it. This gave the results shown in Table 1:

Table 1: TinyTree performance

	TinyTree	Linked Tree	DOM
Build time	5136ms	7933ms	8332ms
Memory used	327Mb	370Mb	796Mb
Query time	35ms	226ms	10603ms

This was run with whitespace stripped from the tree, which makes a significant difference to the figures. The DOM used was the Xerces implementation bundled in JDK 1.5. It can be seen that although the TinyTree beats the linked tree on both time and space, the most noticeable gain is in search speed.

3.2 Pull/Push Pipelining

Pipelining is well established as an execution strategy for functional languages as well as for relational databases. The essence of the approach is that an operator that nominally takes a sequence as input and produces a sequence as output (for example the filter operator represented in XPath by the syntax A[B]), should read its input one item at a time and deliver its output to the parent operator one item at a time. This is a description of a pull pipeline: it is driven by read operations issued by the ultimate consumer of the data. Equally valid is a push pipeline, controlled using write operations issued by the supplier of the data.

Saxon uses a combination of pull and push pipelines, and choosing the right kind of pipeline at each stage appears to make a significant difference to performance.

Pull pipelines are used primarily for evaluating XPath expressions, that is, when reading from the source document. Push pipelines are used primarily when constructing documents (both the initial source document and the result document), and also when serializing. Saxon's schema validator is a complex push pipeline, as is the XML serializer. This split between pull and push was very natural in an XSLT 1.0 processor, where there is a clean split in which XPath expressions read the input and XSLT instructions write the output. In XQuery (and for that matter in XSLT 2.0), the two kinds of operation can be composed in arbitrary ways. Nevertheless there are two very different kinds of operation; and it remains true that many queries are "single-phase" in the sense that they only read nodes from the initial query input and only write nodes to its final output.

Feeding data from a pull to a push pipeline is easy: a program owning the control loop reads from the first pipeline and writes to the second. Doing the opposite is more challenging. In the absence of a language with intrinsic coroutine support, there can only be one control loop. Two solutions are available: either break the pipeline by building the intermediate sequence in memory, or use multiple threads. Both involve overheads. Saxon uses both techniques, though multiple threads are used only in one very specific situation, to support streamed processing where the source document is not built into an in-memory tree. So one of the main design aims is to use pull and push where appropriate, but while minimizing the need to switch from one to the other.

```
for $i in distinct-values(
          /site/people/person/profile/interest/@category)
let $p := for  $t in /site/people/person
          where  $t/profile/interest/@category = $i
          return <personne>
                 ...
                 </personne>
return <categorie><id>{$p}</id></categorie>
```

Figure 1: The XMark query $q10$

To achieve this, Saxon divides query operators into three categories:

- Simple read expressions are always executed in pull mode. These include path expressions and filter expressions, sequence concatenation, union/intersection, and function calls such as `subsequence`, `insert`, or `index-of`.
- Node constructors are generally executed in push mode: they write events to an output pipeline. This works especially well when the output is sent straight to a serializer; in this situation there is no need to materialize the constructed tree in memory. These instructions are also able to operate in pull mode (to deliver events on demand to a client), but this is only done if the application that fires off the query explicitly asks for the query result in this form.
- Other expressions, notably FLWOR expressions, conditional expressions, and function calls can operate in either push or pull mode. In general they operate in the same mode as their caller, so if they are invoked during tree construction they will push, and if invoked in the middle of a path expression they will pull. This means that a function body may execute in either mode depending on the context of the caller.

How can we evaluate the effectiveness of this strategy? As an illustration, XMark query $q10$ (see Figure 1), after rewriting by the Saxon optimizer to inline the unnecessary variable `$p`, is a classic one-phase query; run with default options it takes 1926ms, but if we force it to run in pull mode it takes 3456ms, largely because the result document is materialized in memory before being serialized. This query contains a FLWOR expression (for `$t`) that is logically inside an element constructor, and is therefore evaluated in push mode. If we artificially force the FLWOR expression into pull mode (by a tweak to the Saxon code), the execution time becomes 2720ms. Forcing the variable `$p` to be materialized rather than being pipelined also affects the performance adversely, this time to 2398ms. These figures should be sufficient to illustrate that the impact of pipelining decisions can be significant, though they do not prove, of course, that Saxon always gets it right.

