

An Analysis of the Current XQuery Benchmarks

Loredana Afanasiev
University of Amsterdam
Kruislaan 403 – 1098 SJ Amsterdam
The Netherlands
lafanasi@science.uva.nl

Maarten Marx
University of Amsterdam
Kruislaan 403 – 1098 SJ Amsterdam
The Netherlands
marx@science.uva.nl

ABSTRACT

This paper presents an extensive survey of the currently publicly available XQuery benchmarks — XMach-1, XMark, X007, the Michigan benchmark, and XBench — from different perspectives. We address three simple questions about these benchmarks: How are they used? What do they measure? What can one learn from using them? Our conclusions are based on an usage analysis, on an in-depth analysis of the benchmark queries, and on experiments run on four XQuery engines: Galax, SaxonB, Qizx/Open, and MonetDB/XQuery.

1. INTRODUCTION

In this paper we provide an extensive survey of the current XQuery benchmarks: XMach-1 [12], XMark [26], X007 [15], the Michigan benchmark (MBench) [24], and XBench [32]. We believe that this survey is valuable for both the developers of new XQuery benchmarks and for (prospective) users of the existing benchmarks. We now briefly describe the different aspects of our study. From now on, we simply call the above five mentioned XQuery benchmarks, *the benchmarks*.

How are the benchmarks used? We first look at the actual use of the benchmarks in the scientific community, as reported in the 2004 and 2005 proceedings of the ICDE, SIGMOD and VLDB conferences. Less than 1/3 of the papers on XQuery processing which provide experimental results use the benchmarks. Section 2 contains the results of this survey.

One of the reasons for this disappointing low number might be the current state of the benchmarks: 38% (62 out of the 163 queries in all the benchmarks) of the queries are syntactically incorrect or use an outdated XQuery dialect. We have transformed all queries into standard W3C XQuery syntax and made them publicly available. Section 3 describes the kind of errors we encountered and the way we corrected them.

What do the benchmarks measure? Having all queries in the same syntax made it possible to analyze them systematically. We had two main questions: (1) What *kind* of queries are in the bench-

marks? and (2) What is the rationale behind the collection of queries making up a benchmark? For (1), we looked at individual queries and determined whether they use real XQuery functionality like sorting or defined functions, or whether they are “just” XPath queries. The latter are further broken down into (the navigational fragments of) XPath 1.0 [27] and XPath 2.0 [28]. Out of the 163 queries only 16 were not already in XPath 2.0 (provided that we abstract away from the new element construction functionality of XQuery). This means that all the others can just as well be used as an XSLT benchmark.

For question (2), we looked at the discriminative value of the queries. A benchmark is a collection of queries. To run and interpret a query is costly. So we analyzed the additional value of each query given the rest of the collection. Roughly, a query has no additional value if there exists another query which yields the same execution time results on all documents and on all XQuery processing engines. Our preliminary results indicate that most of the benchmarks contain queries which are redundant from this perspective.

We also consider the influence of the syntactical constructs used on the execution times. We noticed that a seemingly innocent change of a where-clause into an if-then-else construct can cause increases in execution times by two orders of magnitude. The analysis of the benchmarks is in Section 4.

What can we learn from using these benchmarks? We found that benchmarks are especially suitable for finding the *limits* of an engine. Though hardly reported upon in the literature, an analysis of the errors—syntax, out-of-memory, out-of-time—is very useful. Even with our carefully rewritten queries syntax errors occurred on all engines except Saxon. We found that the most informative manner of experimenting and learning from benchmarks is by running them on several engines and *comparing* the results. Such comparisons are most meaningful if the only difference between two experiments is in the used engine. (Another reason to want the queries to be in a standard syntax). Section 5 contains an analysis of errors and a sample of comparison results.

All the experiments in this paper are run with the help of a testing platform, XCheck¹ [8], on four XQuery engines: Galax [18], SaxonB [21], Qizx/Open [10], and MonetDB/XQuery [23].

To summarize, our main contributions are the following:

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

Proceedings of the First International Workshop on Performance and Evaluation of Data Management Systems (EXPDB 2006), June 30, 2006, Chicago, Illinois, USA.

Copyright 2006 ACM 1-59593-463-4 ...\$5.00.

¹<http://ilps.science.uva.nl/Resources/XCheck/>

- Survey of the use of XQuery benchmarks in the scientific literature.
- Standardization of the current XQuery benchmark queries.
- Syntactical and behavior analysis of the benchmark queries.
- The benchmark results on four XQuery engines: Galax, SaxonB, Qizx/Open, and MonetDB/XQuery.

2. SURVEY OF BENCHMARK USAGE

The main goal of this survey is to find out whether the XQuery benchmarks are used by the database research community for evaluating XQuery engines. If yes, *how* are they used? And if not, *what* do they use? The detailed questions that we answer are:

1. What fraction of the articles about processing XML contains experimental results?
2. How many of the papers with experiments use standard benchmarks and which ones?
3. How many experimental papers contain comparisons with other implementations and which ones?
4. What data sets and queries are used in those experiments that do not use the standard benchmarks?
5. What query languages are used in the experiments?

We considered the 2004 and 2005 conference proceedings of ICDE, SIGMOD and VLDB. For our survey, we selected the articles about XML processing, i.e., the articles that are about XQuery, XPath, XSLT and other closely related XML query languages, their evaluation and optimization techniques. We examined the experimental results in these papers and gathered information about the data sets and the queries used in these experiments.

Detailed statistics including the references to the papers we examine can be found on the web: <http://staff.science.uva.nl/~lafanasi/xcheck/survey.html>. Our main conclusions are (1) that, with the exception of XMark, standard benchmarks are not systematically used for evaluating XQuery engines; (2) instead, the authors design specific experiments to analyze in details proposed query processing techniques, and (3) that the majority of these experiments are based on fragments of XPath 1.0. We now briefly show the answers to the listed five questions.

Questions 1 and 2. Out of 51 papers on XML processing, 41 contained experiments. 13 out of 41 use the standard benchmarks: XMark, XBench, and the XMT

test from the W3C XML Query Test Suit (XQTS). XMark is the absolute winner with 11 papers referring to it, though 5 of the experiments were run on only a few selected benchmark queries.

| | # of papers | with experiments | with standard benchmarks |
|--------|-------------|------------------|--------------------------|
| VLDB | 25 | 22 | 7 |
| SIGMOD | 11 | 9 | 4 |
| ICDE | 15 | 10 | 2 |
| total | 51 | 41 (80%) | 13 (31%) |

| Benchmarks | # of papers |
|-----------------|-------------|
| XMark | 11 |
| XBench | 2 |
| XQTS (XMT) [30] | 2 |

Question 3. 30 out of the 41 papers (73%) that contain experiments, contain comparisons with other query engines, or implementations of different algorithms and techniques. Several authors compare the performance of their implementation with the Galax, X-Hive/DB [7], Xalan [6], and xsltproc [3] query engines. Twigstack [16] is the algorithm used most often for performance comparison on tree patterns.

