

Towards micro-benchmarking XQuery

Ioana Manolescu
INRIA Futurs, France

Cédric Miachon
LRI - Université Paris-Sud 11,
France

Philippe Michiels
University of Antwerp, Belgium

ABSTRACT

A substantial part of the database research field focusses on optimizing XQuery evaluation. However, optimization techniques are rarely validated by means of cross platform benchmarking. The reason for this is that there is a lack of tools that allows one to easily compare different implementations of isolated language features. This implies that there is no overview of which engines perform best at certain XQuery aspects, which in turn makes it hard to pick a reference platform for an objective comparison. This paper is a first step in a larger effort to bring an overview of the available implementations along with their strengths and weaknesses. It is meant to guide implementors in benchmarking and improving their products.

1. INTRODUCTION

In the recent past, a lot of energy has been spent on optimizing XML querying. This resulted in many implementations of the corresponding specifications, notably XQuery and XPath. Usually, little time and space is spent on thorough measurements across different implementations. This complicates the task of implementors to compare their implementations to the *state of the art* technology, since no one really knows what system actually represents it.

As is pointed out in [2], there are two possible approaches for comparing systems using benchmarks. Application benchmarks like XMark [12], XMach-1 [6], X007 [7] and XBench [15] are used to evaluate the overall performance of a database system by testing as many query language features as possible, using only a limited set of queries. As such, this kind of benchmarks are not very useful for XPath/XQuery implementors, since they are mainly interested in isolated aspects of an implementation that need improvement.

Micro-benchmarks, on the other hand, are designed to verify the performance of isolated features of a system. We believe that microbenchmarks are crucial in order to get a good understanding of an implementation. Moreover, it rarely

happens that one platform is the fastest on all aspects. Only microbenchmarks can reveal which implementation performs best for isolated features.

Our focus is to benchmark a set of important XQuery constructs that form the foundation of the language and thus greatly impact the overall query engine performance. These features are:

- XPath navigation
- XPath predicates (including positional predicates)
- XQuery FLWORS
- XQuery Node Construction

The selected XQuery processors are chosen to represent both in-memory and disk-based implementations of the language as well as freelance engineering products.

We regard this paper as an initial step to a much larger micro-benchmarking effort that started with MemBeR [2] and we hope to continue, using automated tools such as XCheck [1]. This continuation involves the population of a repository with a large amount of ready-made micro-benchmarks as well as the benchmarking of many more platforms. We hope that this work can guide XQuery implementors to improve their products based on objective, thorough and relevant measurements.

2. SETTINGS

In this section, we present the documents (Section 2.1) and queries (Section 2.2 and 2.3) used for the performance measures in this paper, as well as the rationale for choosing them. Section 2.4 describes our hardware and software environment, and the system versions used.

All documents, queries, settings, and (links to) the systems used in these measures can be found at [14].

2.1 Documents

In order to have full control over the parameters characterizing our documents, we used synthetic ones, generated by the MemBeR project's XML document generator [2, 3]. MemBeR-generated documents consist of simple XML elements, whose element names and tree structure is controlled by the generator's user. Each element has a single attribute

called **@id**, whose value is the element’s positional order in the document. The elements have no text children.

The tests we performed carry over several systems, some of which are based on a persistent store, while the others run completely in memory. While we are clearly aware of the inherent limits that an in-memory system encounters, we believe it is interesting to include both classes of systems in our comparison, since performant techniques have been developed independently on both sides, and we believe the research community can learn interesting lessons from both. To enable uniform testing of all systems, we settled for moderate-sized documents of about **11 MB**, which most systems can handle well.

Three documents have been used in these measures, and their structure is outlined in Figure 1. In this figure, white nodes represent elements, whose names range from **t1** to **t19**; black nodes represent **@id** attributes. All documents have the depth **19**, which we chose so that complex navigation can be studied, and in accordance with the average-to-high document depth recorded in a previous experimental study [11].

- The **exponential2.xml** document’s size is **11.39 MB**. At level i (where the root is at level 1), the document has 2^{i-1} elements labeled t_i .
- The **layered.xml** document’s size is **12.33 MB**. The root is labeled **t1**, and it has **32768** children labeled **t2**. At any level i comprised between 3 and 19, there are **32768** nodes labeled t_i . Each element labeled t_i , with $3 \leq i \leq 18$, has exactly one child labeled $t(i+1)$. Elements labeled **t19** are leaves.
- The **mixed.xml** document’s size is **12.33 MB**. The root is labeled **t1**, and it has **3268** children labeled **t1**. Each such child (at level 2) has **10** children labeled (with equal probability) **t3**, **t4**, ..., **t12**. Nodes at levels comprised between 3 and 18 each have **1** child, labeled (with equal probability) **t3**, **t4**, ..., **t12**. At level 19, all nodes are leaves, and are labeled **t13**.

The rationale for choosing these documents is the following. The document **exponential2.xml** allows studying the impact of increasing number of nodes at a given level, on the performance of path traversal queries. At the same time, in this document, the size of a subtree rooted at level i is exponential in i . At another extreme, the document **layered.xml** has the same depth and approximate tree size as **exponential2.xml**, but the size of subtrees rooted at various levels depends only linearly on the level. The subtree shapes exhibited by both **exponential2.xml** and **layered.xml** are quite extreme; subtrees from real-life documents are likely to be somewhere in between. Controlling both the path depth and the size of the subtrees rooted at each depth is important, since these parameters have important, independent impacts on query performance: the first determines the performance of navigation queries, while the second determines the performance of reconstructing (or retrieving) full document subtrees.

Our last document was chosen so as: (i) to be of overall size and aspects close to the two previous documents; (ii) to feature different tags uniformly distributed over many levels, thus allowing us to vary, in a controlled manner, the *structural selectivity* of various queries (by allowing some tags to range over increasingly large subsets of **{t1,t2,...,t13}**).

For simplicity, and unless otherwise specified, the documents were not characterized by type information.

2.2 XPath Queries

We measured seven parameterized XPath queries, denoted **Q1.1(n)**, **Q1.2(n)**, ..., **Q1.7(n)**, where $1 \leq n \leq 19$, as follows:

- **Q1.1(n)** is: `/t1/t2/.../tn`
- **Q1.2(n)** is: `/t1/t2/.../tn/@id`
- **Q1.3(n)** is: `(/t1/t2/.../tn)[1]/@id`
- **Q1.4(n)** is: `(/t1/t2/.../tn)[position()=last()]/@id`
- **Q1.5(n)** is: `/t1[t2/.../tn]/@id`
- **Q1.6(n)** is: `/t1[t2/.../tn]/t2/.../tn/@id`
- **Q1.7(n)** is: `//tn`

We measured these queries on **exponential2.xml**. We also measured **Q1.1(n)** on **layered.xml**, which provided some interesting insights when compared to the results on the first document.