3.3 Path Expressions

Path expressions in Saxon are evaluated using a nested loop strategy. A path expression such as `x/y/z` finds all the `x` children of the context node; for each of these it finds all the `y` children, and for each of these it finds all the `z` children. In case this seems obvious, it is not the strategy that all products use, and some researchers have expressed surprise that it should perform so well.

Because of pull pipelining, it is actually an inverted nested loop: the client requests the next `z` element, which might cause the next `y` to be found, and so on. Neither the final node sequence delivered by the path expression nor any intermediate results are materialized in memory.

The main optimization carried out by Saxon is to eliminate sorting wherever possible. The semantics require that the results of each "/" operator, and indeed the results of each axis step, are sorted into document order with

duplicates eliminated. In practice such sorting is very rarely needed because the nested-loop evaluation in many cases delivers results already sorted and deduplicated. Saxon goes to considerable trouble to avoid unnecessary sorting. Furthermore, even when the evaluation strategy delivers nodes in the wrong order, the consumer of the results might not care: for example given the expression `exists(x//y//z)`, sorting the node sequence will not affect the outcome.

The main aspect of the analysis is determining combinations of axis steps that are "naturally sorted". This is the case for any sequence of child axis steps. It is also true for an expression such as `a/b//c`, but not (perhaps surprisingly) for `a//b/c`. There is no space here to give the rules in detail.

One case that often causes difficulty is a path such as `$x/a/b/c` that starts with a variable reference. Here, if `$x` is a singleton node sequence, or any sequence that is sorted, contains no duplicates, and contains no node that is an ancestor of any other, then the entire path will be "naturally sorted", making sorting unnecessary. This can sometimes be determined by static analysis, but failing this, Saxon generates conditional code to test at run-time whether `$x` is a singleton, and thus avoids the sort in this common case.

For the query `/site//keyword`, which returns around 70,000 nodes on the XMark 100Mb database, eliminating the sort reduces the TinyTree execution time from 107ms to 60ms. When running against a DOM, where sorting into document order is more expensive, the saving is more dramatic: against the 10Mb database, run time reduces from 1300ms to 290ms; for 100Mb, the query does not even complete without this optimization.

3.4 Join Optimization

Saxon-SA optimizes joins by constructing hash indexes and then using them to support fast filtering of indexed sequences. The optimizer does not actually recognize the concept of a join. What it does is firstly, to break up the condition in the `where` clause of a FLWOR expression and distribute it among the input sequences read by the expression, thereby turning them into XPath filter expressions; and then (independently) it identifies filter expressions that are likely to benefit from indexing.

Two kinds of index are used: indexed documents, and indexed variables. Wherever possible, an index is attached to a document node, which allows it to be reused whenever that document is searched, even in a different query. Where this is not possible, the contents of a variable can be indexed: such an index dies when the variable goes out of scope.

Join optimization is widely discussed in the database literature. A significant difference for Saxon is that there are no pre-existing indexes: any index that is required must be created within the query. Nevertheless, impressive savings are possible in the right circumstances. For example, Table 2 shows the performance of XMark query $q9$ against databases of different sizes using Saxon-B (without join optimization) and Saxon-SA (with).

Table 2: Join optimization

	1Mb	10Mb	100Mb
Saxon-B	41ms	3612ms	381543ms
Saxon-SA	3ms	26ms	246ms

It is plain here that Saxon-SA performance is linear while Saxon-B is quadratic.

3.5 Miscellaneous Rewrites

Further compile-time expression rewrites done by the Saxon-SA optimizer include the following:

- Replace `count(X)=0` by `empty(X)`. This takes advantage of the fact that when `X` is pipelined, the latter expression can exit as soon as it sees the first item in the sequence; there is no need to compute the count.

- Constant folding: constant subexpressions are evaluated at compile time.
- Variable inlining: when a variable is only referenced once, and not in a loop, the reference is replaced by the initializing expression
- Function inlining: calls to non-recursive functions of modest size are replaced by the function body. This often enables further optimization of the new expression.
- Loop lifting: expressions within a repeatedly-evaluated subexpression (for example a filter predicate, or the return clause of a FLWOR expression) that do not depend on the loop variables are moved outside the loop, but taking care to ensure that they are not executed if the loop is iterated zero times.
- Global variable extraction: expressions within a function body that do not depend on the function arguments are promoted to global variables.
- Compound if/then/else expressions acting as switch statements, testing the value of one expression against a range of constant values, are recognized and supported by hashing.