Question 4. 33 of the papers contain (instead of or besides using the standard benchmarks) experiments on especially designed data sets and/or queries. These are made to thoroughly analyze presented techniques. In most cases, these experiments are based on existing (real life or synthetic) data sets and specifically designed queries (often parametrized). Among the most frequently used data sets are existing XML collections: DBLP, PenntreeBank, SwissProt, and NASA; and synthetically generated data from the XMark and XBench benchmarks. For synthetically generated data conform an XML schema or DTD, authors use the ToXGene generator [11] and IBM XML data generator [2].

| Real life and synthetic data sets | # of papers |
|-----------------------------------|-------------|
| XMark | 22 |
| DBLP [1] | 8 |
| PennTreebank [5] | 6 |
| XBench | 5 |
| SwissProt [22] | 4 |
| NASA [4] | 3 |
| Protein [19] | 3 |
| IMDB | 2 |
| Shakespeare [14] | 2 |
| XMach-1 | 1 |
| others | 9 |

Question 5. The overwhelming majority of the articles experiment with XPath 1.0 queries. The queries express often tree patterns and use only downwards axes.

| Query languages (W3C standards) | # of papers |
|---------------------------------|-------------|
| XPath 1.0 | 25 |
| XQuery | 16 |
| others | 2 |

3. CORRECTING THE BENCHMARKS

Each benchmark consists of a set of queries. Queries are given in a formal language together with a natural language description. The minimal requirement for a benchmark is that the queries are syntactically and semantically correct. *Syntactically correct* queries are queries that do not generate errors (unless they are designed to). If the formal semantics of a query (the query result) does not correspond to the query description then the query is *semantically incorrect*. Another requirement for a benchmark is that its queries can be fed to any XQuery engine without syntactic changes.

In order to comply with these requirements, we first had to ensure that the syntax of the benchmark queries corresponds to the current W3C specifications of XQuery [29]. 62 out of a total of 163 benchmark queries were not valid XQuery queries.² XMach-1 and X007 are old benchmarks and their queries were written in older formalisms. The rest of the benchmarks, except the most cited one, XMark, contained syntactic errors. Some of these errors were raised by static type checking, which probably was not considered when designing the benchmarks. Nevertheless, 38% of the benchmark queries were unusable due to syntactic errors.

²All the errors were detected by running the benchmarks on 4 XQuery engines: Galax, SaxonB, Qizx/Open and MonetDB/XQuery. There still might be other errors that we did not detect.

| Benchmark | errors/total | |
|--------------|--------------|--------|
| XMach-1 | 8/8 | (100%) |
| X007 | 22/22 | (100%) |
| XMark | 0/20 | (0%) |
| Michigan | 9/46 | (20%) |
| XBench TC/SD | 4/17 | (23%) |
| XBench DC/SD | 7/16 | (44%) |
| XBench TC/MD | 5/19 | (26%) |
| XBench DC/MD | 7/15 | (46%) |
| total | 62/163 | (38%) |

Syntactically incorrect or outdated queries.

Secondly, we checked for semantic correctness of the queries. We noticed that, for example, some of the Michigan benchmark queries do not correspond to the semantics given by the natural language descriptions. Correcting such errors is not an easy task, due to rather short and often unclear query descriptions and to the risk of changing the benchmark design.

Thirdly, we had to provide a uniform solution for specifying the input data to the queries. The benchmarks had different ways of indicating the input data. X007 and XMark queries used the `fn:doc()` function with a document URI (usually an absolute file name) as argument. The Michigan benchmark queries invoked the `fn:collection()` function on collection name "mbench". XMach-1 queries did not contain input information and all the XPath paths were absolute. Finally, the XBench benchmark referred to the input by using a new function `input()` that was not formally defined.

When correcting the benchmarks we adhered to the following general guidelines:

1. *not to change the query semantics,*
2. *to keep the changes to the syntactical constructs in the queries to a minimum* (an XQuery query can be written in a dozen different ways and the syntactic constructs used might influence the query performance [9] and Section 4.2 below), and
3. *not to use XQuery features that are under development and not widely supported* (for example, the `collection` feature).

For checking the correctness of our changes we relied on SaxonB. Since there is no XQuery reference implementation we picked SaxonB because it scores best at the XML Query Test Suite [31]. All the corrected queries run now without raising any errors. On other engines errors are still raised, but they are due to engine implementation problems (see Section 5). Below we discuss the changes we made to the benchmark queries. The resulting syntactically correct benchmarks can be found on the web³, together with a detailed description of our changes.

3.1 Syntactic errors

The syntactic errors we encountered were due to:

1. typos, or because of not complying to the current XQuery specifications (X007, XMach-1, and Michigan),

³ <http://staff.science.uva.nl/~lafanasi/xcheck/queries.html>

2. type checking errors (X007, XBench, and Michigan), generated by:
 - (a) applying the child step to items of atomic type,
 - (b) value comparisons applied on operands with incompatible types.
3. static type checking errors (all benchmarks expect XMark), generated by: applying operations and functions on sequences with multiple items while only a singleton and empty sequence is allowed.

We now discuss examples of these types of errors and show how we corrected them.

The X007 queries are written in Kweelt [25] – an enriched and implemented variant of Quilt [17]. Quilt is an XML query language that predates, and is at the basis of, XQuery. To give an example, query Q14:

```
FUNCTION year() { "2002" }
FOR $c IN document("small131.xml")
    /ComplexAssembly/ComplexAssembly
    /ComplexAssembly/ComplexAssembly
    /BaseAssembly/CompositePart
Where $c/@buildDate >= .(year()-1)
RETURN
    <result>
        $c
    </result>
orderby (buildDate DESCENDING)
```

becomes in XQuery:

```
declare namespace my='my-functions';
declare function my:year() as xs:integer
{
    2002
};

for $c in doc("small131.xml")
    /ComplexAssembly/ComplexAssembly
    /ComplexAssembly/ComplexAssembly
    /BaseAssembly/CompositePart
where $c/@buildDate >= my:year()-1
order by $c/@buildDate descending
return
    <result>
        {$c}
    </result>
```

Likewise, the XMach-1 and MBench queries were written in an old XQuery syntax, containing wrong function definitions, FLWOR expressions and built-in functions.

As an example of a type checking error, consider query Q3 of XBench TC/MD:

```
for $a in distinct-values(
    input()/article/prolog/dateline/date)
let $b := input()/article/prolog/
    dateline[date=$a]
return
    <Output>
        <Date>{$a/text()}</Date>
```

```

    <NumberOfArticles>
      {count($b)}
    </NumberOfArticles>
  </Output>

```

The output of the built-in function `fn:distinct-values()` is of atomic type, thus `$a` is of type `xdt:anyAtomicType`. The axis step `child::text()` (line 7) cannot be used when the context item is an atomic value. We corrected this by removing the location step `text()` from the path expression in question.

In query Q6 of Xbench DC/MD,

```

for $ord in input()/order
where some $item in $ord/order_lines/order_line
    satisfies $item/discount_rate gt 0.02
return
    $ord

```

when applying the value comparison `gt`, the left operand is first atomized, then the untyped atomic operand is cast to `xs:string`. Since `xs:string` and `xs:decimal` (the type of right operand) are incomparable types an error is raised. The semantics of this query does not require an error to be raised, thus we consider this a type checking error. To solve this problem, we could explicitly cast the left operand to `xs:decimal` or we could use the general comparison operator `>` that assures the conversion of the untyped operand to the numeric type of the other operand. We took the last option.