The query **Q1.1(n)** retrieves nodes at increasing depth in the document. Its output size decreases as the roots of the returned subtrees move lower in the document. To distinguish the impact of navigation from the impact of subtree serialization in the output, we also use the query **Q1.2(n)**, which navigates at the same depth as **Q1.1(n)** but only returns simple attribute values. Query **Q1.3(n)** is used to see if query engines are able to take advantage of the `[1]` predicate to shortcut navigation as soon as a single node is found. Query **Q1.4(n)** is similar, but it uses the `[position()=last()]` predicate, which does not easily allow the same optimization as `[1]` if the engine is coded to navigate over the target nodes in the order dictated by XPath’s semantics [8]. Measuring both **Q1.3(n)** and **Q1.4(n)** allows us to infer which kinds of navigation optimization techniques are supported in the engine. The queries **Q1.5(n)** aim at quantifying the impact of increasingly deeper navigation along existential branches. Query **Q1.6(n)** presents an optimization opportunity (the existential branch can be suppressed without changing the query semantics); its results are to be interpreted together with those of **Q1.2(n)**. Finally, query **Q1.7(n)** retrieves all elements of a given tag. Its results are to be compared with those of **Q1.1(n)**, to see if the user’s knowledge of the depth of the desired elements simplifies the query processor’s task.

2.3 XQuery Queries

We measured a set of six parameterized XQuery queries, denoted **Q2.1(n)**, **Q2.2(n)**, ..., **Q2.6(n)**, where $0 \leq n \leq 9$, as follows:

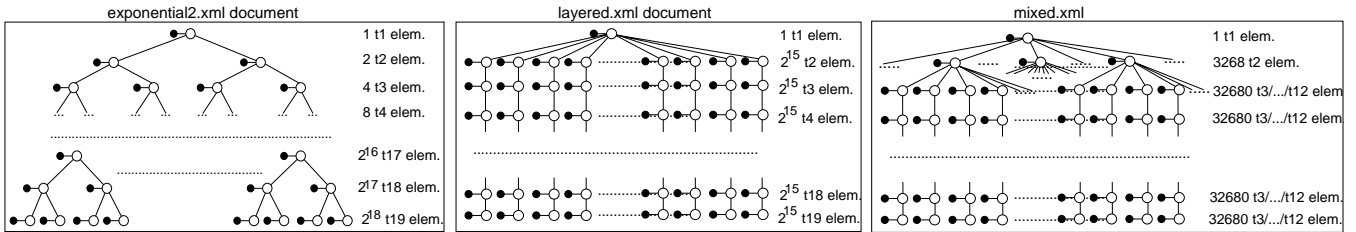


Figure 1: Outline of the documents `exponential2.xml`, `layered.xml` and `mixed.xml` used in the measures.

- **Q2.1(n)** is:
 for $\$x$ in `/t1/t2`
 return `<res>{ for $\$y$ in $\$x$ /*/*/*.../*`
 `return fn:data($\$y$ @id)}`
 `</res>`

where the path expression to which $\$y$ is bound features n child navigation steps starting from $\$x$.

- **Q2.2(n)** is:
 for $\$x$ in `/t1/t2`
 return `<res>{ $\$x$ /*/*/*.../*}</res>`

where the path expression in the `return` clause features n child navigation steps starting from $\$x$.

- **Q2.3(n)** is:
 for $\$x$ in `/t1/t2`
 return `<res>{ $\$x$ //t3, $\$x$ //t4, ...,`
 `$\$x$ //t($n+2$)}</res>`

- **Q2.4(n)** is:
 for $\$x$ in `/t1/t2`
 return `<res>{ $\$x$ /*[position() $\leq n$]}</res>`

- **Q2.5(n)** is:
 for $\$x$ in `/t1/t2`
 return `$\$x$ /*[position() $\leq n$]`

- **Q2.6(n)** have increasingly deeply nested `return` clauses. All the queries retrieve `t13` elements, and return them “wrapped” in increasingly deeper `<res>` elements. Rather than giving the general form, quite difficult to read, we provide here some examples:

Q2.6(0):
 for $\$x$ in `/t1/t2` return
`<res>{for $\$x1$ in $\$x$ /* return`
 `<res>{ $\$x1$ //t13}</res>}`
`</res>`

Q2.6(1):
 for $\$x$ in `/t1/t2` return
`<res>{for $\$x1$ in $\$x$ /* return`
 `<res>{for $\$x2$ in $\$x1$ /* return`
 `<res>{ $\$x2$ //t13}</res>}`
 `</res>}`
`</res>`

Q2.6(2):
 for $\$x$ in `/t1/t2` return
`<res>{for $\$x1$ in $\$x$ /* return`
 `<res>{for $\$x2$ in $\$x1$ /* return`
 `<res>{for $\$x3$ in $\$x1$ /* return`
 `<res>{ $\$x3$ //t13}</res>}`
 `</res>}`
 `</res>}`
`</res>`

Query **Q2.6(n)** returns subtrees consisting of $n+2$ recursively nested `<res>` elements, each of which encloses some `t13` elements.

We measured these queries on `mixed.xml`.

The queries **Q2.1(n)** test the performance of increasingly deeper navigation *in the return clause*, while returning results of modest (and, on `mixed.xml`, constant) size.

The queries **Q2.2(n)** combine increasingly deep navigation with decreasingly large subtrees to be copied in the output (recall that XQuery semantics [5] requires the content of newly created elements to be copied).

The queries **Q2.3(n)** are designed to return results whose size is expected to increase with n , given that elements labeled `t3`, `t4`, ..., `t12` are uniformly distributed over levels 3-18 in `mixed.xml`. Moreover, the elements which **Q2.3(n)** must retrieve are scattered over the document. **Q2.4(n)** returns results whose size linearly grows with n , however in this case, the elements to be returned are grouped in contiguous subtrees in the original document. The performance of **Q2.3(n)** compared with that of **Q2.4(n)** provides interesting insight on the node clustering strategy used by the system (if any).

The queries **Q2.5(n)** are similar to **Q2.4(n)**, however **Q2.5(n)** does not construct new elements (strictly speaking, **Q2.5(n)** could have been expressed in XPath, but we keep it in the XQueries group for comparison with **Q2.4(n)**). Given that **Q2.5(n)** does not construct new elements, there is an opportunity for a more efficient evaluation than in the case of **Q2.4(n)**, since no tree copy operation is needed.

Finally, the queries **Q2.6(n)**, on the document `mixed.xml`, will all return 3260 `<res>` elements, since there are 3260 `t2` elements in `mixed.xml`. Moreover, each such `<res>` elements will include copies of 10 `t13` elements. As n grows, however, the number of “layers” of `<res>` elements in which the `t13` elements are wrapped increases. The queries and the

document have been chosen to capture the impact of result nesting only.

2.4 Hardware and software environment

Our measures were performed on four different computers. While ideally a single machine should have been used, the four computers have overall similar parameters. Moreover, we do not aim at a direct comparison of precise running times across different systems, as (i) the internal differences between some of them would render such comparisons irrelevant and (ii) we are more interested in identifying the tendency of each system's running time across increasingly complex queries; such tendencies are likely to be quite stable, even if different computers are used. The machines are described by the following hardware and software parameters:

M1 is equipped with a 2.00 GHz Pentium 4, 512 MB of RAM, running Linux 2.6.12-10-386.

M2 is a DELL Precision M70 laptop, equipped with a 2.00 GHz Pentium, 1GB of RAM, running Linux 2.6.13.

M3 is equipped with a 3.00 GHz Pentium 4, 512 MB of RAM, running Linux Debian 2.6.16.

