Towards Microbenchmarking XQuery

Philippe Michiels
University of Antwerp

Ioana Manolescu
INRIA Futurs, France

Cédric Miachon
LRI - Université Paris-Sud 11, France

Abstract

A substantial part of the database research field focusses on optimizing XQuery evaluation. However, 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 the first to give an overview of a large subset of the open source XQuery implementations in terms of performance. Several specific XQuery features are tested for each engine on the same hardware to give an impression of the strengths and weaknesses of that implementation. This paper aims at guiding implementors in benchmarking and improving their products.

Key words: XML, query, XQuery, benchmark, microbenchmark, performance

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 [4], there are two possible approaches for comparing systems using benchmarks. Application benchmarks like XMark [18], XMach-1 [8], X007 [9] and XBench [20] 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.

We hope to continue this effort using automated tools such as XCheck [1,5]. 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.

Limitations We take the view that detailed performance measures should document as much as possible the times spent by an XQuery processing engine in each stage of query evaluation - for instance, separating query optimization from query execution and from the XML result serialization time. From our experience, in the case of large-result queries, the serialization time can easily dominate the other evaluation times (sometimes by orders of magnitude)! Unfortunately, some engines do not provide a means to isolate the serialization time from the other execution components, e.g. when execution is streamed. Therefore, we have decided to measure the time to run each query as such, and then the time simply *count the query results, with the hope that the latter time is a reasonable approximation of the time to run the query* without serializing the result.

We are aware of two possible problems of this approach. First, a very simplistic implementation may serialize the results and then count them, thus including, against our will, the serialization time in the counting query running time. Second, a sophisticated implementation may answer counting queries from some data statistics, e.g. histograms or indexes, without actually accessing the data. In this case, the execution time is incomparable with the time to run the simple query, without the count. Despite these shortcomings, we found the counting queries useful in practice as a means to approximate the otherwise inaccessible XML serialization time.

2. Settings

In this section, we present the documents (Section 2.1) and queries (Section 2.2 and 2.4) used for the performance measures in this paper, as well as the rationale for choosing them. Section 2.5 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 [2].

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 [3,4]. 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.

Some of the systems we tested are based on a persistent store, while the others run completely in memory. While we are aware of the inherent limitations 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 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. As such, the stress testing of the systems below has a focus on query scalability, rather than data scalability.

To these documents, we added a family of 10 more documents of varying size, going from 100,000 nodes to 1,000,000 nodes. The purpose of this last document family was to enable an analysis of the way query engines process path queries. The interesting feature of such queries is that a naive implementation requires sorting and duplicate elimination to be performed after each XPath step, whereas efficient engines are able to avoid it. Our 10 chosen documents allow tracing the data scalability of an engine on path queries, thus making inferences about the engine's inner workings.

The document structures are outlined in Figure 1. In this figure, white nodes represent elements, whose names range from **t1** to **t19**; dark nodes represent **@id** attributes. The **exponential2.xml**, **layered.xml** and **mixed.xml** documents have a depth of 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 [17].

- 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 $\mathbf{t}i$.
- 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 $\mathbf{t}i$. Each element labeled $\mathbf{t}i$, with $3 \leq i \leq 18$, has exactly one child labeled $\mathbf{t}(i+1)$. 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

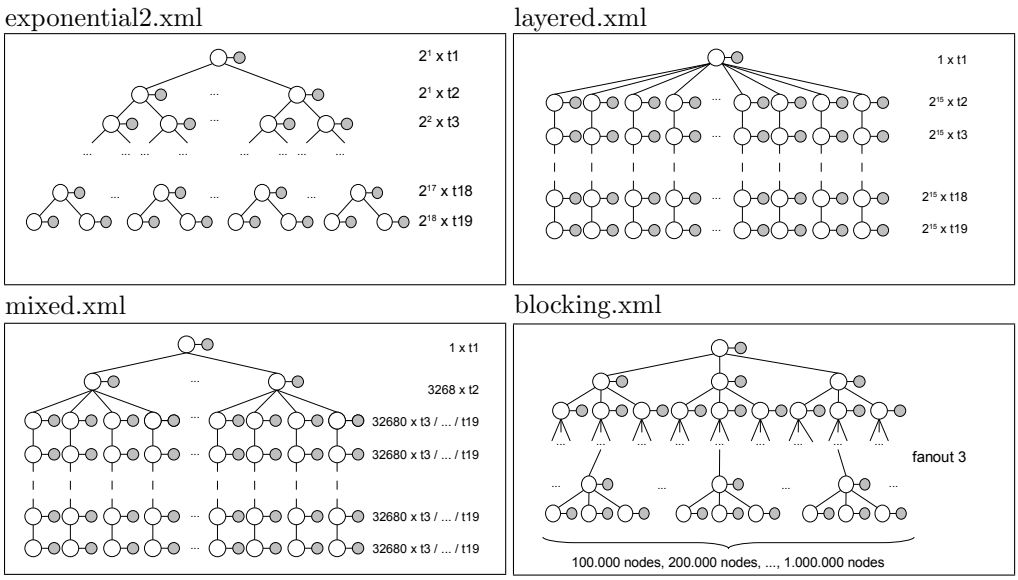

Fig. 1. Outline of the documents **exponential2.xml**, **layered.xml**, **mixed.xml** and **blocking$_n$.xml** used in the measures.

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 **blocking$_n$.xml** documents all have a fixed **fanout of 3** for every node except the leaf nodes. Their depth is variable and depends on the size of the document. Document blocking$_1$.xml contains 100,000 nodes, blocking$_2$.xml contains 200,000 nodes and so on. This corresponds with ten documents whose size vary from 2.09 MB to 21.83 MB.

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.

The third 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}**).

Finally, the series of documents **blocking$_n$.xml** are chosen to have a linearly increasing

amount of nodes in the intermediate results of path expressions that select the entire document tree. A superlinear tendency in the data scalability plot can indicate the presence of sorting operations.

2.2. *XPath Queries*

We measured on the document **exponential2.xml** 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
  This query retrieves nodes at increasing depths of the document. Its output size decreases as the roots of the returned subtrees move lower in the document. This query is designed to test the ability of the query processor to deal with increasing lengths of path expressions. For instance, intermediate materialization will have an increasing performance impact for longer queries. We also measured **Q1.1**(n) on **layered.xml**, which provided some interesting insights when compared to the results on the first document.
- **Q1.2**(n) is: /t1/t2/.../tn/data(@id)
  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.
- **Q1.3**(n) is: (/t1/t2/.../tn)[1]/data(@id)
  This query is used to see if query engines are able to take advantage of the [1] predicate to shortcircuit navigation as soon as a single node is found.
- **Q1.4**(n) is: (/t1/t2/.../tn)[position()=last()]/data(@id)
  This query is similar to **Q1.3**(n), 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 [11]. Measuring both **Q1.3**(n) and **Q1.4**(n) hints at the navigation optimization techniques supported in the engine.
- **Q1.5**(n) is: /t1[t2/.../tn]/data(@id)
  The queries **Q1.5**(n) aim at quantifying the impact of increasingly deeper navigation along existential branches. Once again this verifies whether query engines can get around materializing the predicate results and/or if they are capable to use shortcut evaluation, once a single predicate result has been found.
- **Q1.6**(n) is: /t1[t2/.../tn]/t2/.../tn/data(@id)
  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)
- **Q1.7**(n) is: //tn
  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. *Recursive XPath Queries*

In order to get a better understanding of how exactly XPath expressions are being evaluated, we run the following two queries:

- **Qxpr.1** count(//*/*)
- **Qxpr.2** count(/descendant-or-self::*/descendant-or-self::*)

on all blocking$_n$.xml documents. Examining system performance on these queries allows verifying whether query processors perform sorting and duplicate elimination.

2.4. *XQuery Queries*

We measured on the document *mixed.xml* 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:

- **Q2.1**(n) is:

```
for $x in /t1/t2 return
<res>{
  for $y in $x/*/*/.../*
  return string($y/@id)
}</res>
```

  where the path expression to which **$y** is bound features n child navigation steps starting from **$x**. These queries test the performance of increasingly deeper navigation *in the return clause*, while returning results of modest (and, on **mixed.xml**, constant) size.

- **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**. The queries **Q2.2**(n) combine increasingly deep navigation with decreasingly large subtrees to be copied in the output (recall that XQuery semantics [7] requires the content of newly created elements to be copied).

- **Q2.3**(n) is:

```
for $x in /t1/t2 return
<res>{ $x//t3, $x//t4, ..., $x//t(n + 2) }</res>
```

  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. This allows us to verify how increasingly big and scattered results are dealt with by the processor.