Some engines (e.g. MonetDB/XQuery) implement the optional XQuery *static type checking*, which is more strict than the *dynamic type checking*. A number of benchmark queries are not strongly typed and raise static type checking errors. For example, Q6 of Xbench TC/SD,

```

for $word in doc()/dictionary/e
where some $item in $word/ss/s/wp/q
    satisfies $item/qd eq "1900"
return
    $word

```

applies the value comparison `eq` on a XPath expression that might yield a sequence of elements with size larger than one. We added the `fn:zero-or-one` function invocation that tests for cardinality of the left operand of the value comparison:

```

zero-or-one($item/qd) eq "1900"

```

The adjusted query will pass the static type checker and the cardinality test will be done during the query evaluation.

3.2 Semantic errors

During the syntactic processing of the queries we noticed that some benchmark XQuery queries do not correspond with their semantics given by the natural language descriptions. For example, query A2 of the Michigan benchmark says:

“Compute the average value of the `aSixtyFour` attribute of all nodes at each level. The return structure is a tree, with a dummy root and a child for each group.

Each leaf (child) node has one attribute for the level and one attribute for the average value. The number of returned trees is 16.” [24]

The corresponding XQuery query is:

```

declare namespace my='my-functions';
declare function my:one_level($e as element(*)
{
  <average avgSixtyFour="{
    avg(for $a in $e/eNest return $a/@aSixtyFour)
  }">
    aLevel="{zero-or-one($e[1]/@aLevel)+1}">
    {my:one_level($e/eNest)}
  </average>
};
my:one_level(doc()/eNest)

```

This query exhibits two type of errors. First of all, the function `my:one_level()` falls into an infinite recursion when it gets as input an empty sequence. For each tree level of the input document the function is recursively called on the sequence of elements of the next level. For the last level of the tree the function is called on an empty sequence and it ends up in an infinite recursion. Thus, this query does not produce an answer at all, instead an engine error occurs. This can be fixed by adding to the body of the function an if-condition:

```

if(empty($e)) then ()
else
  <average avgSixtyFour="{
    avg(for $a in $e/eNest return $a/@aSixtyFour)
  }">
    aLevel="{zero-or-one($e[1]/@aLevel)+1}">
    {my:one_level($e/eNest)}
  </average>

```

The second error is a mismatch between query semantics and the query description. If the first error is fixed, then the query yields a deep tree with one more level than there are levels in the input document. This is due to the fact that the recursive function call is nested in the result element construction. This does not conform with the query description, which talks about a shallow tree with a dummy root and as many children as levels in the input documents. This can be corrected in two ways: changing the syntax of the query to fit the description, or changing the description to fit the formal semantics of the query. The first option goes against the first general correction guideline we adopted earlier, while the second option goes against preserving the benchmark design. The Michigan benchmark authors explicitly say that the natural language description is the normative query definition. Thus, we leave this unsolved.

A rigorous semantic check of the benchmarks still needs to be done in order to ensure that the benchmark results correspond with their goals and intended measures.

3.3 Specifying the input data

We changed the benchmark queries to access their input data by invoking the `fn:doc()` function. The document URI is left out to be filled at query execution time. Most benchmarks test data-scalability, so they run the same queries on different documents. Thus the input document(s) of a query is a *parameter* which should be filled in by the testing platform.

| Benchmark | # Queries | Core XPath | XPath 1.0 | Nav. XPath 2.0 | XPath 2.0 | sorting | recursive functions | intermediate results |
|--------------|-----------|------------|-----------|----------------|-----------|---------|---------------------|----------------------|
| Michigan | 46 | 12 | 4 | 22 | 5 | 1 | 2 | 0 |
| XMark | 20 | 3 | 3 | 5 | 8 | 1 | 0 | 0 |
| X007 | 22 | 1 | 8 | 6 | 6 | 1 | 0 | 0 |
| XMach | 8 | 0 | 3 | 1 | 2 | 1 | 1 | 0 |
| XBench TC/SD | 17 | 1 | 3 | 5 | 6 | 2 | 0 | 0 |
| XBench TC/MD | 19 | 0 | 1 | 8 | 8 | 2 | 0 | 0 |
| XBench DC/MD | 15 | 0 | 4 | 4 | 4 | 3 | 0 | 0 |
| XBench DC/SD | 16 | 0 | 4 | 5 | 5 | 2 | 0 | 0 |
| total | 163 | 17 | 30 | 56 | 44 | 13 | 3 | 0 |

Table 1: Language analysis of the benchmarks

XMach-1 and two of the XBench benchmarks (TC/MD and DC/MD) are *multi-document scenario* benchmarks, i.e. a query is executed against a collection (of unbounded number) of documents at once without explicitly invoking each of them in the query via the `fn:doc()` function. XQuery has a special built-in function `fn:collection()` to deal with this scenario. This feature is still under development and is considered at risk [29]. More importantly, not all of the engines implement this feature yet. In order to run this scenario in a uniform way, we create an XML document input `collection.xml` that contains the list of documents in the collection and their absolute URIs:

```
<collection>
<doc>/path/doc1.xml</doc>
<doc>/path/doc2.xml</doc>
...
<doc>/path/docn.xml</doc>
</collection>
```

Then we query this document to obtain the sequence of document nodes in the collection. We added the computation of this sequence as a preamble to each query. The result is stored in a variable that is further used instead of the `fn:collection()` function call. So the query:

```
for $a in fn:collection()//tagname
return $a
```

becomes:

```
let $doc :=
  for $d in doc("collection.xml")//doc/text()
  return doc($d)
for $a in $doc//tagname
return $a
```

We now turn to an analysis of the benchmarks themselves.

4. BENCHMARK QUERY ANALYSIS

We first syntactically analyze the individual queries occurring in the benchmarks. After that we look at each benchmark as a whole and analyze the discriminative value of queries within a benchmark. We also look at the influence of the syntactical query formulation on the execution times.

4.1 Syntactical analysis

All queries in the studies benchmarks are formulated in XQuery. But how much of XQuery power do they use? XQuery contains

XPath as a subset. It is interesting to see how much of the queries in the benchmarks can be expressed already in XPath. Here we abstract away from the new element construction, which cannot be expressed in XPath for trivial reasons. The goal of this exercise is to determine how much of the XQuery querying functionality the benchmarks test.

We consider four flavors of XPath:

Core XPath The navigational fragment of XPath 1.0, as defined in [20]. Excluded are the use of position information, all functions and the general comparisons.

XPath 1.0

Navigational XPath 2.0 Excluded are the use of position information and all aggregation and arithmetic functions. But we can do comparisons of data values.

XPath 2.0

Table 1 contains our results. 47 out of 163 queries (29%) are XPath 1.0 queries, 100 (61%) are XPath 2.0 queries, and only 16 (10%) queries cannot be expressed in XPath. 13 of them use sorting and the other 3 use recursive functions.

The next paragraph shows the procedure we followed for rewriting XQuery queries into XPath queries.