M4 is equipped with a 3.00 GHz Pentium 4 CPU, with 2026 MB of RAM, running Linux version 2.6.12.

We tested the following systems:

CDuce/CQL version 0.4.0.

eXist version 1.0 beta 1.

Galax version 0.5.0.

MonetDB/XQuery version 0.10, 32 bit compilation with no optimizations.

QizX/open version 1.0.

Saxon version 8.6.1.

We chose systems that (i) were freely available (if possible open source), (ii) had a user community and/or (iii) were the target of recent published research works. Our choice of systems includes some endowed with a persistent store (eXist, Galax and MonetDB), as well as purely in-memory systems (CDuce, QizX and Saxon). In this work, we did not specifically target our measures at disk-based retrieve times of disk-resident systems (although of course this aspect is also interesting), rather, we aimed at studying the performance of various algorithms implemented in the engines once they run in memory.

To that effect, we ran *each measure 4 times, and report the average of the last 3 (hot) runs.*

We are aware that more systems meeting our criteria exist, and plan to extend our tests to such systems in the near future.

3. RESULTS BY SYSTEM

This section discusses the results for each of the benchmarks per individual system.

3.1 MonetDB

Figure 2 plots MonetDB's execution times on the XPath queries we considered. These measures were performed on **M4**.

Note that the times for **Q1.1(n)** and **Q1.7(n)** coincide, showing that descendent navigation does not pose particular difficulties for MonetDB. The times for **Q1.3(n)** and **Q1.5(n)** are very similar, and very low for all n values. The times for **Q1.2(n)** and **Q1.6(n)** are very low until $n = 16$, then an exponential growth projects the curves upwards (the actual value for $n = 19$ is shown on top of the graph). Similarly, **Q1.4(n)** exhibits an abrupt growth at $n = 11$.

The overall descending tendency of **Q1.1(n)** demonstrates that deep navigation does not pose difficulties for MonetDB; the dominating cost component here is clearly serialization, which decreases as n grows. Comparing **Q1.1(n)** and **Q1.2(n)** (on its initial segment) highlights the impact of the size of each returned subtree on the evaluation time; **Q1.2(n)** only returns simple data nodes, while **Q1.1(n)** returns more complex subtrees. The almost identical curves for **Q1.2(n)** and **Q1.6(n)** shows that the extra existential branch in **Q1.6(n)** is either eliminated at query compile time, or very efficiently executed.

The very low cost of **Q1.3(n)**, compared with the cost of **Q1.4(n)** which explodes above $n = 11$ and the cost of **Q1.2(n)** as a baseline, leads to the following observation. The **[1]** predicate after a long path expression in **Q1.3(n)** is exploited to make the evaluation more efficient than plainly enumerating all nodes of the path expression, and then returning just the first one. Thus, **Q1.3(n)** is stable and cheap, while **Q1.2(n)** (which *has* to enumerate exponentially many **@id** nodes) explodes at some point. On the contrary, the **[position()=last()]** predicate in **Q1.4(n)**, far from giving optimization opportunities, only makes matters worse than if no predicate is present, since **Q1.4(n)** costs rise before **Q1.2(n)** costs do.

The behavior exhibited by **Q1.2(n)**, **Q1.4(n)** and **Q1.6(n)** can be attributed to the handling of exponentially many nodes in intermediary XPath results. Curiously enough **Q1.2(n)** exhibits a significant performance difference with **Q1.1(n)**, indicating that attribute axes are not handled in the same way as child axes.

Figure 3 plots MonetDB's detailed times for **Q1.1(n)**, on **exponential2.xml** and **layered.xml**. The partial times reflect, respectively, *query compile*, *query execution*, and *result serialization*. Figure 3 clearly shows that serialization is the single most important cost component for **Q1.1(n)**; both query compilation and query execution (understood here as the time to locate the return nodes only) are very small (more precisely, smaller than 0.2s across all documents and n).

MonetDB execution time on XQueries is depicted in Figure 4. **Q2.3(n)** and **Q2.4(n)** are of similar costs, and by far

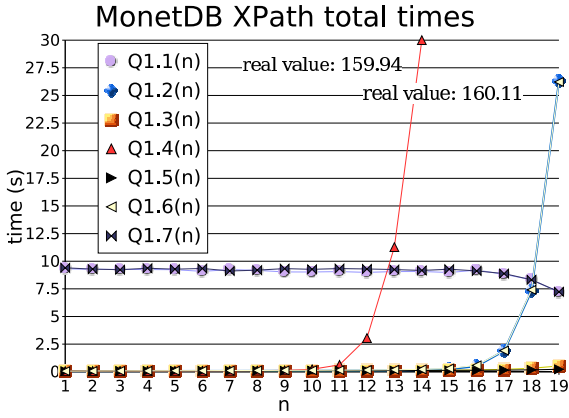


Figure 2: MonetDB XPath total times.

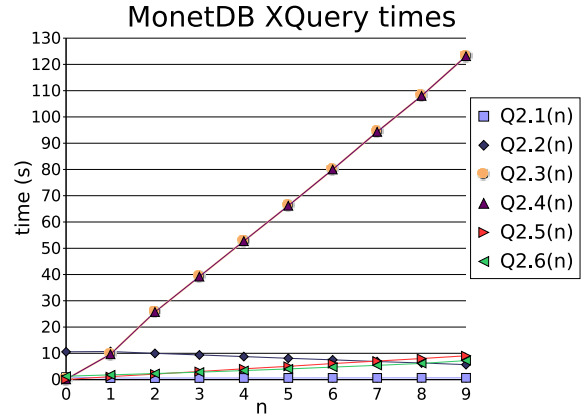


Figure 4: MonetDB XQuery total times.

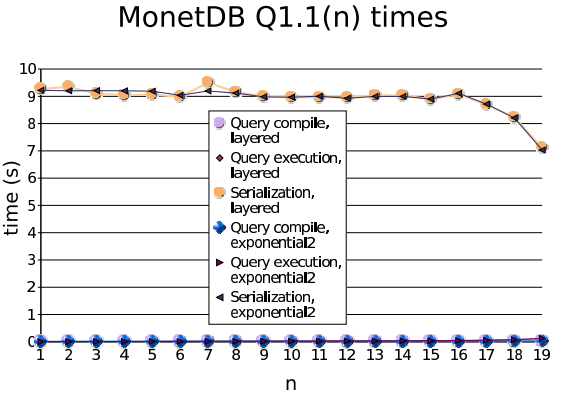


Figure 3: MonetDB XPath component times.

the most expensive. Both these queries return large results, constructed either from contiguous subtrees ($Q2.4(n)$) or from elements scattered all over the document ($Q2.3(n)$). Clearly, the location of the elements to be copied in the output does not have a big impact on evaluation performance. In contrast, $Q2.5(n)$ which returns similar data volumes takes much less time. This shows that MonetDB correctly realizes no tree copy is needed for this query, but only a serialization of the *existing* returned nodes. The time of $Q2.5(n)$ is of the same order of magnitude as the times for $Q1.1(n)$ (Figure 2). Thus, we conclude that tree copying is significantly more expensive than serializing existing nodes in MonetDB (while keeping a linear scaleup), and that MonetDB correctly recognizes the situations where each of these should be used.