- **Q2.4**(n) is:

```
for $x in /t1/t2 return
<res>{ $x/*[position() ≤ n] }</res>
```

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).

– **Q2.5**(n) is:

```
for $x in /t1/t2 return
$x/*[position() ≤ n]
```

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.

– **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. 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.

2.5. *Hardware and software environment*

Our measures were performed on a single machine. Although this allows absolute comparisons of the tested systems, we are mostly interested in identifying the tendency of each system's running time across increasingly complex queries. The relevant machine parameters are as follows:

Processor 2.00 GHz Pentium 4 (512 KB Cache);
Memory 512 MB DDR SDRAM;
Hard Disk Maxtor DM+8, 40 GB, 2MB buffer, avg. seek time < 10 ms;
Operating System Linux 2.6.12-10-386

The limited amount of memory on the testing machine can be a disadvantage, especially to the main memory engines. Most engines do well in terms of memory for the tests we are running here. Some however, run onto trouble very fast and require a lot of memory swapping. This is why we measure CPU time here, rather than wall clock time. Whenever the XQuery processor fails to produce usable results for a benchmark due to swapping, this is mentioned in the engine's discussion in Section 3. the discussion of the engine. We tested the following systems:

Berkeley DB XML *(v 2.2.13)* – Oracle Berkeley DB XML is an open source, embeddable XML database with XQuery-based access to documents stored in containers and indexed based on their content. Oracle Berkeley DB XML is built on top of Oracle Berkeley DB. Queries can be passed to the database system by means of scripts. We used the command line `dbxml -s dbxml.script` to execute queries, where `dbxml.script` is a straightforward wrapper script that contains the query.

ℂDuce/ℂQL version 0.4.1. ℂDuce (pronounced "seduce") is a general purpose typed functional programming language implemented in OCaml, which is 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 [13]. Recently, a "Select-From-Where" syntax similar to XPath has been added for user convenience on top of ℂDuce [6]; it is translated into ℂDuce. An important characteristic of ℂDuce is its *pattern algebra*, which extends the XDuce [15] 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. 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 [7]. ℂDuce execution proceeds in two stages: first the query is compiled by invoking `cduce –compile query.cd –obj-dir cdo`, then the query is executed by invoking `cduce –run query.cdo -I cdo`.

eXist *(v 1.1rc)* – eXist is an Open Source native XML database, implemented in Java. It features index-based XQuery processing and automatic indexing. The database implements the current XQuery 1.0 working drafts, with exception of the schema import and schema validation features defined as optional in the XQuery specification. Performance measurements were performed by launching an eXist server using the `startup.sh` script part of the distribution and then executing each query by passing

the query to the client: client.sh -F query -c /db -n 10000. Here query is the plain query file, /db is the collection containing the document. Note that eXist limits serialization to a maximum of 10000 nodes, although usually, the engine will run out of memory before being able to compute reasonably large results.

Galax *(v 0.6.10)* – Galax is an open-source OCaml based implementation of XQuery. Galax closely tracks the definition of XQuery 1.0 as specified by the W3C. The command line used to launch the experiment was: galax-run -inline-variables on -streaming-shebang on -factorization on -monitor-time on query.xq.

MonetDB/XQuery *(Pathfinder v 0.12.0)* – MonetDB/XQuery provides an XQuery implementation, which is constructed as an independent compiler, producing code for the MonetDB server backend. Both systems are implemented in C++. We used a 32-bit compilation with no optimizations. Queries where run as follows: pf query.xq — Mserver.

QizX/open *(v 1.1.p2)* – Qizx/open is an open-source Java implementation of the XML Query specifications, working in main memory. We launched the execution by calling `qizxopen_batch.sh query.xq > buf`, where query.xq is a file containing the query, and buf is a temporary buffer file receiving the output. The script qizxopen_batch.sh is part of the distribution.

Qexo/Kawa *(v 1.8)* – Qexo is a partial main-memory implementation of the XML Query language, which attempts to achieve high performance by compiling queries down to Java bytecodes using the Kawa framework. We used the following command line to evaluate the queries:
qexo -f /tmp/qexo_query.xq.

Saxon-B *v 8.8J* – Saxon is a high performance implementation of the XQuery, XPath and XSLT recommendations. The system is implemented in Java and operates in main memory. The command line used was: java -Xmx1024m -cp saxon8.jar net.sf.saxon.Query -t query.xq as recommended by the documentation.

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 (Berkeley DB, eXist and MonetDB), as well as purely in-memory systems (Saxon, Galax, QizX, Qexo 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 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 CPU time of the last 3 (hot) runs*. Although we cover a substantial part of the XQuery implementation field, we are aware that more systems meeting our criteria exist. We plan to extend our tests to such systems in the near future.

3. Benchmark Results

This section presents the results for each of the benchmarks. It is organized as follows.

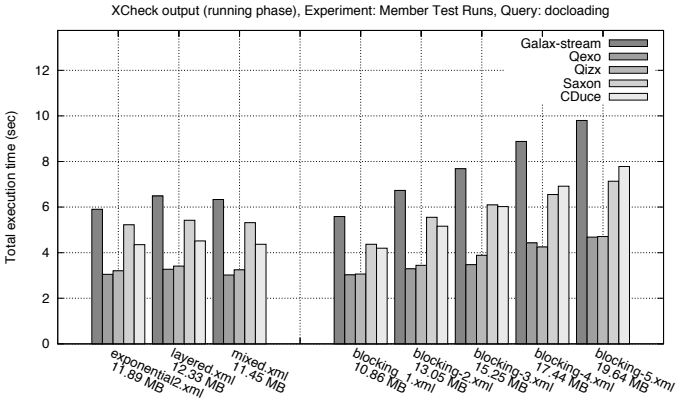

Fig. 2. Document processing times for the main memory XQuery engines.

(i) First, we take a look at XPath query scalability over non-recursive documents. Here the result size for each additional navigation step is kept constant in order to monitor query scalability independently of the result size;

(ii) Secondly, we run additional tests for obtaining more information about the evaluation strategy for XPath expressions. More specifically, we see if and how XQuery processors deal with order and duplicates during XPath evaluation;

(iii) Lastly, we run all XPath (Q1.$x(n)$, $1 \leq x \leq 7$) and XQuery (Q2.$x(n)$, $1 \leq x \leq 6$) queries on exponential2.xml and mixed.xml respectively. We do this for each engine separately and relate the results to the two benchmarks above.

For completeness, we first list the document processing times for main memory processors. These are always included in the measurements below. As Figure 2 points out, processing times scale linear with the document size for al engines except for Saxon, which seems to scale sublinearly. We point out that the document processing time for Galax does not include materialization. Many queries indeed do not require materialization, but in case they do, there will be an extra performance penalty. It is unknown to what extent this holds for other processors.

3.1. *XPath performance for non-recursive documents*

We started by running queries **Q1.1**(n) on the *layered.xml* document for all engines. The queries were run once more, wrapped in a `count()` function call, to eliminate the cost of serialization. The results are depicted in Figure 3.

The eXist engine failed to produce results beyond **Q1.1**(13). The right graph of Figure 3 shows that eXist scales linearly, suggesting that the evaluation of sequences of child steps scales linearly with the size of the output. The same holds for Saxon, Qizx and MonetDB. Examples of XPath algorithms that have these properties are nested loop with delayed sorting and duplicate elimination [12], Staircase Joins [14] as used by

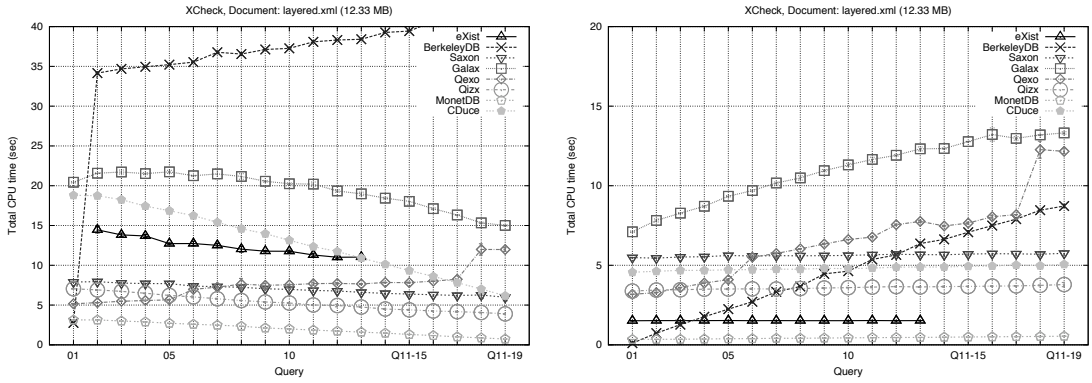

Fig. 3. Results for queries **Q1.1(*n*)** using layered.xml on all processors, including serialization (left) and without serialization (right).