How to rewrite an XQuery into an XPath query? Often, XQuery queries use the element construction to produce the output, while XPath 2.0 just copies the retrieved parts of the input document to the output. So how can we rewrite an XQuery into an XPath 2.0 query? We remove the additional XML markup from the generation of the output and just output a sequence of items retrieved by the query. By doing so the structure of the output flattens, but we keep the initial document order of the output. The following example illustrates the process. Consider query QR2 from MBench:

QR2: Select all elements with `aSixtyFour = 2` (Return the element and all its immediate children).

```
for $e in doc()//eNest[@aSixtyFour=2]
return
  <eNest aUnique1="{ $e/@aUnique1 }">
  {
```

| Benchmark | # of queries in original | # of queries in discriminative core | groupings |
|-------------|-----------------------------|--|--------------------------------------|
| XMach-1 | 8 | 7 | 2,1,1,1,1,1, |
| X007 | 22 | 9 | 13,2,1,1,1,1,1,1,1 |
| XMark | 20 | 12 | 9,1,1,1,1,1,1,1,1,1,1,1 |
| MBench | 46 | 17 | 28,2,2,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1 |
| XBench TCSD | 17 | 11 | 5,2,2,1,1,1,1,1,1,1,1 |
| XBench DCSD | 16 | 9 | 6, 3,1,1,1,1,1,1,1 |
| XBench TCMD | 19 | 4 | 16,1,1,1 |
| XBench DCMD | 15 | 7 | 9,1,1,1,1,1,1 |

Table 2: Discriminative cores of the studied benchmarks

```

    for $c in $e/eNest return
      <child aUnique1="{ $c/@aUnique1} ">
    </child>
  }
</eNest>

```

Thus the output returns attributes from the source document in this order:

$e_1, \text{child}(e_1), \dots, \text{child}(e_1), e_2, \text{child}(e_2), \dots, \text{child}(e_2), e_3, \dots$

Whereby the e_i are given in document order, and their children as well. This order is reflected by our rewriting into XPath 2.0:

```

for $e in doc()//eNest[@aSixtyFour=2]
return
  ($e/@aUnique1,
  for $c in $e/eNest return $c/@aUnique1)

```

Note that the second for-loop could be rewritten into a location step:

```

for $e in doc()//eNest[@aSixtyFour=2]
return
  ($e/@aUnique1, $e/eNest/@aUnique1)

```

This is not possible for the first for-loop because of the structure of the required output. Both queries are examples of Navigational XPath 2.0 queries.

4.2 Behavior analysis

We run the benchmarks and use the execution time⁴ results to analyze them. We discuss two topics: the discriminative value of queries and the effect of their syntactical form.

Determining the core of a benchmark. Now we look at the execution time behavior of benchmarks. We view running a set of queries on one document as one experiment and wonder what the individual queries tell us about the query engines. Consider Figure 1 which describes the total execution times of running XBench DC/SD on a 10Mb and a 100 Mb document. (Note the log scale on the y-axis for the 100Mb document). The output of queries 12 and 14 is almost the same on all four engines, on both documents. Query 20 is different because of the missing value for one of the engines.

An important aspect of a benchmark is its discriminative value: what differences in engines' behavior does it convey? Looking at

⁴See Section 5 for a precise definition.

Figure 1 it is clear that after running query 12 we gain no more additional insight from running query 14. So we can say that these queries are equivalent. As another example, Figure 2 shows the behavior of SaxonB and MonetDB/XQuery on the XMark benchmark on increasing document sizes. Note the different scaling on the time axis for the two engines. The hard queries clearly jump out.

Because running a query and interpreting the result is costly we would like that a benchmark does not contain equivalent queries. Table 2 presents our results on determining the *discriminative core* of the studied benchmarks. Each row contains one benchmark. The first column gives the total number of queries; the second the number of queries in its discriminative core, and the third shows the grouping of the original query set into equivalent queries. The groupings are determined as follows: each group contains one query (the “representative query” of the group) which is equivalent to all other queries in the group. The measure for equivalence is as described above: queries q_i and q_j are equivalent if for each query engine E and for each document D , the difference in execution times of the two queries is less than 15% of the average execution time of these two queries. In a formula,

$$\begin{aligned}
 q_i &\equiv q_j \\
 &\Leftrightarrow \\
 \forall E, \forall D & |\text{ExecTime}(q_i, E, D) - \text{ExecTime}(q_j, E, D)| \leq \\
 & 15\%(\text{ExecTime}(q_i, E, D) + \text{ExecTime}(q_j, E, D))/2.
 \end{aligned}$$

Based on the results obtained from running XBench DC/SD on four engines and on the two documents displayed in Figure 1, we obtained the following groups of equivalent queries:

$$\begin{array}{cccccccc}
 \text{Q1, Q8} & \text{Q2, Q5, Q7,} & \text{Q3} & \text{Q4} & \text{Q6} & \text{Q10} & \text{Q11} & \text{Q17} & \text{Q20} \\
 & \text{Q9} & & & & & & & \\
 & & & & & & & & \text{Q19}
 \end{array}$$

A tentative conclusion is that the current benchmarks contain quite some redundancy. If we want that papers publish results on experiments with a complete benchmark, it might be good to trim them down to their core.

The influence of the syntactical form of queries. When rewriting the XQueries into their structurally equivalent XPath 2.0 versions we noticed extreme differences in execution times, depending on the position of the join-test which is normally put in the *where* clause. This indicates that tiny syntactical changes may have dramatic effects on execution times. As a case study we considered the four join-testing queries QJ1–QJ4 in the Michigan benchmark. All of them are designed to test how engines deal with joins on attribute values which possibly require huge intermediate results. Queries QJ1 and QJ2 have the following form:

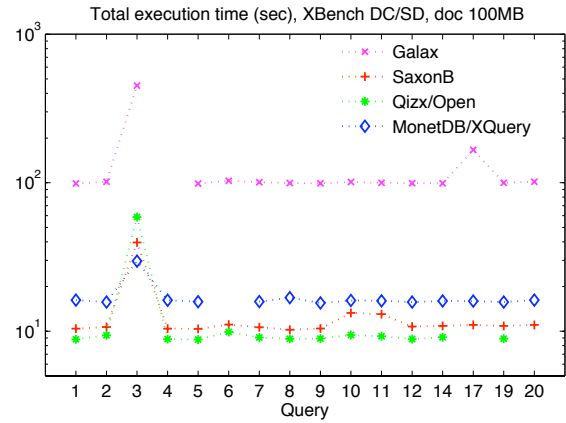
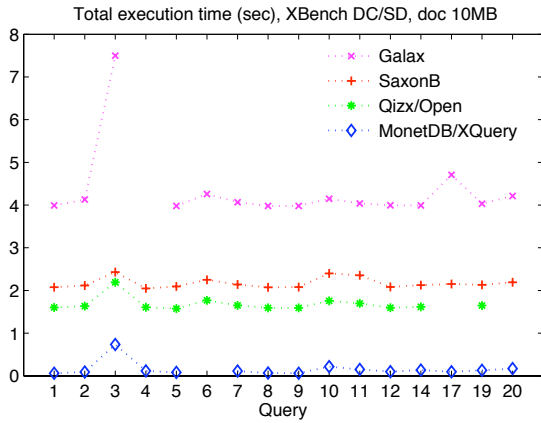
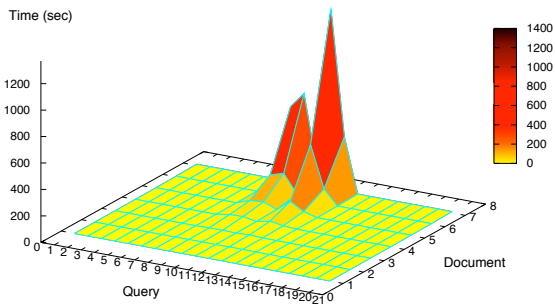


Figure 1: Xbench DC/SD on Galax, SaxonB, Qizx/Open, and MonetDB/XQuery, with 10MB and 100MB documents.