3.2 QizX

Figure 5 shows QizX execution times for the XPath queries we considered, on the `exponential2.xml` document. All measures of this section were performed on `M2`.

For low n values, the execution times of $Q1.1(n)$ and $Q1.7(n)$ decrease with n , due to the decrease in the volume of returned data. $Q1.1(n)$ and $Q1.7(n)$ exhibit an exponential growth with large n values, justified by the exponential increase in the number of returned nodes. Interestingly, the times for the `//t`i queries are quite different from the times

of the equivalent query based on child steps, suggesting that QizX’s search for elements to return at any depth in the document actually does search at all levels (no structural or tag-based index seems to be in place).

QizX allows to isolate in a measure:

the evaluation time: presumably the time spent processing the navigation (XPath) part of a query (in other words, the locate time);

the display time: the time needed to produce a serialization of the returned nodes (in other words, the serialization time).

We refine our understanding of QizX’s behavior on $Q1.1(n)$ by measuring the evaluation and the display times both on `exponential2.xml` and on `layered.xml`. Figure 6 shows the results. We notice that QizX’s reported evaluation time is almost unchanged for all documents and path depths, which is quite surprising, and lead to believe that it corresponds to some document pre-processing time (e.g. to build a tree representation of the document and/or index all nodes by their paths etc.). The display (probably serialization) time decreases almost linearly with the growth of n , which is the expected behavior for the `layered.xml` document. However, the display time on `exponential2.xml` shows an exponential increase for large n values. On this document, the size of the query result decreases as n grows, and so does the number of nodes *included in the result subtrees*, however the *number of result subtrees* increases exponentially with n . Thus, the serialization complexity of QizX is strongly influenced *number of result subtrees*, and also (to a lesser extent) by the *number of nodes in the result subtrees*.

Figure 7 shows QizX’s total XQuery execution times on `mixed.xml`. For all queries, QizX’s reported evaluation time was between 12ms and 30ms, thus the times in Figure 7 practically coincide with its display time. Such insignificant (and constant) evaluation times lead to believe that for an XQuery, QizX reports as evaluation time the time needed to process *the for clause of the outermost FLWOR expression* only, while the rest of the query execution is reported as display time.

QizX XPath total times

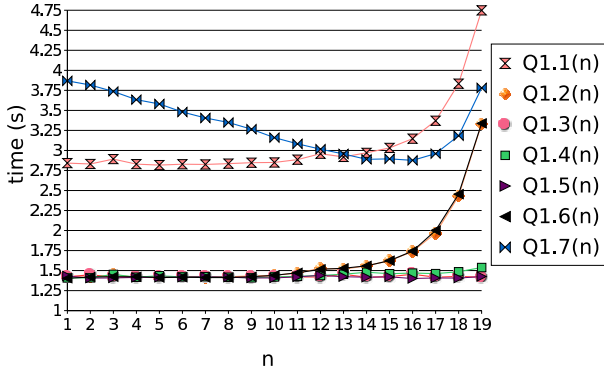


Figure 5: QizX XPath total times.

QizX Q1.1(n) times

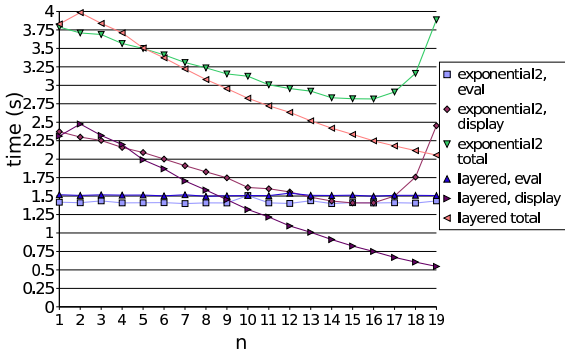


Figure 6: QizX XPath component times.

Figure 7 shows that $Q2.3(n)$, which returns massive results scattered over the whole document, is the single most expensive query. Comparing this with the times for $Q2.4(n)$, which returns similar data volumes but contiguous subtrees, allows to infer that `//` navigation, in the `return` clause as well as in `for` clauses (or XPath) brings quite a penalty to QizX.

Figure 8 plots the times for all XQueries except $Q2.3(n)$. $Q1.1(n)$ takes almost constant time; given that the cardinality and size of its result is constant as n grows, we can conclude that deep step-by-step navigation has no big impact on QizX performance, and confirm that the size and cardinality of the result strongly impacts serialization. The decreasing times of $Q2.2(n)$, producing shallower trees as n grows, support the same assertion. $Q2.4(n)$ and $Q2.5(n)$ are quite close, with a small (but persistent) overhead of $Q2.4$ over $Q2.5$, probably due to the construction of a new `<res>` node as required by $Q2.4$. However, the difference seems too small to reflect fully copying the subtrees included in `<res>`.

The overhead of nested `<res>` elements in $Q2.6(n)$ is noticeable, while the scaleup remains linear.

QizX XQuery times

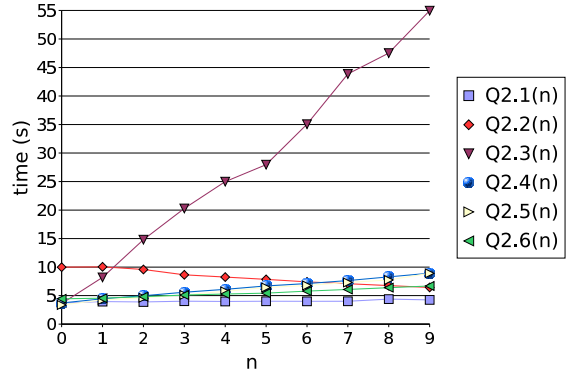


Figure 7: QizX XQuery total times.

QizX XQuery times

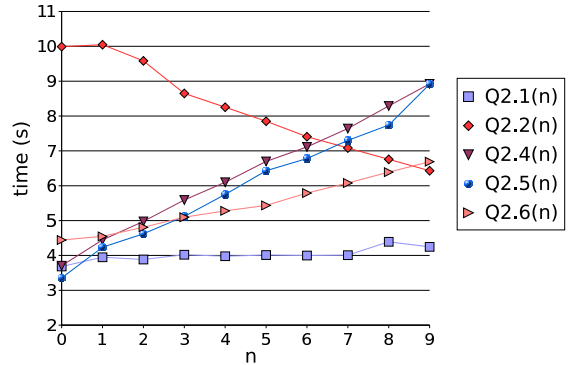


Figure 8: QizX XQuery total times.

3.3 CDuce

Figure 9 depicts the total running time of CDuce for our XPath queries on `exponential2.xml`. These measures were performed on `M3`.

CDuce (pronounce "seduce") is a general purpose typed functional programming language, specifically targeted to XML applications. It conforms to basic standards such as DTDs, Namespaces etc. Theoretical foundations of the CDuce's type system can be found in [9]. The system is implemented in OCaml.

Recently, a "Select-From-Where" syntax similar to XPath has been added for user convenience on top of CDuce [4]; it is translated into CDuce. The CDuce system, a sample demo, and online documentation can be found at [13]. An important characteristic of CDuce is its *pattern algebra*, which extends the XDUCE [10] pattern algebra, allowing complex and powerful pattern matching. Pattern matching in CDuce is implemented by means of automata constructed "just-in-time".