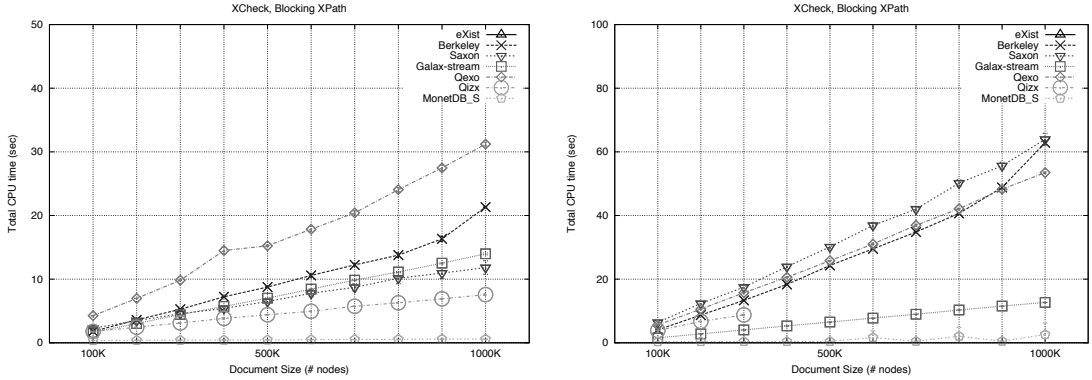

Fig. 4. Results for queries **Qxpr.1** (left) and **Qxpr.2** (right) on blocking$_n$.xml for all XQuery processors.

MonetDB and Twig Joins [10]. Galax uses streaming for this query. Clearly, there is an observable extra cost for each extra step in the pipeline, though Galax scales sublinearly. The streaming approach has the additional advantage that the cost of subsequent operations, like serialization can be amortized over the query evaluation cost. BerkeleyDB shows a similar scalability here, although it unclear what causes this. The Qexo graph is too irregular to draw any conclusion from it. We now continue our discussion engine by engine for each benchmark.

3.2. *XPath, order and duplicates*

We ran **Qxpr.1** and **Qxpr.2** on n blocking*n*.xml documents. The results are depicted in Figure 4. EXist either returned an error or a wrong result for both queries on all

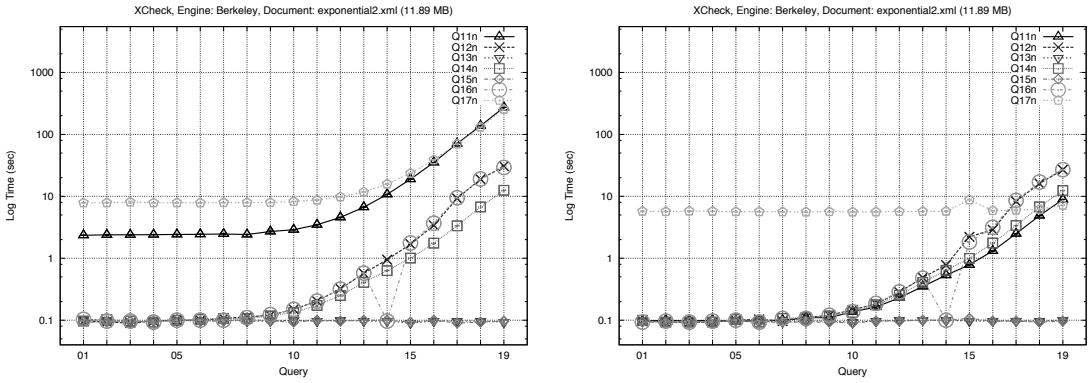

Fig. 5. Results for queries **Q1.1**(n) - **Q1.7**(n), using exponential2.xml on **BerkeleyDB**, including serialization (left) and without serialization (right).

documents. ℂDuce raised an exception mentioning a stack overflow for both queries on every document. This is because descendent navigation is implemented in ℂDuce via a stack that keeps all descendant matched on recursive function calls (and automata), unlike child, parent and sibling navigation, which are implemented by automata alone. Hence, eXist and ℂDuce are excluded from the discussion below.

Qizx failed to execute Qxpr.2 on documents with more than 300,000 nodes. This suggests that Qizx uses an evaluation strategy that does not deal well with duplicates in the intermediate result. Naive nested loop evaluation may cause such behavior. The same holds for BerkeleyDB, which shows a clear superlinear tendency for both queries. Saxon seems to scale linearly for **Qxpr.1**, but scales slightly superlinear for **Qxpr.2**. This may mean that Saxon delays sorting until the end if possible, but this is not helping for the second query. However, the effect of sorting in Saxon is seemingly small, compared to the total query processing time. Qexo behaves in a similar way, but does not seem to delay sorting and duplicate removal for the first query.

The other processors seem to use more advance evaluation strategies that eliminate the need for sorting and duplicate elimination for our queries. Galax uses a streaming approach for this, whereas MonetDB is known to use the staircase join.

3.3. *Berkeley DB*

XPath The BerkeleyDB XQuery implementation successfully completed all XPath queries. In Figure 5, **Q1.3**(n) and **Q1.5**(n) show nearly constant execution times for all n, which suggests that BerkeleyDB has an efficient way of handling existential predicates and some positional predicates. All other queries show quite poor scalability when serialization is involved. Only **Q1.7**(n) scales better when serialization is not included, but the results in Section 3.2 reveal a naïve evaluation strategy for descendant as well. However, child queries like **Q1.1**(n) do not seem to benefit from this index and the plot suggests level-per-level navigation. The graph for **Q1.6**(n) and **Q1.2**(n) almost coincide, meaning that the redundant predicate test induces little or no extra cost. In general, the numbers sug-

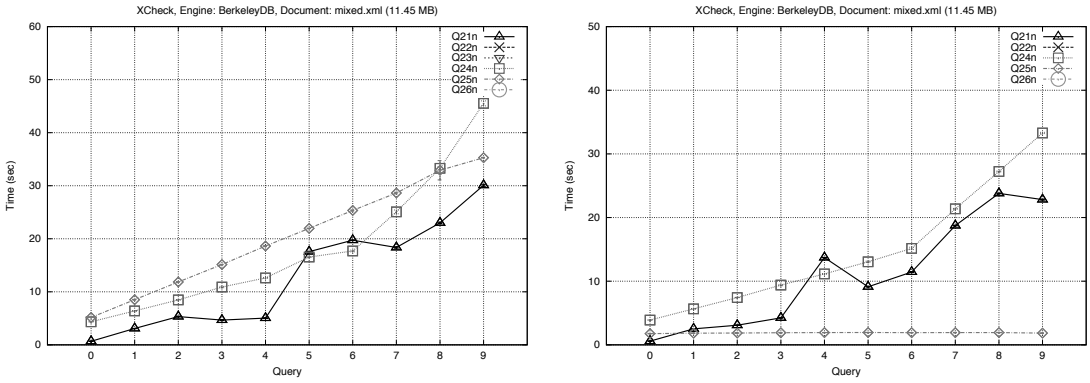

Fig. 6. Results for queries **Q2.1**(n), **Q2.4**(n) and **Q2.5**(n), using mixed.xml on **BerkeleyDB**, including serialization (left) and without serialization (right).

gest that there is a lot of room for improvement with respect to common XPath steps in BerkeleyDB.

XQuery BerkeleyDB experienced serious problems running the XQuery experiments. Notably, the queries **Q2.2**(n), **Q2.3**(n) and **Q2.6**(n) hit the swapping boundary very early on in the experiment. As such, BerkeleyDB failed to produce usable results for these benchmarks within a reasonable time span. Clearly, the BerkeleyDB developers have some memory issues to solve here. Moreover, the difference between **Q2.5**(n) in the left and right graphs shows how expensive serialization is in absolute numbers, although it does scale linearly. **Q2.5**(n) is the only query for which the behavior is comparable to the other engines. The plot for **Q2.1**(n) is a bumpy ride from which there seems little to deduce. BerkeleyDB is the only engine that shows a pronounced superlinear scalability for **Q2.4**(n). The fact that **Q2.5**(n) is the only query without constructors suggests that BerkeleyDB has some issues with those.

3.4. ℂDuce

XPath 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 (Figure 7 left). 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 time and are almost constant. The translation of these XPath queries to ℂQL, combined with the pattern matching concept underlying the system, allows to propagate the [**1**] and [**position()=last()**] predicates at up, thus the efficient evaluation.