Plot 3D, SaxonB, Total execution time (sec), Benchmark: xmark



Plot 3D, MonetDB, Total execution time (sec), Benchmark: xmark

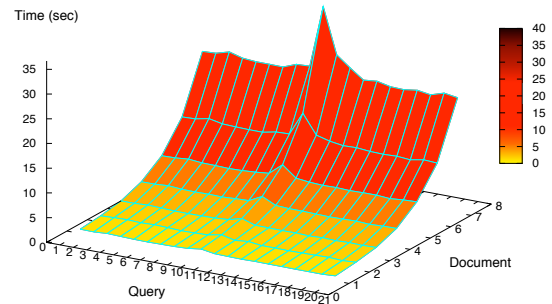


Figure 2: XMark on SaxonB and MonetDB/XQuery on documents ranging from 1.8MB (doc 1) to 114MB (doc 7).

```
for $e1 in Path1 for $e2 in Path2
where test($e1,$e2)
return Result
```

Queries QJ3 and QJ4 are, for no apparent reason, written like this:

```
for $e1 in Path1 for $e2 in Path2
return if (test($e1,$e2)) then Result else ()
```

Obviously the two forms are interchangeable, and there seems no a priori reason for a user to prefer one to the other. We ran the four queries in both the `where` and the `if` syntactical form on the 46Mb document and got quite varying results. One engine crashed on all 8 queries; one engine did not mind the difference in syntax and gave the same times for the different versions; the last engine clearly preferred the `where` formulation, with these results:

| Query | Original syntax | test in where clause | test in if clause |
|-------|-----------------|----------------------|-------------------|
| QJ1 | 3.6 | 3.6 | 330.4 |
| QJ2 | 3.8 | 3.8 | 1405.6 |
| QJ3 | 338.8 | 3.3 | 338.8 |
| QJ4 | 396.1 | 3.5 | 396.1 |

Execution times (in seconds).

These results show the following:

- rewriting queries without the use of element construction is better because it highlights the performance of other syntactic constructs used.
- when testing one aspect in a benchmark with a number of queries (e.g., joins) all queries should have the same syntactic structure, and one should avoid (even harmless looking) variations.
- benchmarks which express the same query in several equivalent ways may yield very useful information regarding query optimization.

Here the main conclusion is that the benchmark that use unintentionally one structural syntactic construct while many are possible may be biased.

This ends the analysis of the benchmarks themselves. In the next section we show some examples of how they can be used to analyze query engines.

5. RUNNING THE BENCHMARKS

In this section we report the results obtained by running the benchmarks on the following four XQuery engines:

- Galax version 0.5.0
- SaxonB version 8.6.1
- Qizx/Open version 1.0
- MonetDB/XQuery version 0.10, 32 bit compilation.

MonetDB/XQuery is an XML/XQuery database system, while the other engines are stand-alone query processors.

We used an Intel(R) Pentium(R) 4 CPU 3.00GHz, with 2026MB of RAM, running Linux version 2.6.12. For the Java applications (SaxonB and Qizx/Open) 1024MB memory size was allocated. We run each query 4 times and we take the average of the last 3 runs. The times reported are CPU times measuring the complete execution of a query including loading and processing the document and serializing the output. All the engines were executed in a command line fashion.

The results reported below are obtained by running all the engines on the following data sizes:

| | doc1/coll1 | doc2/coll2 |
|--------------|------------|------------|
| XMach-1 | 19MB | 179MB |
| X007 | 13MB | 130MB |
| XMark | 14MB | 113MB |
| MBench | 46MB | – |
| XBench TC/SD | 10MB | 105MB |
| XBench TC/MD | 9.5MB | 94MB |
| XBench DC/SD | 11MB | 104MB |
| XBench DC/MD | 16MB | 160MB |

The second document has the largest document size we could run on these engines on this machine, with the condition that at least two engines managed to process the document, and produce an answer.

Figures 3 and 4 contain the results of running the benchmarks on these two documents on the four engines. The individual queries in the benchmark are given on the x-axis and the total execution times on the y-axis. Where appropriate we used a log scale for the execution times. We briefly go through the results.

Comparing engines. In our experience it is most useful to *compare* the behavior of different engines on a benchmark. The comparison can be on absolute times, but also on the *shape* of the graphs. For instance, on X007 and XMark (Figure 3) MonetDB/XQuery shows very constant and robust behavior, though it is not the fastest on all queries. Saxon and Qizx are often faster, but on some queries they use considerably more time or even crash.

Missing values. Comparisons are made difficult because of missing values. A quick glance at the graphs can be misleading because engines may crash on hard queries. Missing values are due to syntax errors and engine crash errors. The latter can have several different causes: out of memory, out of Java heap space, materialization out of bounds, segmentation fault, etc. Table 3 list the syntax errors which still occur: for Qizx/Open all the errors are related to the `fn:zero-or-one()` function. The two errors of Galax are caused by the fact that it does not implement the `preceding` axis. The MonetDB/XQuery errors are diverse, for more details see

our web page.⁵ Table 4 lists the engine crash errors for Galax and MonetDB. For Galax, these were “materialization out of bounds” errors. For MonetDB/XQuery they differed: big intermediate results do not fit in the main-memory nor on the virtual memory; it cannot deal with a big collection of small documents; it is optimized for another usage scenario.

| | SaxonB | Galax | Qizx/Open | MonetDB/XQuery |
|--------------|--------|-------|-----------|----------------|
| XMach-1 | 0 | 0 | 0 | 1 |
| X007 | 0 | 0 | 1 | 2 |
| XMark | 0 | 0 | 4 | 0 |
| MBench | 0 | 0 | 0 | 1 |
| XBench TC/SD | 0 | 1 | 0 | 2 |
| XBench DC/SD | 0 | 1 | 2 | 0 |
| XBench TC/MD | 0 | 0 | 0 | 0 |
| XBench DC/MD | 0 | 0 | 2 | 1 |

Table 3: Syntax errors given by the engines

| | Galax | MonetDB/XQuery |
|--------------|------------------|----------------------|
| XMach-1 | – | all queries on coll2 |
| X007 | Q18 on doc2 | Q5 |
| XMark | Q11, Q12 on doc2 | – |
| MBench | – | QJ1, QJ2, QJ3 QA4 |
| XBench DC/SD | – | Q6 |
| XBench DC/MD | – | all queries on coll2 |

Table 4: Engine crash errors given by Galax and MonetDB.