CDuce has a strong type system, which enables static verification of safe program composition. Of course, CDuce can also evaluate queries in the absence of type information, using a more general automaton. A large subset of XQuery (excluding, for instance, function calls) is covered by CQL,

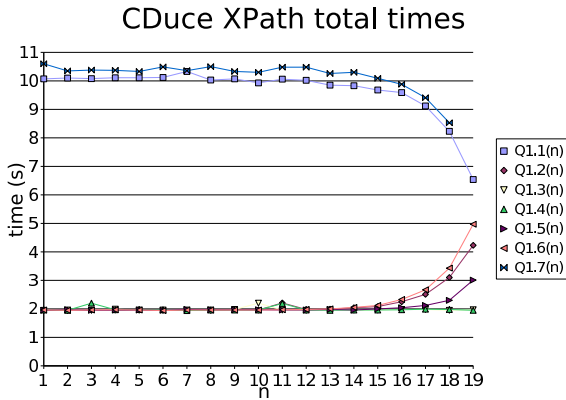


Figure 9: CDuce XPath total times.

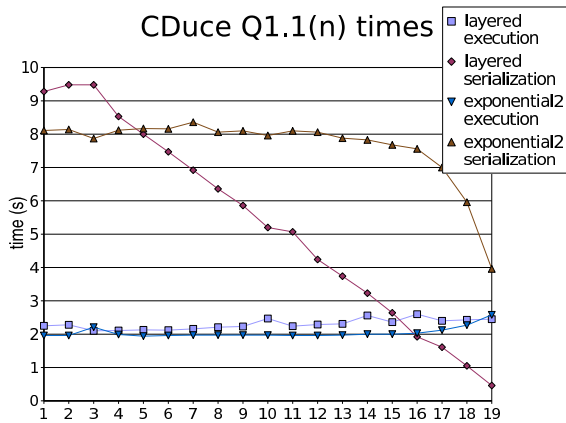


Figure 10: CDuce XPath component times for $Q1.1(n)$.

and all the queries we measured in this paper are part of this subset. (Other XQuery features not considered here can be expressed directly in the CDuce language.) Among the XQuery features not supported is *node identity*: CDuce is value-based, that is, it does not distinguish two distinct nodes having the same serialized value (other than by their respective positions in the original document). In particular, there is no way to test if two variable bindings correspond to *the same* node, in XQuery sense [5].

In Figure 9, the running times of $Q1.1(n)$ and $Q1.7(n)$ are comparatively more important, and decrease as n grows, due to the decreasing total size of the query result. Interestingly, $Q1.7(n)$ causes a stack overflow for $n = 19$. This is because descendant navigation is implemented in CDuce a stack that keeps all descendant matched on recursive function calls; in contrast, navigation along the child, parent and sibling axes is implemented by automata alone. However, the curves for $Q1.1(n)$ and $Q1.7(n)$ are roughly similar.

Both $Q1.3(n)$ and $Q1.4(n)$ have very short running times, which are almost constant. The translation of these XPath queries to CQL, combined with the pattern matching concept underlying the system, allows to push the [1] and [position()=last()] predicates at all levels up, thus the efficient evaluation.

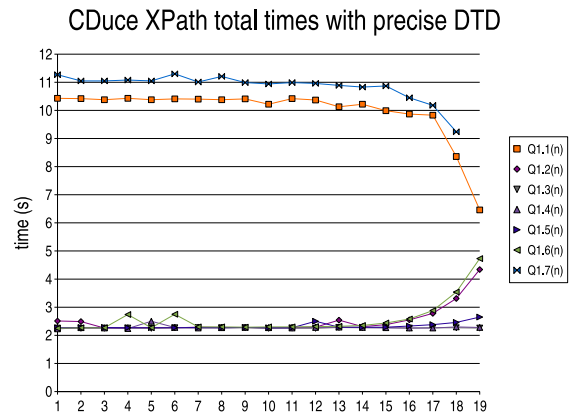


Figure 11: CDuce XPath total times in the presence of precise DTD information

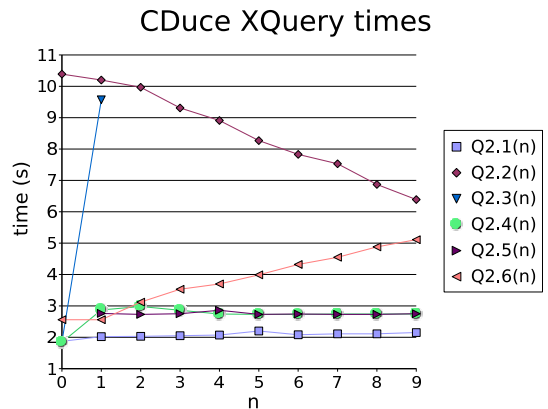


Figure 12: CDuce XQuery total times.

The time for $Q1.2(n)$ exhibits an exponential growth as the number of returned nodes grows exponentially. The same observation holds for $Q1.6(n)$, which is more expensive than $Q1.2(n)$, although they are equivalent. The reason is that the pattern resulting from the translation of $Q1.6(n)$ has twice the size of the pattern for $Q1.2(n)$ (in other words, the pattern is not minimized to eliminate the redundant existential branch).

$Q1.5(n)$ times tend to grow sensibly for large values of n . The reason is that the corresponding pattern's size grows linearly with n , and the curve reflects the complexity of automata-based evaluation for such patterns.

CDuce reported times are the sum of: **the automaton construction time**, determined by the query and the input document's DTD; **the execution time**, during which the results are constructed but not serialized yet; and **the serialization time**. We refine our analysis of CDuce running times in Figure 10, which plots execution and serialization time for $Q1.1(n)$ on *exponential2.xml* and *layered.xml*. The serialization time linearly reflects the total result size, and the number of returned subtrees (the former dominates, however, in the case of *exponential2.xml*). The execution (retrieval) time is almost constant for increasing n , due to the efficient pattern matching operator.

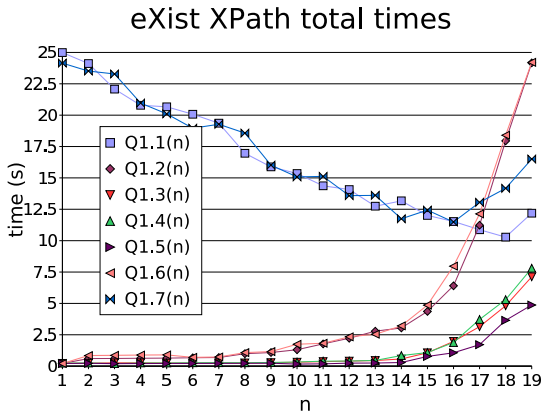


Figure 14: eXist XPath total times.

In the above results (as in all other curves of this paper), no type information was present, and the automaton construction time was in all cases of about 20 ms.

Given CDuce’s strong reliance on types (the evaluation is made by automata, which are built based on types), we re-ran the XPath queries, this time providing CDuce with a precise level-by-level DTD according to which **exponential2.xml** is valid. The results are shown in Figure 11. From **Q1.1(n)** to **Q1.7(n)**, the running times track quite closely those from Figure 9. This demonstrates the relatively low overhead of CDuce’s strong typing, but can also be seen as a sign that type information does not significantly improve query evaluation performance. This points to a source of potential future system improvements.