The benefits achieved by these rewrites are highly variable. In each case it is easy to find example queries where the rewrite gives an order-of-magnitude improvement. It is less easy to quantify how many queries benefit from each rewrite. Very often these rewrites are most effective in combination: one apparently minor rewrite simplifies the expression sufficiently to enable another more powerful one, in particular, the join optimizations discussed in the previous section.

3.6 Schema-Aware Processing

Schema-aware processing allows a query to be compiled with knowledge of the schema that a source document will conform to.

The major benefits of schema-aware processing are usability and reliability: it enables easier debugging of queries, and increases the likelihood that a query that is put into production with inadequate testing (as many are) will turn out to be bug-free.

The effect of schema-aware processing on performance is in fact mixed. For some applications, the overhead of performing schema validation on the input outweighs any savings achieved through greater intelligence in the query execution plan. There are also cases where manipulating the document as raw text turns out to be faster than processing it as typed content.

An example where schema-aware processing has a negative effect on performance is in XMark query $q11$, which is dominated by the predicate

```
where $p/profile/@income > (5000 * $i)
```

If the attribute `@income` is typed as `xs:decimal`, and if `$i` is also `xs:decimal`, which will happen if the schema for the XMark database is written to use `xs:decimal` for money amounts, then this will involve a decimal comparison; whereas without schema-awareness, the comparison will use double-precision floating point. In Java, on a typical platform, double arithmetic is much faster than decimal, because it is supported in the hardware. A user who is aware of this problem can work around it, but by default, the query will run more slowly.

On the other hand, knowledge of the paths that exist in the source data can sometimes be exploited to great advantage. The XMark benchmark queries tend to be written with full paths, such as

```
let $ei := $site/people/person/creditcard
```

but real users are often less patient, and write

```
let $ei := $site//creditcard
```

Given sufficient type information, Saxon-SA will rewrite the abbreviated path to use the step-by-step form, which can greatly reduce the number of nodes that need to be searched.

With schema-aware processing, the second query takes around 6ms on the 100Mb XMark dataset; without schema-awareness, it takes 51ms. However, schema validation increases the parsing time for the source document from 5s to 15s.

3.7 Streaming

Saxon-SA provides the ability to execute certain queries in streaming mode. This is not done as an automatic optimization, but must be explicitly requested using a pragma. In this mode, simple expressions can be evaluated without first building a tree in memory.

This does not make the query itself run faster, but it saves the cost of building the tree, and of course it enables source documents to be processed that are too large to fit in memory (transforming a 20Gb document has been timed at 50min [24]).

Streaming is a natural extension of pipelining: it pipelines together the operations of parsing and query evaluation, removing the need to materialize the intermediate data, that is, the tree representation of the source document.

For a query such as `count(//person)` on 100Mb of input, the execution time including parsing is around 5s with streaming, 5.6s without. The big difference is that with streaming, memory is reduced from 450Mb to 1.7Mb. So the effect is not so much on the speed of the query, as on its scalability. This illustrates the message that performance cannot be considered a one-dimensional property.

Speed improves greatly when the file is not read to completion. The query `exists(//africa)` on the same data takes just 180ms with streaming, 5.6s without.

3.8 Document projection

Document projection (see [9]) is a technique for building a tree containing only that subset of the source document that is needed to execute a query, as determined by static analysis of the query. As with streaming, the technique is only suitable where the document is being parsed in order to execute one query that is known in advance, but unlike streaming, it works with any query.

At present Saxon never does document projection automatically, only on request. The main reason for this is that the risk of bugs is considered high, since it relies on inferencing about the access paths used by every single construct in the language.

Document projection, like streaming, has more effect on memory usage than on execution time: with XMark, it reduces the tree size by 90% or more for 15 out of 20 queries, but only two are speeded up by more than 25% ($q6$ by 75%, and $q7$ by 95%).

3.9 Java code generation

Saxon-SA offers the option to generate Java bytecode representing the logic of the query, as an alternative to interpreting the query execution plan. (This is currently done indirectly, via generation of Java source.) The generated code may be executed from the command line, via an API, or as a Java servlet. Many operations, of course, are still handled by calls to the run-time library, the same library that the interpreter uses.