Ranking the engines? It seemed a nice idea to organize a tournament between engines, like reported in [13]. They consider 16 engines and one benchmark: XMark. The difficulty there is that the data are gathered from the literature and the different circumstances under which the tests were performed make a comparison hard. We wanted to perform a better controlled tournament, now using all available benchmarks. The data in Figures 3 and 4 show that this is not yet feasible. Missing values make the comparisons difficult and possibly unfair. The huge amount of parameters also makes it hard to create a ranking on engines. Already on one benchmark, different document sizes can almost reverse a ranking as happens with XBench TC/MD in Figure 4. The vast number of measurement points (163 queries \times 2 documents \times 4 engines) does not make the task easier either. The strategy to base such a ranking on one benchmark is dangerous: According to [13], MonetDB/XQuery outperforms all other engines on XMark, but it has great difficulties with XMach-1 and XBench DC/MD.

Given the present state of the benchmarks we think that performance rankings of engines are too sensitive to noise and hence not very informative. They are much more useful for developers of engines. For instance, consider Qizx on MBench in Figure 3. It shows very nice almost constant behavior with two peaks on queries J3 and J4. Qizx developers can compare their times with those of Saxon and Galax which give much more stable behavior on the related queries J1–J4. Then they quickly realize that a small investment in query optimization (rewrite an if clause into a where

⁵ <http://staff.science.uva.nl/~lafanasi/xcheck/results.html>

clause, as described in Section 4.2) has great performance benefits.

6. CONCLUSIONS

We analyzed the XQuery benchmarks that are currently available with three main questions in mind: How are they used? What do they measure? What can we learn from using them?

Concerning the benchmark usage, our literature survey shows that, with the exception of XMark, the benchmarks are *not* being used for the evaluation of XQuery processors. One reason for this might be that the benchmarks do not comply anymore with the latest W3C standard of XQuery. We found that 38% of the benchmark queries cannot be run on the current XQuery engines due to syntactic errors. We fixed these errors and we put all the benchmark queries in a standard format. A second reason might be that many of the papers contain an in-depth analysis of a particular XPath/XQuery processing technique, and the standard benchmarks are not suitable for this. In such cases, specialized micro-benchmarks are more appropriate [9].

Concerning what the benchmarks measure, our analysis of the benchmark queries shows that they are not suitable for a thorough investigation of an XQuery engine. Many benchmark queries are similar to each other, using the same XQuery syntactical constructs. Moreover, only 10% of all queries use XQuery properties not present already in XPath 1.0 or XPath 2.0, like sorting and recursive functions. With the exception of XMark and the Michigan benchmark, the rationale behind the benchmark query design is not clear, nor systematically implemented.

So, what can we learn from using the currently available benchmarks? The benchmarks are an excellent starting point for analyzing an XQuery engine. They can give a general view of its performance and quickly spot bottlenecks. Our experiments show that they are useful for checking the maturity of an engine. However, we found that it is very important to check an engine on all benchmarks, instead of only one. The relative performance of engines differs per benchmark. Perhaps this is because each engine is tuned towards a particular scenario captured in a particular benchmark.

From this study we can draw the following conclusions and recommendations:

- The XQuery community will greatly benefit from a well-designed, stable, scalable, and extensible XQuery benchmark in the spirit of XMark.
- Such a new benchmark should consist of clear, well-described categories of queries, in the spirit of the Michigan benchmark, and advocated in [9].
- Each category should be as small as possible (no redundancy) in the number of “information needs” it contains.
- Each information need should be formalized into an XQuery query which focuses on the structure of the query and the desired output. The use of new element construction should be avoided when testing other functionalities. Instead, new queries should be added, that specifically address the construction of new elements. It is also desirable to program the information need in as many different natural ways as possible, using different (all possible) syntactic constructs of XQuery.

- The use of standardized benchmarks (or standardized parts of them) should be encouraged. Experiments must be documented in such a way that they can be repeated by others without too much ado and guessing. A standardized testing platform like XCheck [8] can help in making experiments better comparable.

We hope that this study will facilitate the use of the benchmarks. As future work we want to clean the benchmarks of redundant queries and unify them in the form of a single benchmark collection, easy to use together.

Acknowledgments

This work was sponsored by the Stichting Nationale Computerfaciliteiten (National Computing Facilities Foundation, NCF) for the use of supercomputer facilities, with financial support from the Nederlandse Organisatie voor Wetenschappelijk Onderzoek (Netherlands Organization for Scientific Research, NWO). Loredana Afanasiev is also supported by NWO, grant number 017.001.190.

7. REFERENCES

- [1] DBLP XML records. <http://dblp.uni-trier.de/xml/>.
- [2] IBM Alpha Works XML Generator. <http://www.alphaworks.ibm.com/tech/xmlgeneratorhp>.
- [3] libxslt—the xslt c library for gnome. <http://xmlsoft.org/XSLT/>.
- [4] NASA XML Group. <http://xml.gsfc.nasa.gov/>.
- [5] Penntreebank. <http://www.cs.washington.edu/research/xmldatasets/data/treebank>.
- [6] Xalan. <http://xalan.apache.org/>, 2002.
- [7] X-hive/db. <http://www.x-hive.com/products/db/index.html>, 2005.
- [8] L. Afanasiev, M. Franceschet, M. Marx, and E. Zimuel. XCheck: a Platform for Benchmarking XQuery Engines. Submitted to VLDB 2006 DEMO., June 2006.
- [9] L. Afanasiev, I. Manolescu, and P. Michiels. MemBeR: a micro-benchmark repository for XQuery. In *Proceedings of XSym, Trondheim, Norway*, number 3671 in LNCS, pages 144–161. Springer, August 2005.
- [10] Axyana software. Qizx/open. An open-source Java implementation of XQuery. <http://www.axyana.com/qizxopen>, 2006.
- [11] D. Barbosa, A. O. Mendelzon, J. Keenleyside, and K. A. Lyons. Toxgene: An extensible template-based data generator for xml. In *WebDB*, pages 49–54, 2002.
- [12] T. Böhme and E. Rahm. XMach-1: A benchmark for XML data management. In *Proceedings of BTW2001*, Oldenburg, 2001.
- [13] P. Boncz, T. Grust, M. van Keulen, S. Manegold, J. Rittinger, and J. Teuber. MonetDB/XQuery: a fast XQuery processor powered by a relational engine. In *Proc. SIGMOD 06*, 2006.
- [14] J. Bosak. Shakespeare. <http://www.ibiblio.org/xml/examples/shakespeare/>.
- [15] 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 *Proceedings of CIKM*, pages 167–174, 2001.
- [16] N. Bruno, N. Koudas, and D. Srivastava. Holistic twig joins: optimal xml pattern matching. In *SIGMOD Conference*, pages 310–321, 2002.