Figure 12 presents CDuce’s running times on XQueries. **Q2.1(n)** grows very slowly with n , reflecting a moderately increasing navigation time, and a small serialization effort. **Q2.2(n)** running time decreases as the size of the returned subtrees decreases. **Q2.3(n)**, which builds the larger results, failed to run for $n \geq 2$. The reason is that each serialized element in CDuce is written in an OCaml string variable, and the maximum size of such string variables is constrained by the language environment. Elements returned by **Q2.3(n)** grow larger with n , and outgrow at some point the available space. Clearly, this is an engineering problem, whose solution involves the usage of some buffered data structure continuously writing to a file.

Q2.4(n) and **Q2.5(n)** have roughly the same running time, since for CDuce, a newly constructed node is just another value (there is no need to deeply copy trees). Finally, **Q2.6(n)** grows linearly with the result nesting levels.

3.4 eXist

Figure 14 shows eXist’s running times on our XPath queries. These measures were performed on **M2**.

The queries **Q1.1(n)** and **Q1.7(n)** have similar running times, although there is some difference for large n values. **Q1.2(n)** and **Q1.6(n)** are very close, reflecting the negligible influence of the extra existential branch in **Q1.6(n)**. Both **Q1.3(n)** and **Q1.4(n)** are significantly faster than **Q1.2(n)**, showing that eXist is capable of optimizing path

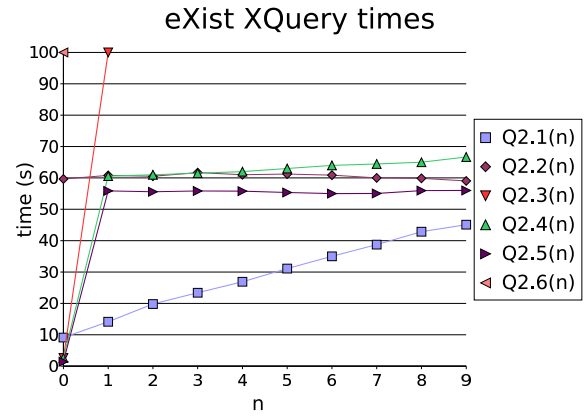


Figure 15: eXist XQuery times.

navigation based on global position predicates. **Q1.5(n)** is the fastest, yet it does feature an exponential growth with large n values.

All the curves tend to grow exponentially for large n values, whether the query returns exponentially many nodes (as **Q1.1(n)** and **Q1.2(n)** do), or a constant number of nodes (as **Q1.5(n)** does). A possible explanation for this is that navigation *traverses* nodes at all layers from 1 to n for each of the queries **Q1.1(n)**-**Q1.7(n)**; the number of such nodes is exponential with n , and the cost of this traversal dominates XPath evaluation costs.

To verify this hypothesis, we take a closer look at the partial times recorded for **Q1.1(n)**, on **exponential2.xml** and also on **layered.xml**. The *locate time* (reported by eXist as **execution**) and the *serialization time* (reported by eXist as **display**) are shown in Figure 13. For readability, we separated the locate times (quite small) from the serialization times.

Figure 13 shows that the locate time grows linearly on the **layered.xml** document, and exponentially with the **exponential2.xml** document. This is consistent with the hypothesis that eXist navigates top-down and thus traverses all nodes from the root to the targets of the XPath expression. Concerning serialization times, Figure 13 shows that it is directly correlated with the *serialized result size* (thus the decrease on the **layered** document), but also with the *number of returned subtrees* (thus the exponential increase on the **exponential2.xml** document).

Figure 15 shows eXist’s running times on the XQueries we considered. For **Q2.3(1)** and **Q2.6(0)** we stopped the execution after failing to get any output in 180 seconds. Clearly, descendent navigation in return clauses is hard to handle for eXist.

For the remaining queries, we record almost constant times for **Q2.2(n)**, **Q2.4(n)** and **Q2.5(n)**.

For **Q2.2(n)**, the execution time grows from 2 seconds ($n=0$) to 27 seconds ($n=9$) as longer paths are navigated, while the serialization time decreases correspondingly as smaller subtrees are output. The two tendencies almost compensate

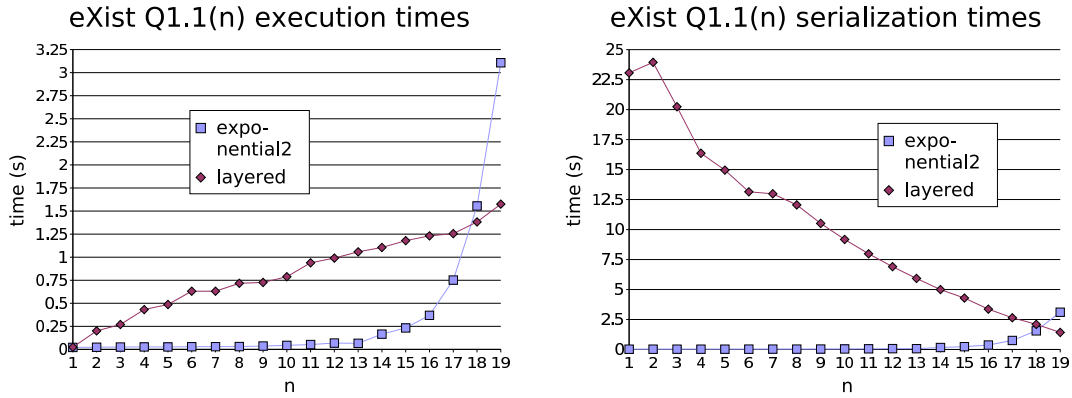


Figure 13: eXist XPath locate and serialization times.

each other, thus the quasi-constant overall running time.

For $Q2.4(n)$ and $Q2.5(n)$, the execution and serialization times are both quite unchanged as n increases. This is a surprising result, for which we do not have a satisfactory explanation. We plan to approach the eXist developers for further explanations.

Finally, $Q2.1(n)$ running times grow linearly with n . The execution time is the main factor here, ranging from 7.3s ($n=0$) to 43s ($n=9$); the serialization time is about unchanged, as same-volume results are returned.

From the detailed running times of $Q2.1(n)$ and $Q2.2(n)$, we may infer that *execution times* as reported by eXist also reflect navigation performed inside `return` clauses. “Execution time” as reported by some other systems may be more restricted (see Section 3.2). This highlights the importance of carefully measuring and interpreting various partial times reported by the engines.

3.5 Galax

These tests were run with galax 0.5.0 on machine M1 without any arguments. Galax is a main memory XQuery engine that implements the XQuery Data Model (XDM). XPath evaluation happens iteratively, i.e., using nested loops with intermediate sorting and duplicate elimination. Constructors are implemented as cursors and should have a limited impact on query performance. The greatest drawback of Galax’s evaluation strategy is the need for materialization of intermediate results of path expressions, which requires a substantial amount of memory.