The time for **Q1.2**(n) exhibits an exponential growth in Figure 7 as the number of returned/visited 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

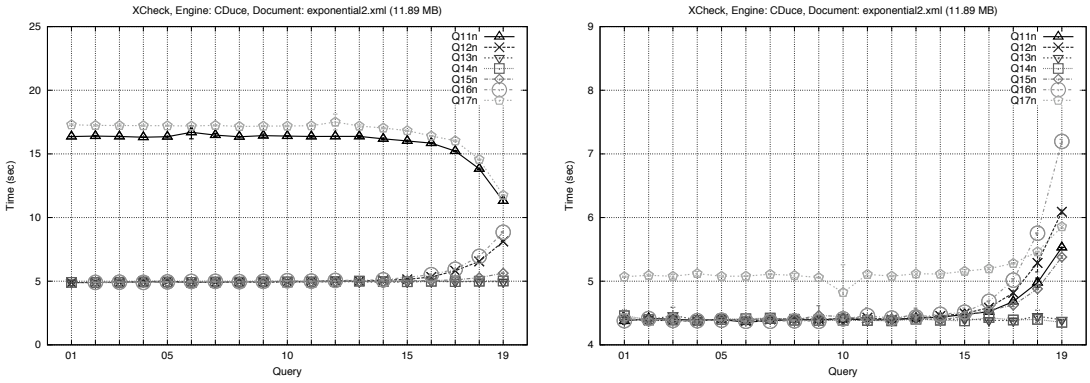

Fig. 7. Results for queries **Q1.1**(n) - **Q1.7**(n), using exponential2.xml on ℂDuce, including serialization (left) and without serialization (right).

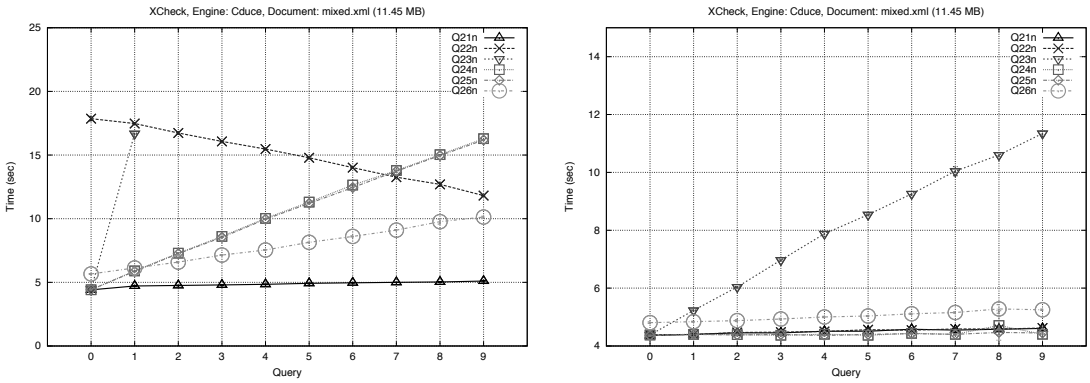

Fig. 8. Results for queries **Q2.1**(n), **Q2.4**(n) and **Q2.5**(n), using mixed.xml on ℂDuce, including serialization (left) and without serialization (right).

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 in the left graph of Figure 7 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**.

These results confirm earlier results discussed in [16], which also gives some details on the impact of the presence of a precise level-by-level DTD according to which **exponential2.xml** validates.

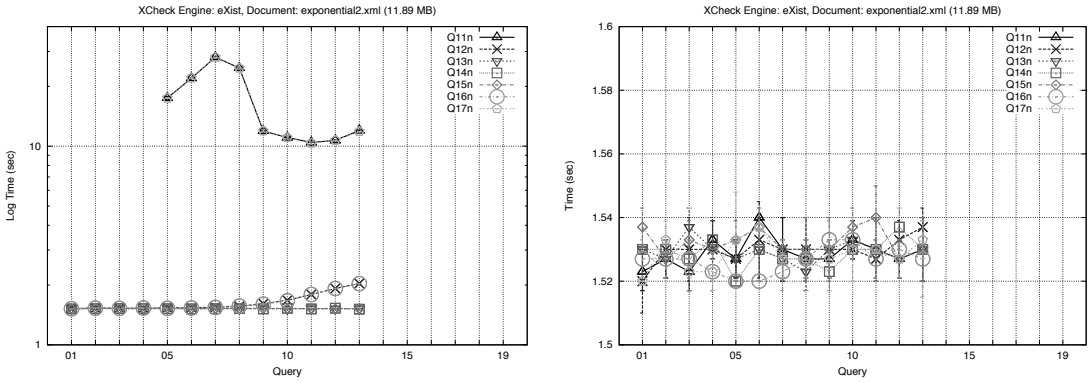

Fig. 9. Results for queries Q1.1n - Q1.7n, using exponential2.xml on **eXist**, including serialization (left) and without serialization (right).

XQuery Figure 8 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.

3.5. *eXist*

XPath The eXist version that was tested consistently failed to produce output for all queries Q1.Xn where $n \geq 14$. The eXist engine did not report any errors, leaving this behavior unexplained. For queries Q1.1n and Q1.7n where $n \leq 4$, eXist failed with an error message indicating that eXist had run out of memory. This may be related to the relatively large size of the result for these queries, indicating the need for materialization when serialization is required. These problems did not occur during prior experiments [16] and we suspect this difference in behavior to be related to the reduced amount of physical memory of the testing machine. Figure 9 shows the running times for the queries that were successfully executed.

Queries Q1.1n and Q1.7n virtually coincide, indicating the use of a path index for simple path queries. The relatively large result size is responsible for the high serialization cost for these queries. The evaluation times without serialization are all located within a 0.05 second interval and it is hard to see any pattern in those timings, in fact query evaluation time is almost constant. Note however that we have observed degrading scalability

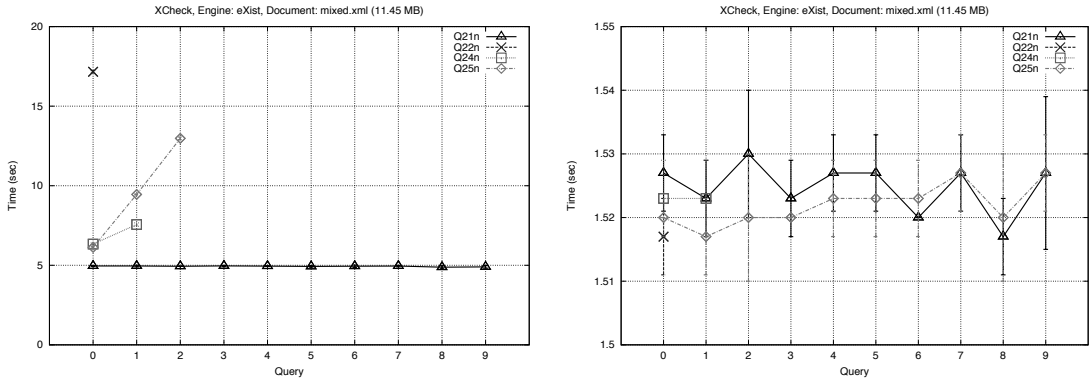

Fig. 10. Results for queries Q2.1n - Q2.6n, using mixed.xml on **eXist**, including serialization (left) and without serialization (right).

of eXist for all queries except Q1.1n and Q1.7n for $n \geq 14$ in earlier experiments [16]. The experiment results in Figure 3 show that the evaluation of simple path expressions like **Q1.1**(n) scale linearly with the size of the result. The lack of results for longer path expressions keeps us from drawing definitive conclusions for the eXist XPath experiments, but given the exponential growth of the result, an equally exponential tendency is to be expected.

XQuery We experienced many problems while running the XQuery benchmarks on eXist. For most queries, i.e., **Q2.2**(n) ($n < 1$), **Q2.3**(n), **Q2.4**(n) ($n < 2$), **Q2.6**(n) eXist reported an error looking as follows: XMLDBException during query: The constructed document fragment exceeded the predefined size limit (current: 10001; allowed: 10000). The query has been killed. which results in the absence of measurements in Figure 10, for those queries. It does come as a surprise that an XQuery engine would limit the output to a certain amount of nodes. Only **Q2.5**(n) succeeded when serialization was disabled. Our earlier measures reported in [16] were performed using twice the amount of memory used here, and they show substantially better results. Thus, the repeated failures of eXist reported here are likely due to the limited available memory.