- [17] D. Chamberlin, J. Robie, and D. Florescu. Quilt: An XML Query Language for Heterogeneous Data Sources. In *Proceedings of WebDB*, LNCS. Springer-Verlag, 2000.
- [18] M. Fernández, J. Siméon, C. Chen, B. Choi, V. Gapeyev, A. Marian, P. Michiels, N. Onose, D. Petkanics, C. Ré, M. Stark, G. Sur, A. Vyas, and P. Wadler. Galax. The XQuery implementation. <http://www.galaxquery.org>, 2006.
- [19] Georgetown Protein Information Resource. Protein Sequence Database. <http://www.cs.washington.edu/research/xmldatasets/>, 2001.
- [20] G. Gottlob and C. Koch. Monadic Queries over Tree-Structured Data. In *Logic in Computer Science*, pages 189–202, Los Alamitos, CA, USA, July 22–25 2002. IEEE Computer Society.
- [21] M. H. Kay. SaxonB. An XSLT and XQuery processor. <http://saxon.sourceforge.net>, 2006.
- [22] G. Miklau. UW XML Repository. <http://www.cs.washington.edu/research/xmldatasets>.
- [23] MonetDB/XQuery. An XQuery Implementation. <http://monetdb.cwi.nl/XQuery>, 2006.
- [24] K. Runapongsa, J. Patel, H. Jagadish, Y. Chen, and S. Al-Khalifa. The Michigan Benchmark: A Microbenchmark for XML Query Processing Systems. In *Proceedings of EEXTT*, pages 160–161, 2002.
- [25] A. Sahuguet, L. Dupont, and T.-L. Nguyen. Kweelt – a framework to query XML data. <http://kweelt.sourceforge.net/>.
- [26] A. R. Schmidt, F. Waas, M. L. Kersten, M. J. Carey, I. Manolescu, and R. Busse. XMark: A Benchmark for XML Data Management. In *Proceedings of VLDB*, pages 974–985, 2002.
- [27] World Wide Web Consortium. XML Path Language (XPath) Version 1.0. <http://www.w3.org/TR/xquery>, November 1999.
- [28] World Wide Web Consortium. XML Path Language (XPath) 2.0. <http://www.w3.org/TR/xpath20>, November 2005.
- [29] World Wide Web Consortium. XQuery 1.0: An XML Query Language. <http://www.w3.org/TR/xquery>, November 2005.
- [30] World Wide Web Consortium. XQuery Test Suite. <http://www.w3.org/XML/Query/test-suite/>, 2006.
- [31] World Wide Web Consortium. XQuery Test Suite Result Summary. <http://www.w3.org/XML/Query/test-suite/XQTSReport.html>, 2006.
- [32] B. Yao, T. Özsu, and N. Khandelwal. XBench benchmark and performance testing of XML DBMSs. In *Proceedings of ICDE*, pages 621–633, 2004.

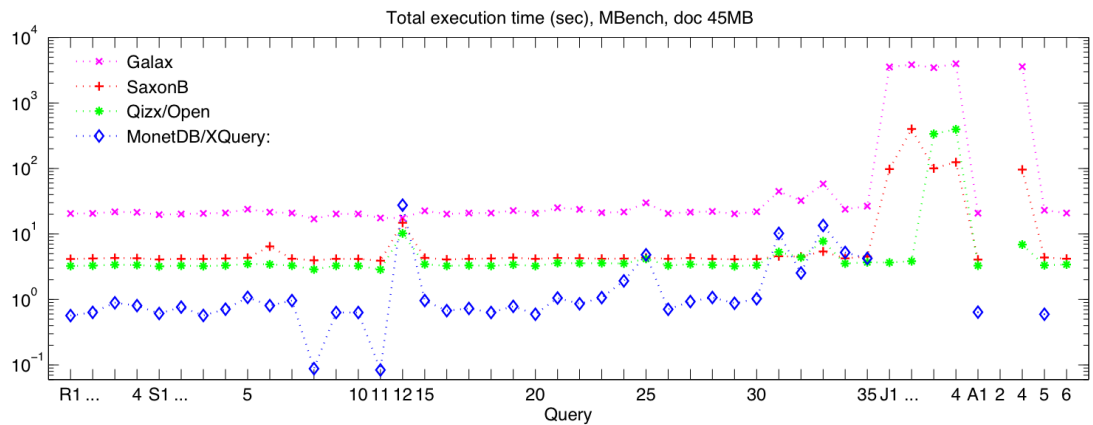
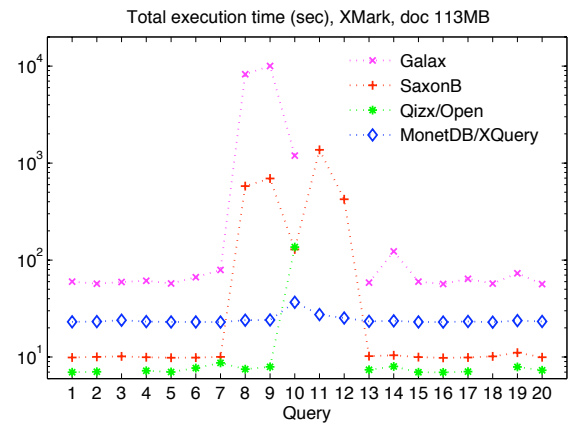
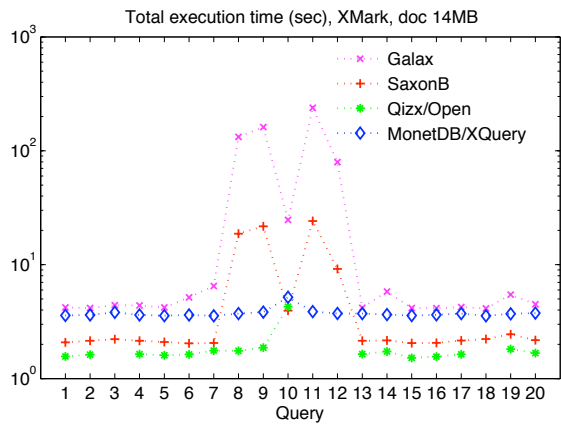
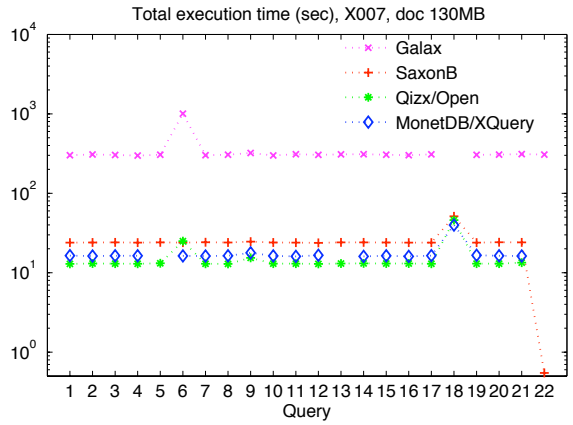
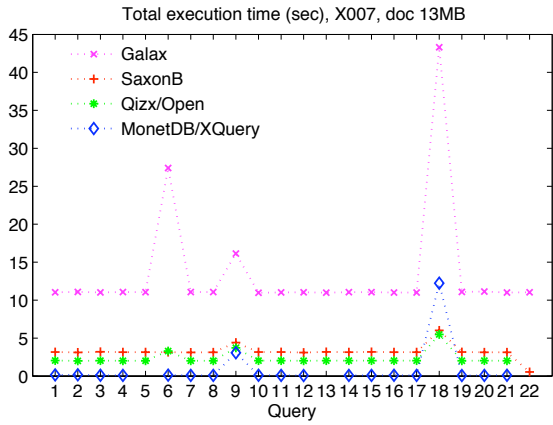
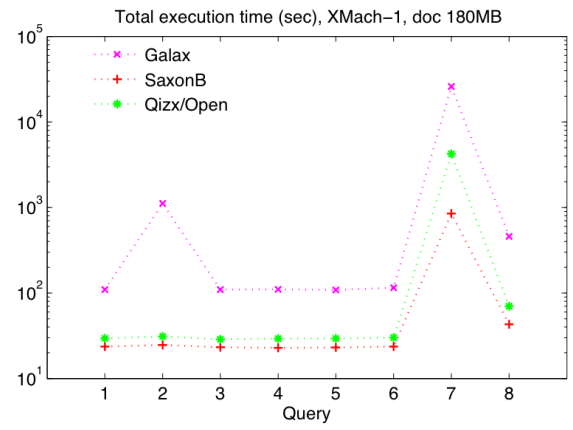
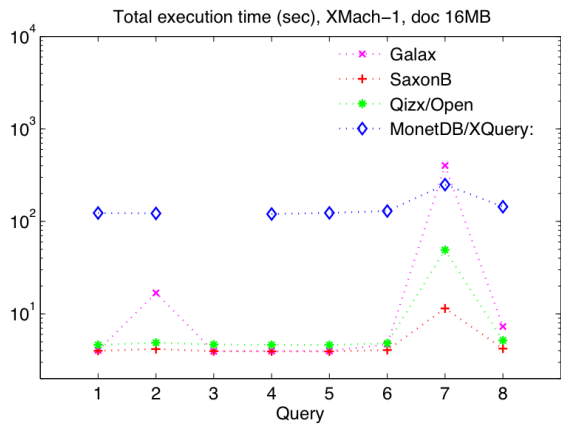