The speed-up obtained by compilation is not as great as one might expect: 25% is typical. For XMark (10Mb), the biggest improvement (54%) is to the slowest query, $q11$, from 3344ms to 1541ms. The saving appears to be greatest for queries dominated by arithmetic or string manipulation – simple path expressions

show very little improvement over the interpreter. This suggests that an equally effective (and more convenient) strategy might be to do just-in-time compilation of a few selected subexpressions.

I experimented at one time [25] with generating code for path expressions that was committed to a particular tree model such as the TinyTree, rather than working generically on any tree model. The results were not encouraging, so the experiment was abandoned. Part of the reason is that evaluating path expressions is already very fast.

3.10 Methodology

I said I would give ten reasons why Saxon is fast, and the first nine have been technical characteristics of the delivered product. The final reason is deeper, and relates to the engineering discipline used to develop the software. Here are a few lessons learnt from the experience of developing Saxon over a period of ten years:

- Investigate every user-supplied performance problem in depth. There is no better raw material for understanding how the code behaves, and without such understanding there can be no improvement.
- Optimize the code that typical users write, whether it is well-written code or not. Try to educate users on how to write code that works well on your product, but recognize that you will only reach a small minority.
- Never make performance improvements to the code without measuring the impact. If you cannot measure a positive impact, revert the change (easily said, but psychologically very difficult when you've put a lot of effort in). Keep records of what you learnt in the process.
- Avoid performance improvements that rely on user-controlled switches. Most users (including people who publish comparative benchmarks) will never discover the switch exists; of the remainder, a good number will set the switch sub-optimally.
- Remember that every optimization you make to your code is likely to require a substantial investment in new test material, and even then, is likely to result in several new bugs escaping into the field. Do not do it unless the gain is worth it.
- Maintain a set of performance regression tests to ensure that performance gains made in one release are not lost in the next.
- Separately, maintain tests to show that query optimizations are taking place as intended. In Saxon this is done by outputting an XML representation of the query execution plan for test queries, and checking assertions about these plans expressed as auxiliary queries.

For the other nine ways of achieving good performance in Saxon, I have tried to quantify the benefit. For this tenth cause, I am afraid I cannot do so – I do not have anything to compare with.

4 Conclusions

In this paper I have presented ten characteristics of the Saxon XQuery implementation that contribute to its performance, and for most of these, I have attempted to quantify the size of that contribution for some selected queries.

Few of these mechanisms are unique to Saxon; what makes Saxon distinctive is the deployment of a balanced portfolio of techniques to deliver efficient query execution over a variety of user workloads, coupled with a determination to place other qualities of the product (standards conformance, reliability, usability) ahead of raw performance. In a crowded marketplace with over 50 XQuery implementations competing for user attention, I believe it is this balanced approach that has led many users to make Saxon their preferred choice.

References

[1] http://saxon.sf.net/

[2] http://www.saxonica.com/

[3] http://www.w3.org/TR/xquery/

[4] http://www.w3.org/TR/xquery-update-10/

[5] http://www.w3.org/TR/xslt20/

[6] http://www.w3.org/TR/xpath20/

[7] http://www.w3.org/TR/xmlschema-1/

[8] http://www.w3.org/TR/xmlschema11-1/

[9] Amélie Marian and Jérôme Siméon, *Projecting xml documents*, in Proc. of 29th International Conference on Very Large Data Bases, 2003, pp. 213–224.

[10] http://www.ikvm.net/

[11] http://saxonica.blogharbor.com/blog/_archives/2006/8/13/2226871.html

[12] http://ilps.science.uva.nl/Resources/MemBeR/other-benchmarks.html

[13] http://gemo.futurs.inria.fr/events/EXPDB2006/PAPERS/Afanasiev.pdf

[14] http://portal.acm.org/citation.cfm?id=1324679

[15] http://www.ibm.com/developerworks/library/x-xslt2/

[16] http://www.w3.org/TR/xpath-datamodel/

[17] http://www.w3.org/DOM/

[18] http://www.jdom.org/

[19] http://www.dom4j.org/

[20] http://xom.nu/

[21] http://xml.apache.org/xalan-j/dtm.html

[22] K. Jones, J. Li, and L. Yi, *Building a C++ processor for large documents and high performance*, in Extreme Markup Languages, 2007.

[23] http://www.xml-benchmark.org/

[24] http://saxonica.blogharbor.com/blog/_archives/2007/9/25/3252121.html

[25] http://saxonica.blogharbor.com/blog/_archives/2006/7/24/2157486.html