The results for the XPath queries are depicted in Figure 16. The result for $Q1.1(n)$, $Q1.3(n)$, $Q1.4(n)$ and $Q1.7(n)$ are quite similar. The curious fallback starting at $n = 9$ is believed to be an artifact of Galax serialization, which relies on OCaml pretty printing. This thesis is supported by more detailed monitoring information, given in Figure 17. The similarity of the results indicates that all four kinds of queries are handled in a similar fashion, implying materialization of all the result sequences. The exponential growth of query evaluation time is cancelled out by decreasing serialization overheads for the shrinking results. This effect

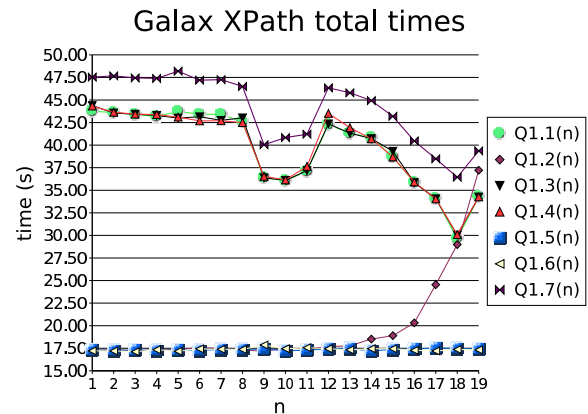


Figure 16: Galax XPath total times.

is not noticeable for $Q1.3(n)$ which selects only attribute nodes. The exponential growth here corresponds with the exponentially increasing result size. The constant behavior of queries $Q1.5(n)$ and $Q1.6(n)$ is an indication that galax uses shortcut evaluation for predicates.

The XQuery results for Galax in Figure 18 show similar performance problems for $Q2.3(n)$ as seen on other platforms, which are related to materializing large intermediate results in main memory. In the case of Galax, queries $Q2.3(n)$ for $n \geq 2$ did not even finish within a reasonable time. Similar to MonetDB, Galax performs much better for $Q2.5(n)$, which returns similar result sizes but has no need for materializing the result in memory before serializing it. The other queries seem to scale along with the result size, since their performance is largely determined by the cost of serialization.

3.6 Saxon

Figure 20 shows Saxon’s running time on our XPath queries. These measures were performed on M4.

A first remark is that $Q1.1(n)$ and $Q1.7(n)$ coincide, demonstrating that descendent navigation is quite efficient in Saxon; these results indicate the presence of a path index and pipelined evaluation. Essentially this is done with a straightforward “nested loop” or rather, a Jackson-inverted nested loop us-

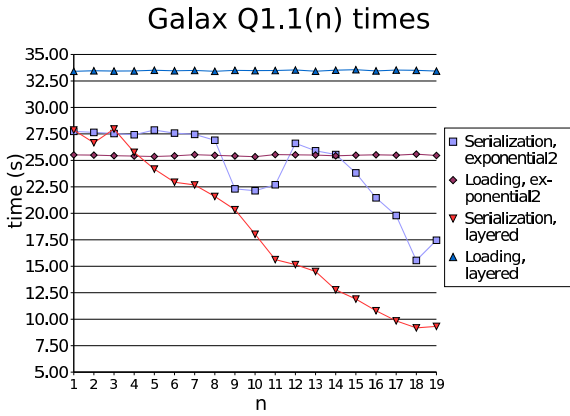


Figure 17: Galaxy XPath partial times.

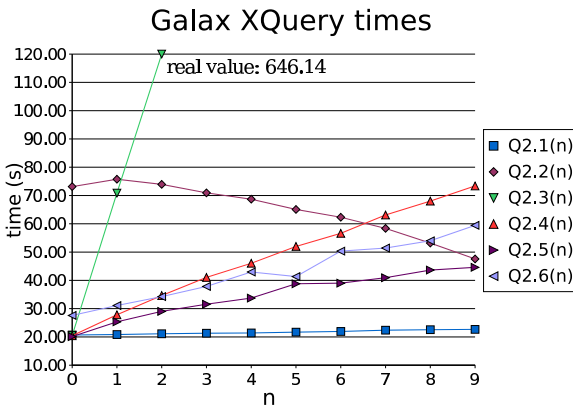


Figure 18: Galaxy XQuery total times.

ing a cascading set of iterators. The main trick that Saxon uses is efficient testing of the element names by comparing integer fingerprints. Also, a lot of effort goes into static analysis of the expression to decide whether a sort and deduplication is needed or not.

Compare queries **Q1.2**(n), returning all **@id** attributes at a given depth, **Q1.3**(n), returning just the first **@id** attribute, and **Q1.4**(n), returning just the last **@id** attribute at a given depth. **Q1.2**(n) is the most expensive. **Q1.4**(n) is faster, but still exhibits an exponential growth with large n values, which allows to infer that all **@id** nodes at a given level are enumerated, prior to choosing the last one. Finally, **Q1.3**(n) is the fastest, and runs in almost constant time as n grows. This hints to a short-circuited evaluation of path queries followed by [1], which stops after the first node at each level is found.

Q1.5(n) is fast and runs almost in constant time, showing the existential predicate of **Q1.5** is efficiently handled. Finally, **Q1.6**(n) has running times very similar to those of **Q1.2**(n), confirming the efficient handling of the existential branch.

Saxon's times on XQuery queries are reported in Figure 19. The standalone, expensive query is **Q2.3**(n), which copies deep trees, starting at locations scattered over the docu-

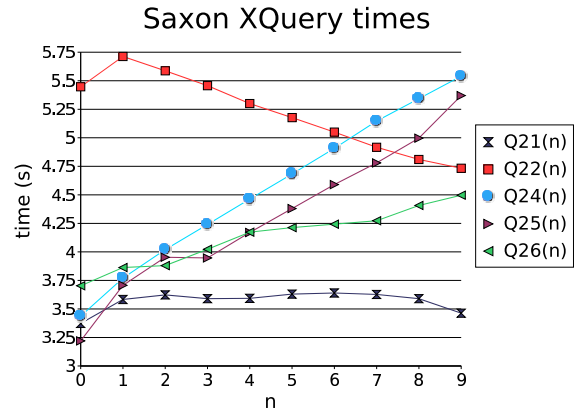
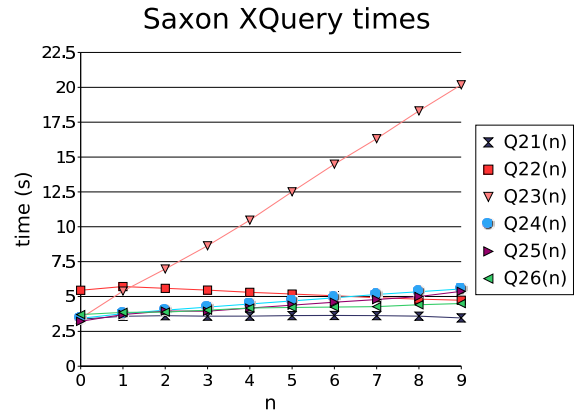


Figure 19: Saxon XQuery total times.

ment. In contrast, **Q2.4**(n), whose copied subtrees are smaller and are found at contiguous locations, is much faster, and **Q2.5**, which does not need to copy subtrees, is yet a little bit faster.

4. CONCLUDING REMARKS

In this section, we summarize some of our experiments' conclusions, and hint to avenues for future work.