3.6. *Galax*

XPath Compared to earlier experiments [16] Galax has undergone substantial changes. The most important change is the addition of pull based query evaluation [19]. The only XPath queries that show poor scalability in Figure 11 are **Q1.4**(n) and – to a lesser degree – **Q1.6**(n). Obviously, Galax materializes intermediate results in order to evaluate arbitrary positional predicates, although the position() = 1 predicate seems to be handled fine. The redundant existential predicate in **Q1.6**(n) is not optimized and as a result, Galax is unable to evaluate these queries without materialization. The linear scalability of queries **Q1.1**(n) to **Q1.3**(n) and the constant running time for **Q1.7**(n) is a property of Galax's evaluation strategy. Query **Q1.5**(n) scales well because the intermediate results

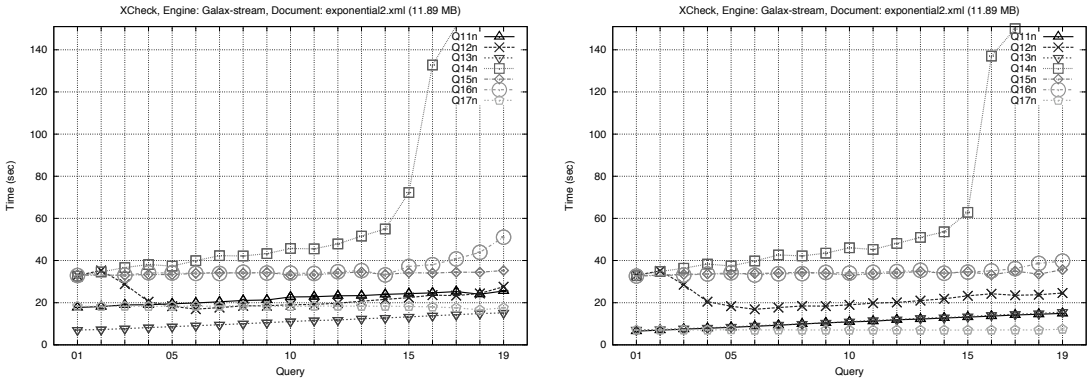

Fig. 11. Results for queries **Q1.1**(n) - **Q1.7**(n), using exponential2.xml on **Galax**, including serialization (left) and without serialization (right).

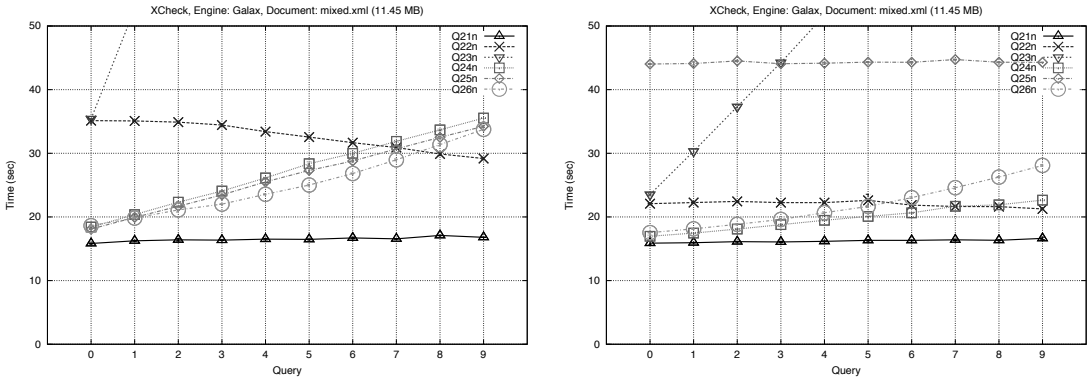

Fig. 12. Results for queries **Q2.1**(n) - **Q2.6**(n), using mixed.xml on **Galax**, including serialization (left) and without serialization (right). The plot for **Q2.3**(n) linearly grows to 200 and 89 seconds for query 9 in the left and right graph respectively.

that need materialization for evaluating the predicate are small. The extreme behavior of **Q1.4**(n) may be related to the fact that the Galax data model has a large memory footprint. Notice that the serialization cost is being amortized over the query evaluation cost. This results in only a limited additional serialization cost for queries **Q1.1**(n) and **Q1.7**(n).

XQuery The results for the XQuery experiments for Galax are given in Figure 12. The difference with the measurements in [16] is surprisingly large. The steepness of the curve for Q2.3n has decreased substantially. The evaluation times for all queries except Q2.3n has dropped well under 50 seconds, compared to more than 70 seconds before.

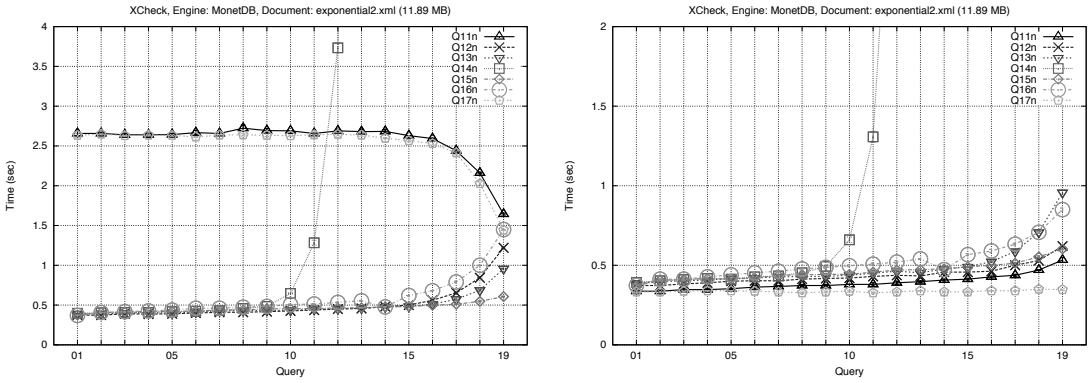

Fig. 13. Results for queries **Q1.1**(n) - **Q1.7**(n), using exponential2.xml on **MonetDB**, including serialization (left) and without serialization (right). The plot for **Q1.4**(n) grows exponentially to 15.4 and 14.8 seconds for query 13 in the left and right graph respectively. For queries larger than Q14(13), MonetDB no longer produces output.

The serialization overhead in the left graph is responsible for the downward curve for **Q2.2**(n), which produces smaller results for larger queries. However, the right hand side curve for **Q2.2**(n) also shows a decrease in evaluation time, indicating that the size of the intermediate results matters here. The pull based execution of Galax allows the serialization cost to be amortized over the rest of the query execution cost, explaining the difference with the more sharply decreasing slope of earlier measurements [16]. When serialization is avoided, queries except **Q2.1**(n), **Q2.2**(n) and **Q2.5**(n) show near constant execution times. The relatively high cost of **Q2.3**(n), compared to other similar queries is due to the inability to avoid materialization, caused by multiple uses of the variable $x. Adding individual navigation steps to **Q2.1**(n) and **Q2.2**(n) hardly influences the total cost of the query, since these steps do not affect the cardinality of the result. Hence, the graphs on the right for **Q2.1**(n) and **Q2.2**(n) show a constant tendency. For all other queries, the output size grows linearly, explaining the linear tendency in the corresponding plots.

The constructor operation in Galax has a streaming interface, i.e., it returns a SAX token cursor. As a result, the Galax compiler picks up a special implementation for the count function, triggering lazy evaluation. In contrast, the query without the constructor does not pick up this approach and uses the default count function, which materializes before actually counting. This explains the staggering difference between **Q2.4**(n) and **Q2.5**(n) in the right graph of Figure 12.

3.7. *MonetDB*

XPath The MonetDB engine seemed to have some problems with the use of the fn:data function. The reported error was "type error: can't cast type 'untypedAtomic*' to type 'node*' ". Therefore, we ran MonetDB on the same queries without the function call. Normally, this should raise an error since the serialization of free standing nodes is not

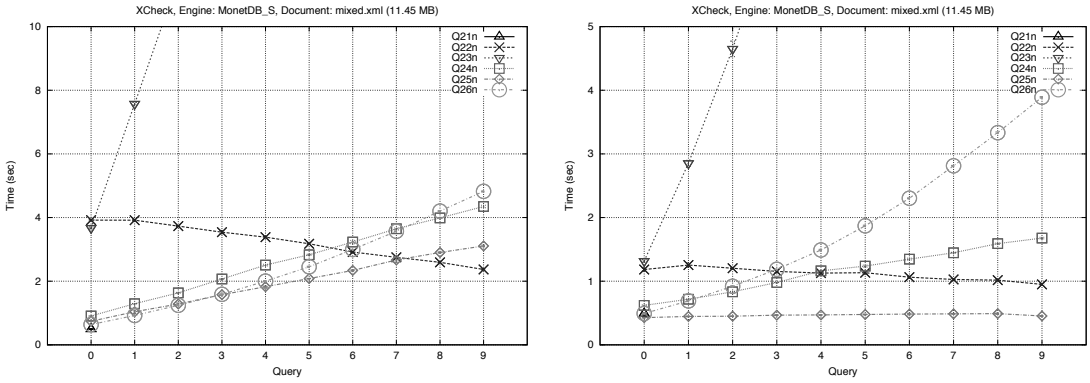

Fig. 14. Results for queries **Q2.1**(n) - **Q2.6**(n), using mixed.xml on **MonetDB**, including serialization (left) and without serialization (right). The plot for **Q2.3**(n) linearly grows to 48.3 and 24.5 seconds for query 9 in the left and right graph respectively.

allowed by the XQuery 1.0 CR, but fortunately MonetDB did not complain about this.