Figure 3: XMach-1, X007, XMark, and XBench on Galax, SaxonB, Qizx/Open, and MonetDB/XQuery.

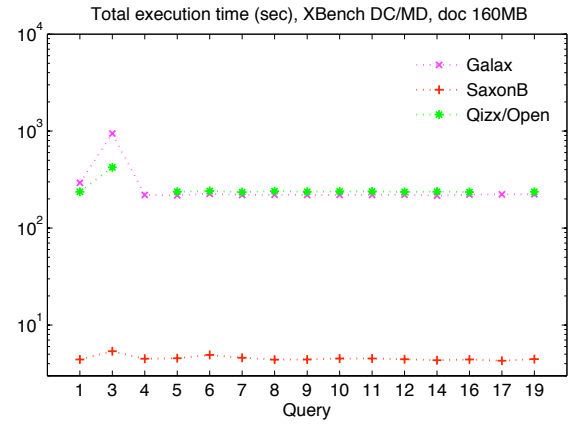
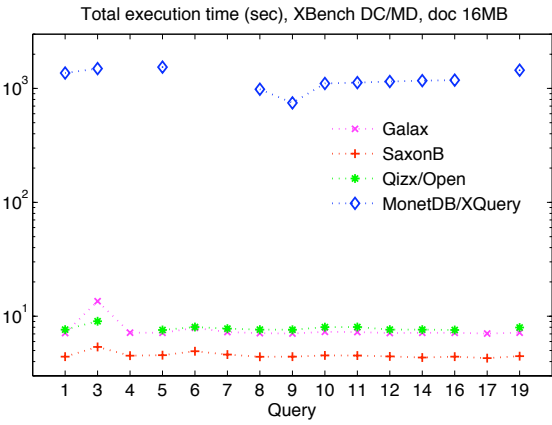
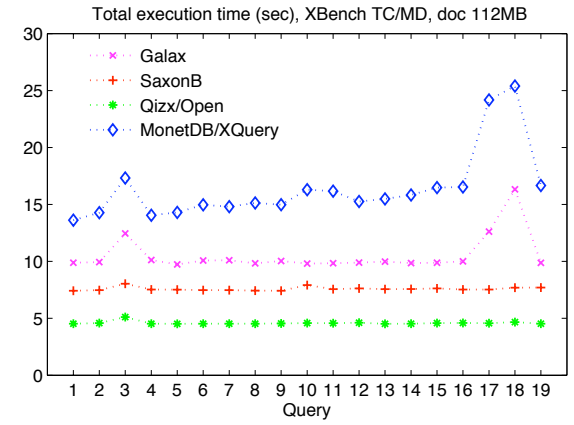
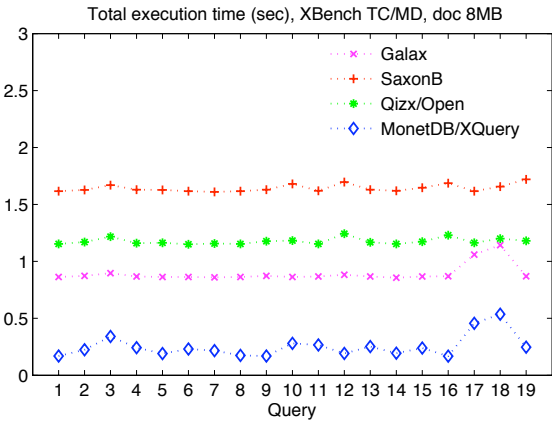
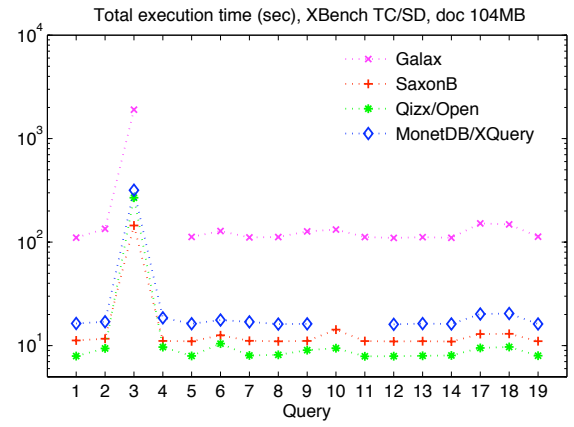
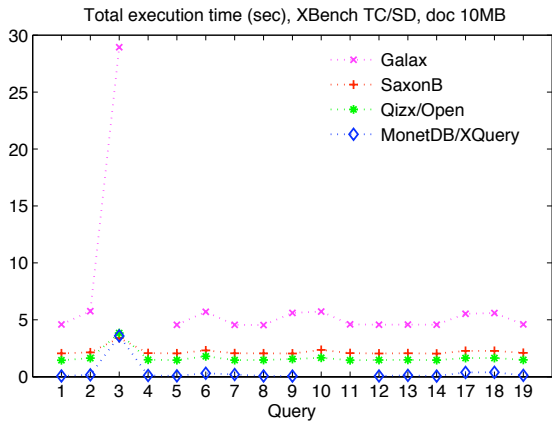


Figure 4: Xbench TC/SD, DC/SD, TC/MD, and DC/MD on Galax, SaxonB, Qizx/Open, and MonetDB/XQuery.