4.1 Lessons Learned

A first observation is that benchmarking needs to be performed for a quite substantial number of data points. Failing to do so comes at the risk of not picking up important facts. For instance, in the QizX case exponential behavior of XPath evaluation only becomes apparent for path expressions that are long enough (see Figure 6).

Another important part of our approach is to vary only one benchmark parameter at a time. For instance, varying both document size and query size at the same time may cause effects that cancel each other out, eventually blurring or hiding the impact of the separate changes.

We also believe it is important to decompose query evaluation times into their components, as this is the only way to understand the impact of several parameters on the times.

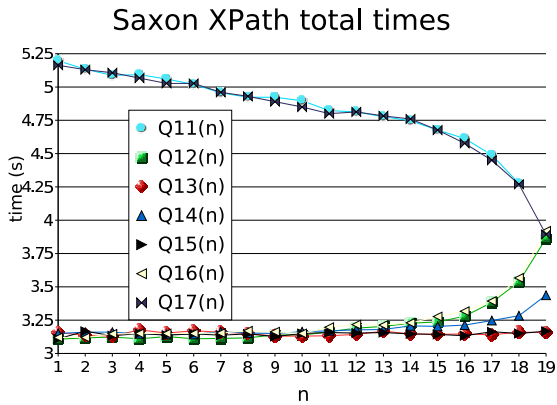


Figure 20: Saxon XPath total times.

For instance, for Galax (Figure 17) and QizX (Figure 6) we have seen that query evaluation times are sometimes dwarfed by the time needed to serialize the result. It would be quite inappropriate to only report query evaluation time for such queries and systems, ignoring serialization. It would just be puzzling to report the overall time, without checking the respective evolution of its components.

Further, it is important to point out that there is no such thing as *the best XQuery processor*. Different systems perform well at different features and under different circumstances. Hence, it is the circumstances that will determine the best solution for a problem. Compare for instance, the results for MonetDB (Figure 2) on **Q1.3(n)** with those for QizX (Figure 5).

Tradeoffs between features and efficiency are also quite visible. For instance, CDuce sacrifices node identity for efficiency. We also remark the the good performance of main memory implementations, notably Saxon, for the moderate-sized documents used in the experiments.

4.2 Future Work

As mentioned earlier, this paper is only a first in a greater effort, which targets a better understanding of the drawbacks and advantages of XPath/XQuery implementation strategies. An important step towards such a better understanding is the development of many more microbenchmarks, to reveal detailed information regarding the performance of implementations at isolated language features. We need to test more systems, such as e.g. BerkeleyDB/XML and Timber, and encourage developers to use our microbenchmarks to identify potential bottlenecks in their implementations (the “bad” cases for Galax and CDuce detected in this paper were news to their implementors !)

Important additional tests that could be performed are system stress tests, that identify the operational boundaries of XPath/XQuery implementations. Other interesting aspects to be tested include:

- the impact of XML Schema information on navigation

queries,

- handling of atomic values, value joins etc.,
- handling of increasing branching factors by XPath,
- handling complex twigs queries etc.

In this paper, we measured the performance of the six systems considered without using any special compiling or execution option. (An exception was made for the eXist server, unable to load the test documents if launched with the default 256 MB of RAM). Clearly, some of the test queries could have been sped up by using different settings and/or options. However, it is difficult to pick one set of special settings that would be fair to all systems. Therefore, we believe measuring with standard setting is the most interesting available performance indicator.

Last but not least, we believe the experimental results here would have been best presented with explanations confirmed by the respective systems’ implementors, as we planned to do, but did not carry out at the time of this writing. We have already gathered feedback from the implementors of Galax, CDuce, and Saxon implementors; we will contact the authors of the other platforms for their opinion and comments.

Acknowledgments. Philippe Michiels is supported by the IWT (Institute for the Encouragement of Innovation by Science and Technology) Flanders, grant number 31016. Ioana Manolescu is partly supported by a French Government “Young Researcher” grant.

The authors thank Loredana Afanasiev for helping to run some of the measures. We are also grateful Giuseppe Castagna, Véronique Benzaken and Alain Frisch, respectively, Michael Kay, for their help in interpreting CDuce, respectively, Saxon results.

5. REFERENCES

- [1] L. Afanasiev, M. Franceschet, M. Marx, and E. Zimuel. XCheck, a platform for benchmarking XQuery engines. Demo proposal, submitted for publication, 2006. <http://staff.science.uva.nl/~lafanasi/xcheck/XCheck.html>.
- [2] L. Afanasiev, I. Manolescu, and P. Michiels. Member: A micro-benchmark repository for XQuery. In *XSym*, volume 3671 of *Lecture Notes in Computer Science*, pages 144–161. Springer, 2005.
- [3] L. Afanasiev, I. Manolescu, and P. Michiels. Member: A micro-benchmark repository for XQuery., 2005. <http://tilcara.science.uva.nl:8080/exist/benchmarks/index.xq>.
- [4] V. Benzaken, G. Castagna, and C. Miachon. A full pattern-based paradigm for XML query processing. In *PADL 05, 7th International Symposium on Practical Aspects of Declarative Languages*, number 3350 in LNCS, pages 235–252. Springer, Jan. 2005.

- [5] S. Boag, D. Chamberlin, M. Fernández, D. Florescu, J. Robie, and J. Siméon. XQuery 1.0 An XML Query Language, W3C Working Draft, April 2005. <http://www.w3.org/TR/xquery>.
- [6] T. Böhme and E. Rahm. Xmach-1: A benchmark for XML data management. In *Proceedings of BTW2001, Oldenburg, 7.-9. Mz, Springer, Berlin*, March 2001.
- [7] S. Bressan, G. Dobbie, Z. Lacroix, M. Lee, Y. Li, U. Nambiar, and B. Wadhwa. X007: Applying 007 benchmark to XML query processing tool. In *CIKM*, pages 167–174. ACM, 2001.
- [8] W. W. W. Consortium. XML path language (XPath) version 2.0 – W3C Working Draft, 2005. <http://www.w3.org/TR/xpath20/>.
- [9] A. Frisch, G. Castagna, and V. Benzaken. Semantic Subtyping. In *Proceedings, Seventeenth Annual IEEE Symposium on Logic in Computer Science*, pages 137–146. IEEE Computer Society Press, 2002.
- [10] H. Hosoya and B. Pierce. XDuce: A typed XML processing language. *ACM Transactions on Internet Technology*, 3(2):117–148, 2003.
- [11] L. Mignet, D. Barbosa, and P. Veltri. The XML web: A first study. In *WWW Conference*, 2003.
- [12] A. R. Schmidt, F. Waas, M. L. Kersten, D. Florescu, M. J. Carey, I. Manolescu, and R. Busse. Why and How to Benchmark XML Databases. *SIGMOD Record*, 3(30):27–32, 2001.
- [13] The CDuce project web site. <http://www.cduce.org>.
- [14] Towards micro-benchmarking XQuery: the experimental data page, 2006. <http://www-rocq.inria.fr/~manolesc/microbenchmarks.html>.
- [15] B. Yao, T. Özsu, and N. Khandelwal. XBench benchmark and performance testing of XML DBMSs. In *ICDE*, pages 621–633. IEEE Computer Society, 2004.