As for many other engines, queries **Q1.4**(n) is problematic for MonetDB, probably due to the need for materializing the results in order to evaluate the predicate. For $n > 13$ MonetDB suddenly stops producing output and exists with the following error: glibc detected free(): invalid pointer: 0x8a957008 . **Q1.7**(n) is the only series of queries showing constant scalability in the absence of serialization. All other queries (including **Q1.1**(n)) show a superlinear tendency for larger values of n. The superlinear tendency for **Q1.1**(n) is explained by pointing out that the staircase join scales with the sum of the input and output nodes for a step, which grow exponentially here by navigating deeper into the document tree. This behavior is propagated to all other queries except **Q1.7**(n) and is confirmed by the experiments at the beginning of this Section and the corresponding results in Figure 3. Despite the superlinear tendency, the predicate queries – with the exception of **Q1.4**(n) – seem to scale well in absolute numbers. However, as **Q1.4**(n) shows, if the intermediate results would grow significantly larger for these queries, performance may suffer more. The small difference in absolute numbers here leaves us inconclusive about how the redundant predicate in **Q1.6**(n) is handled by MonetDB. Finally, the downward curves in the left graph of Figure 13 shows that serialization accounts for a substantial part of query evaluation cost. The smaller the serialized values get, the faster the queries finish.

XQuery MonetDB failed to execute query **Q2.1**(n), reporting a problem with the signature of the string function. The message was type error: no variant of function fn:string accepts the given argument type(s): attribute id { atomic }. Other queries did finish successfully and their results are depicted in Figure 14. The downward slope in the plot of **Q2.2**(n) in the right graph corresponds with the decreasing result size and – as in Galax – must partly be caused by the relatively large intermediate results, since it also shows up when serialization is not performed. There is a slight superlinear tendency for queries **Q2.3**(n) and **Q2.6**(n), but it is too small to draw conclusions from it. Queries **Q2.2**(n),

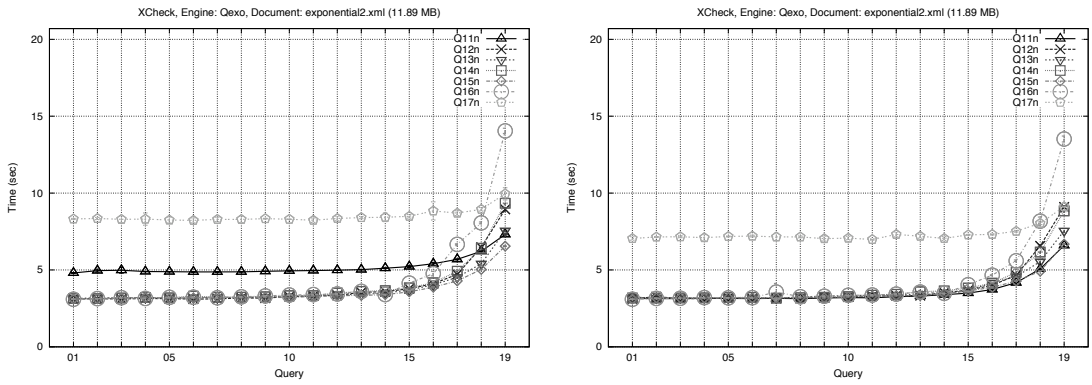

Fig. 15. Results for queries **Q1.1**(n) - **Q1.7**(n), using exponential2.xml on **Qexo**, including serialization (left) and without serialization (right).

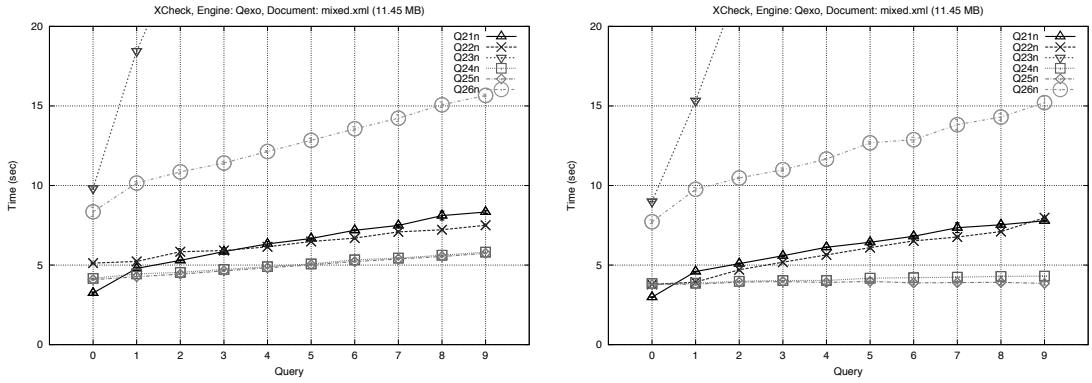

Fig. 16. Results for queries **Q2.1**(n) - **Q2.6**(n), using mixed.xml on **Qexo**, including serialization (left) and without serialization (right). The plot for **Q2.3**(n) linearly grows to 76.5 and 77.7 seconds for query 9 in the left and right graph respectively.

Q2.4(n) and **Q2.5**(n) scale linearly with the size of the output. As the **Q2.4**(n) plot shows, adding a constructor to **Q2.5**(n) changes the behavior of the query from constant to linear. Nesting constructors in **Q2.6**(n) seems considerably more expensive.

3.8. *Qexo*

XPath Like MonetDB, Qexo was unable to evaluate the queries that contained the fn:data function call. In Qexo, the function is simply not implemented. We resorted to running the same queries as we did for MonetDB.

The Qexo results are depicted in Figure 15. All queries have superlinear scalability,

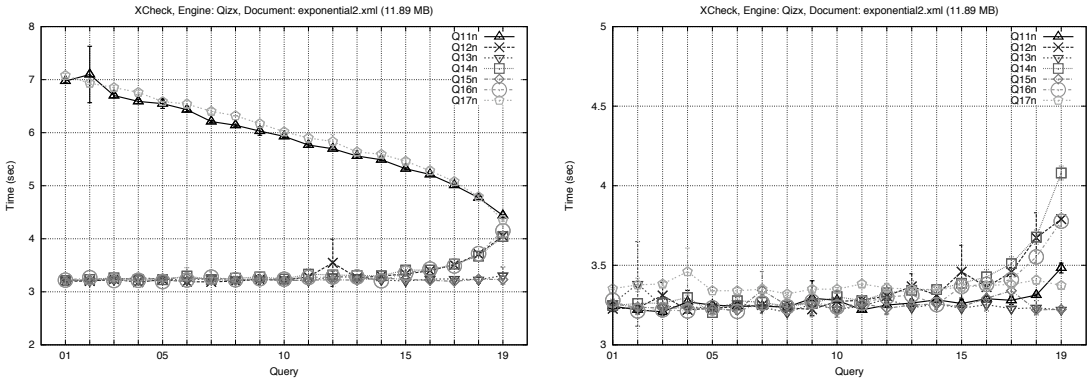

Fig. 17. Results for queries **Q1.1**(n) - **Q1.7**(n), using exponential2.xml on **Qizx**, including serialization (left) and without serialization (right).

where **Q1.4**(n) and **Q1.6**(n) unsurprisingly show the worst scalability. The bad scalability of **Q1.2**(n) (coinciding on the graph with **Q1.4**(n)) is a bit surprising considering the little difference with **Q1.1**(n), i.e., just one extra attribute step. The superlinear behavior for **Q1.7**(n) is unique among all the processors. The experiments in Section 3.2 suggest a naïve XPath evaluation strategy. Despite the superlinear tendency, the Qexo engine performs remarkably well in absolute numbers and shows a very low extra cost for serialization.

XQuery The Qexo results for the XQuery experiments are depicted in Figure 16. The very small difference between serializing and non-serializing queries is again quite striking. Qexo is the only engine for which **Q2.2**(n) does not show a downward slope. It seems that Qexo is less sensitive to the size of the (intermediate) results. This may be in part because query processing time dominates serialization time. There is very little difference between **Q2.4**(n) and **Q2.5**(n), indicating a marginal cost for constructors. Constructors, however, are relatively expensive (consider **Q2.6**(n)).

3.9. *QizX*

XPath Figure 17 shows the results for Qizx. The decreasing times in the left graph for **Q1.1**(n) and **Q1.7**(n) correspond to the decreasing volumes of data that need to be materialized. The scalability of the XPath queries **Q1.1**(n), which use only child steps, is worse than the scalability of the equivalent query, using the descendant step in **Q1.7**(n). The reason for this is that the cost of the separate steps scales linearly with the size of the result, as we have seen in the experiment at the beginning of this Section. It is also interesting to see that **Q1.3**(n) scales better than **Q1.2**(n), suggesting that Qizx deals well with the positional predicate. The redundant predicate in **Q1.6**(n) and the positional predicate in **Q1.4**(n) are clearly not optimized.

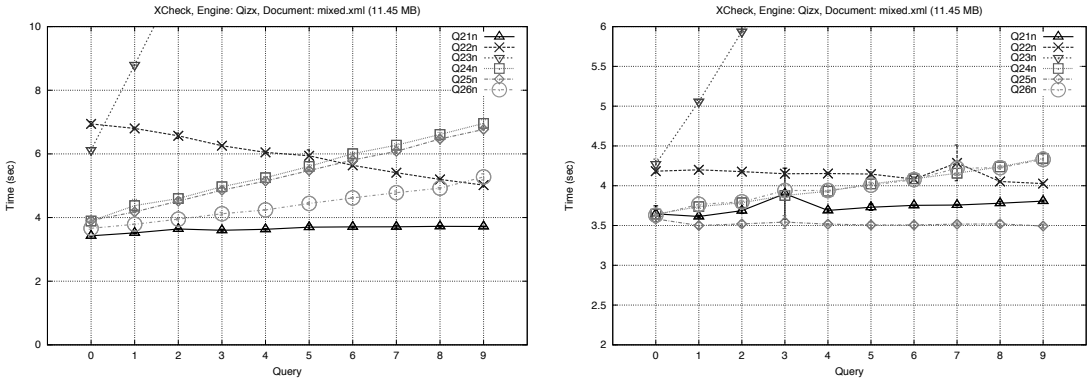

Fig. 18. Results for queries **Q2.1**(n) - **Q2.6**(n), using mixed.xml on **Qizx**, including serialization (left) and without serialization (right). The plot for **Q2.3**(n) linearly grows to 30.4 and 13.5 seconds for query 9 in the left and right graph respectively.

XQuery The Qizx experiments run substantially faster than the Qexo ones, but show surprisingly similar graph shapes in Figure 18. The only difference is that in the absence of serialization there seems to be a very slight superlinear tendency for **Q2.3**(n). The similarity with other main memory processors suggests similar evaluation strategies, in which case the same conclusions apply, i.e., adding a constructor to a query can change its behavior from constant to linear,s and nesting constructors steepens the slope of the query scalability plot. **Q2.3**(n) is consistently more expensive, since it requires materializing quite large intermediate results.

3.10. *Saxon*

XPath The results for Saxon are given in Figure 19. **Q1.1**(n) and **Q1.7**(n) nearly coincide in the both graphs, but start diverging from the x-axis for larger values of n in the right graph, suggesting superlinear behavior. However, the absolute difference between the numbers is too small to draw definitive conclusions on this. The evaluation of child and descendant steps seems to scale linearly with the size of the result. **Q1.4**(n) shows somewhat poorer scalability, although performance is still very good in absolute numbers, especially when compared to some of the secondary storage systems. **Q1.3**(n) performs better than **Q1.2**(n) in the absence of serialization, indicating a smart way to deal with the positional predicate. The redundant predicate in **Q1.6**(n) seems to be handled quite well, since the graph for **Q1.6**(n) almost coincides with that for **Q1.2**(n). Generally speaking, Saxon's combination of scalability and absolute performance make it stand out, even compared to the secondary storage systems.

XQuery The results for the Saxon experiments are given in Figure 20. The graphs are quite similar to those of other engines. Adding a constructor to **Q2.5**(n) (which runs in near-constant time) makes the query suddenly scale linearly. This may indicate that

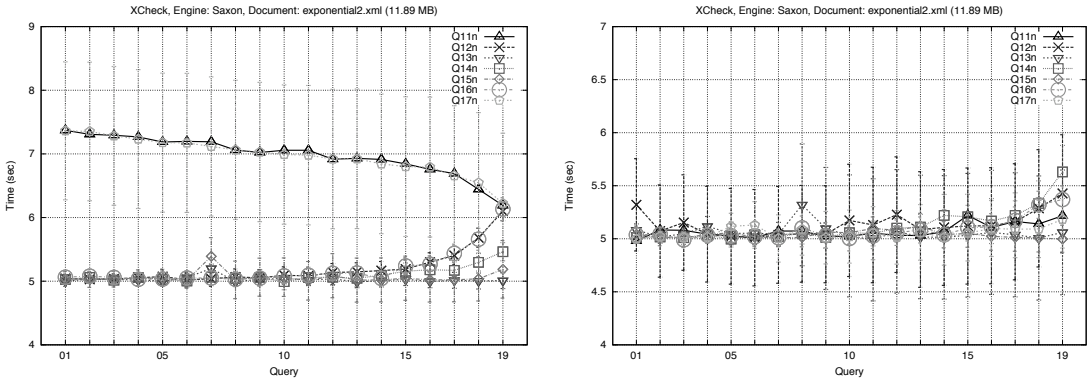

Fig. 19. Results for queries **Q1.1(**n**)** - **Q1.7(**n**)**, using exponential2.xml on **Saxon**, including serialization (left) and without serialization (right).

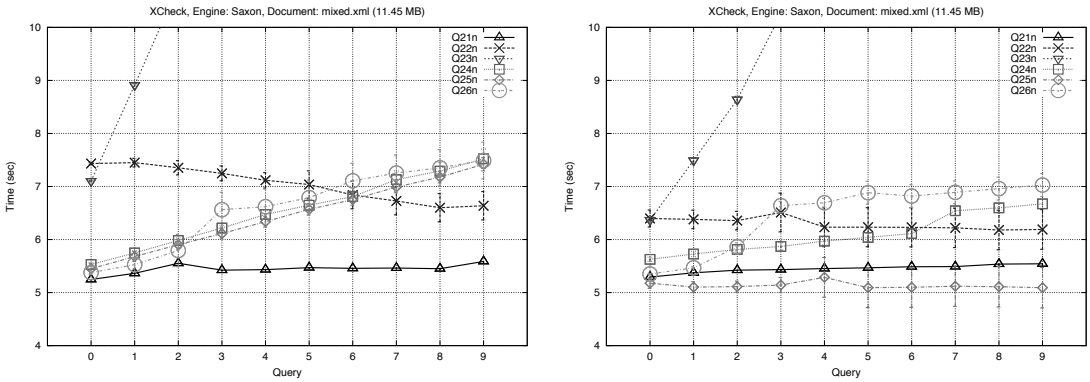

Fig. 20. Results for queries **Q2.1(**n**)** - **Q2.6(**n**)**, using mixed.xml on **Saxon**, including serialization (left) and without serialization (right). The plot for **Q2.3(**n**)** linearly grows to 23.4 and 16.2 seconds for query 9 in the left and right graph respectively.

some buffering is going on. Nesting constructors in **Q2.6(**n**)** is slightly more expensive. The inability to avoid buffering in **Q2.3(**n**)** has obvious consequences.

4. Concluding Remarks

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

4.1. *Methodology Lessons Learned*

A first observation is that benchmarking needs to 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 Saxon case exponential behavior of XPath evaluation only becomes apparent for path expressions that are long enough (see Figure 19).

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. This is why we ran the queries **Q1.1(**n**)** on the layered document in order to see the relation between the scalability of XPath and the size of the result.

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. Especially serialization time and – at least for main memory processors – document loading times are important. For instance, for Galax (Figure 11) and QizX (Figure 17) 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.

A major problem is that an easy way to report an exact decomposition of the query processing time does not always exist. Lazy evaluation poses some serious challenges on this. Furtheron, current system releases make it increasingly difficult or impossible to properly evaluate component times. The names and interpretations of such components time also vary. For instance, while the separation between "execution" and "serialization" seemed clean in all systems, in the case of QizX, this separation is quite arbitrary, as our previous study has shown [16]. Extreme caution is therefore advised to the careful tester. *Few assumptions should be made on what a system does, or what its intermediate times mean, as long as these assumptions have not been checked with the system developers.* This is why we have resorted to *tricks* like using the `count()` function to discover the impact of several XQuery processing phases on the total execution time. Special care should be taken that the tested engines do not use this function application to enhance query performance.

Another unexpected twist came up by running all the implementations on one machine with a rather limited amount of physical memory. Clearly, some implementations – especially BerkeleyDB and eXist – suffer from severe performance penalties for memory swapping or are simply uncapable of completing queries when the system's memory is rather small. This is important information and calls for the extension of the benchmarking effort towards measuring *memory consumption*. Clearly, BerkeleyDB is not a good candidate for XML query processing on embedded systems.

4.2. *Performance Lessons Learned*

Our study does not warrant a claim of having found *the best XQuery processor* among the systems we tested. It seems likely that different systems perform well at different features and under different circumstances. For instance, MonetDB (Figure 13) performs quite badly on **Q1.4(**n**)** compared to Qizx (Figure 17) but it performs better for all other

queries. Galax performs bad in absolute numbers, but has a very favorable scalability for streamable queries.

The impact of the implementation language on performance is quite limited. We might have expected a system implemented in C++ to perform some order of magnitude faster, however Java and OCaml times are very competitive and sometimes even better. This is reason for optimism, as we would indeed prefer algorithms and efficient techniques to have a bigger impact on performance than the simple choice of the development language.

The performance of child axis navigation, a basic XPath feature, can be assessed by inspecting the execution time of **Q1.1(n)** on **exponential2.xml** and **layered.xml**.

- On **exponential2.xml**, the number of returned nodes is in $O(2^n)$, and the number of nodes visited by a naïve XPath evaluation strategy is also in $O(2^n)$.
- On **layered.xml**, the number of returned nodes is constant (and equal to the number of nodes returned by **Q1.1**(14), whereas the number of nodes visited by a naïve XPath evaluation strategy is in $O(n)$.

A linearly increasing execution time for **Q1.1(n)** on **layered.xml** may show that the engine indeed implements a naïve top-down XPath evaluation strategy, which visits all intermediate nodes between the root and the target nodes. This is probably the case for Qexo (Figure 3, right graph). On the contrary, for systems such as MonetDB (Figure 13), we see that the execution time is extremely small (and roughly constant).

Apparently, many engines (BerkeleyDB, Qexo, Qizx, Saxon) use a straightforward XPath implementation strategy that requires some form of *intermediate sorting and duplicate removal*. Others (eXist, Galax, MonetDB) use more specialized algorithms that avoid these blocking operations.

The impact of imprecise navigation (expressed with // steps) can be assessed by comparing **Q1.1(n)** with **Q1.7(n)**. In our tests, these queries are equivalent, while the second uses //. There is little difference among these queries for eXist, Galax, MonetDB and Saxon, which shows that these systems do not need to visit irrelevant nodes when evaluating **Q1.7(n)**. BerkeleyDB, Qexo and Qizx exhibit some performance differences, suggesting that // navigation is suboptimal for these engines.

Early-stop optimizations allow the evaluation of queries like **Q1.3**(n) to proceed much faster than **Q1.2**(n). ℂDuceis a clear winner here. Results for MonetDB and Qexo show that **Q1.3**(n) is actually running slower than **Q1.2**(n) indicating that these optimizations are not picked up.

Existential branches or path predicates exhibit different performance depending on the tested system. This can be assessed by comparing **Q1.2(n)** with **Q1.6(n)**, as well as comparing **Q1.2(n)** with **Q1.5(n)**. Each of these pairs contains equivalent queries on the tested documents. For Galax, MonetDB and Qexo, **Q1.6**(n), featuring an existential test, is more expensive than **Q1.2**(n), whereas for BerkeleyDB and Saxon, the existential branch seems to cause no overhead at all. In Galax, **Q1.5**(n) is more expensive, since it prohibits streaming evaluation, but for the other engines this is not the case, probably due to efficient application of shortcut evaluation.

Result serialization times corresponding to our queries are an important component of total running times, and in some cases, the dominant one (see, for instance Figure 13). This depends of course on the chosen queries and data, however our tests stress the importance this (often ignored) part of query execution times may take. Overall, serialization time reflects the number and size of the returned subtrees. The impact of some OCaml (or system programming) issues leads either to surprising performance variations

(Figure 11), or to unfeasible queries such as for eXist (Figures 9 and 10).

Complex result construction such as required by **Q2.3**(n) and **Q2.6**(n) is related to the problem of serializing large results; **Q2.6**(n) constructs more complex new trees, but of smaller size. Interestingly, Galax (Figure 12) handles **Q2.6(n)** well, whereas BerkeleyDB and eXist do not (missing measurements in Figures 10 and 6).

New node creation is also an interesting feature in our measures. Queries **Q2.4**(n) and **Q2.5**(n) differ only by the fact that **Q2.4**(n) constructs new trees (which requires copying some t2 descendents) whereas **Q2.5**(n) returns these nodes from the original documents. Several strategies are possible. Galax copies the nodes in a lazy manner, avoiding the full materialization of the copied trees. All other engines – except Qexo, apparently – copy and materialize the subtrees under the constructed nodes, causing **Q2.4**(n) to be substantially less performant.

Finally, we also remark the the good performance of main memory implementations, for the moderate-sized documents used in the experiments.

4.3. *Future Work*

This paper is only a starting point 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 also need to encourage developers to use our microbenchmarks, which can help – as proven before – to identify bugs and bottlenecks in their implementations.

Feedback from the experiences in running the experiments in this paper will be used to improve the XCheck platform. One thing that we can take away from the results of our tests is that a more clear separation of query evaluation times and serialization times is desirable. This could be included in the platform mentioned above by doing an additional run without serialization and/or a run for measuring document loading times. Additionally, tools for measuring memory usage would be a very welcome addition.

Important additional tests to be performed are system stress tests, that identify the operational boundaries of XPath and 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 etc.

References

[1] L. Afanasiev, M. Franceschet, M. Marx, and E. Zimuel. Xcheck: A platform for benchmarking XQuery engines (demo). In *VLDB*, 2006.

[2] L. Afanasiev, I. Manolescu, C. Miachon, and P. Michiels. Micro-benchmarking XQuery experiments page, 2006. `http://www-rocq.inria.fr/~manolesc/microbenchmarks.html`.

[3] L. Afanasiev, I. Manolescu, and P. Michiels. Member: A micro-benchmark repository, 2005. `http://ilps.science.uva.nl/Resources/MemBeR/`.

[4] 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.

[5] L. Afanasiev and M. Marx. XCheck, an automated XQuery benchmark tool, 2005. `http://ilps.science.uva.nl/Resources/XCheck`.

[6] 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.

[7] 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`.

[8] 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.

[9] 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.

[10] N. Bruno, N. Koudas, and D. Srivastava. Holistic twig joins: Optimal XML pattern matching. In M. J. Franklin, B. Moon, and A. Ailamaki, editors, *SIGMOD*, pages 310–321. ACM, 2002.

[11] W. W. W. Consortium. XML path language (XPath) version 2.0 – W3C Working Draft, 2005. `http://www.w3.org/TR/xpath20/`.

[12] M. F. Fernández, J. Hidders, P. Michiels, J. Siméon, and R. Vercammen. Optimizing sorting and duplicate elimination in XQuery path expressions. In K. V. Andersen, J. K. Debenham, and R. Wagner, editors, *DEXA*, volume 3588 of *Lecture Notes in Computer Science*, pages 554–563. Springer, 2005.

[13] 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.

[14] T. Grust, M. van Keulen, and J. Teubner. Staircase join: Teach a relational DBMS to watch its (axis) steps. In J. C. Freytag, P. C. Lockemann, S. Abiteboul, M. J. Carey, P. G. Selinger, and A. Heuer, editors, *VLDB*, pages 524–525. Morgan Kaufmann, 2003.

[15] H. Hosoya and B. Pierce. XDuce: A typed XML processing language. *ACM Transactions on Internet Technology*, 3(2):117–148, 2003.

[16] I. Manolescu, C. Miachon, and P. Michiels. Towards XML microbenchmarking. In *First International Workshop on Performance and Evaluation of Data Management Systems (EXPDB)*, Chicago, 2006.

[17] L. Mignet, D. Barbosa, and P. Veltri. The XML web: A first study. In *WWW Conference*, 2003.

[18] 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.

[19] M. Stark, M. Fernández, P. Michiels, and J. Siméon. XQuery streaming á la carte. In *ICDE*, 2007. to appear.

[20] B. Yao, T. Özsu, and N. Khandelwal. XBench benchmark and performance testing of XML DBMSs. In *ICDE*, pages 621–633. IEEE Computer Society, 2004.