UvA-DARE (Digital Academic Repository)

An analysis of XQuery benchmarks

Afanasiev, L.; Marx, M.

DOI

[10.1016/j.is.2007.05.002](https://doi.org/10.1016/j.is.2007.05.002)

Publication date

2008

Published in

Information systems

[Link to publication](#)

Citation for published version (APA):

Afanasiev, L., & Marx, M. (2008). An analysis of XQuery benchmarks. *Information systems*, 33(2), 155-181. <https://doi.org/10.1016/j.is.2007.05.002>

General rights

It is not permitted to download or to forward/distribute the text or part of it without the consent of the author(s) and/or copyright holder(s), other than for strictly personal, individual use, unless the work is under an open content license (like Creative Commons).

Disclaimer/Complaints regulations

If you believe that digital publication of certain material infringes any of your rights or (privacy) interests, please let the Library know, stating your reasons. In case of a legitimate complaint, the Library will make the material inaccessible and/or remove it from the website. Please Ask the Library: <https://uba.uva.nl/en/contact>, or a letter to: Library of the University of Amsterdam, Secretariat, Singel 425, 1012 WP Amsterdam, The Netherlands. You will be contacted as soon as possible.

An Analysis of XQuery Benchmarks

Loredana Afanasiev^{a,1} Maarten Marx^a

^a*University of Amsterdam*

Kruislaan 403 – 1098 SJ Amsterdam, The Netherlands

Abstract

This paper presents a survey and an analysis of the XQuery benchmark publicly available in 2006 — 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? One focus of our analysis is to determine whether the benchmarks can be used for micro-benchmarking. 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.

Key words: XQuery, Benchmarks, Micro-benchmarks

1 Introduction

In this paper, we provide a survey and an analysis of the XQuery benchmarks publicly available in 2006: XMach-1 [13], XMark [28], X007 [16], the Michigan benchmark (MBench) [26], and XBench [33].² We believe that this analysis and survey are valuable for both the developers of new XQuery benchmarks and for (prospective) users of the existing benchmarks. Henceforth, we call the above mentioned five benchmarks simply *the benchmarks*. We now introduce the questions we address in this study.

Email addresses: lafanasi@science.uva.nl (Loredana Afanasiev), marx@science.uva.nl (Maarten Marx).

¹ Supported by the Netherlands Organization for Scientific Research (NWO) under project number 612.000.207.

² X007 is technically not an XQuery benchmark, since its queries are expressed in a predecessor of XQuery.

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. Fewer than 1/3 of the papers on XML query processing that provide experimental results use the benchmarks. The remaining papers use ad-hoc experiments to evaluate their research results. Section 2 contains the results of this literature survey.

One of the reasons for this limited use might be the current state of the benchmarks: 29% of the queries in the benchmarks contain errors or use outdated XQuery dialects. We have corrected these errors and rewritten all queries into standard W3C XQuery syntax and made these updated queries publicly available. Section 3 describes the kind of errors we encountered and the way we corrected them.

Another reason that the benchmarks are not widely used might be that the benchmarks do not provide suitable measures for the intended purpose of the surveyed experiments. Most of the experiments apply *micro-benchmarking* to test their research results. Micro-benchmarks, as opposed to *application benchmarks*, are benchmarks that focus on thoroughly testing a particular aspect of the query evaluation process, such as the performance of a query optimization technique on a particular language feature [10]. Out of the five benchmarks only the Michigan benchmark (MBench) was designed for micro-benchmarking [26]. The rest of the benchmarks are application benchmarks. We will investigate in detail the micro-benchmarking properties of the MBench queries in Section 5.

What do the benchmarks measure? Our goal is to discover the rationale behind the collection of queries making up a benchmark. Having all queries in the same syntax makes it possible to analyze them systematically. We address this subject with three follow-up questions: (1) How representative of the XQuery language are the benchmark queries? and (2) How concise and focused is the set of queries making up a benchmark? and (3) Are the MBench queries adequate for micro-benchmarking? The first two questions are answered in Section 4; the third, in Section 5.

To answer question (1), we look at functional coverage. The goal is to see how much of XQuery functionality is used in comparison to XPath functionality. If we consider only the retrieval capabilities of XQuery, only 16 of the 163 benchmark queries could not be expressed in XPath 2.0. This indicates that the benchmark queries are heavily biased toward testing XPath rather than XQuery.

For question (2), we analyse each benchmark as a whole. Running and interpreting each query in a benchmark can be 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 is another query that yields the same execution times on all documents on all XQuery processing engines. Our preliminary results indicate that for most of the benchmarks, a well-chosen subset of its queries give the same information as

the entire benchmark.

We give a partial answer to question (3) by looking at the MBench queries that test the performance of value-based joins. Based on initial experiments, we identify two different sub-tasks with the evaluation of joins expressed in XQuery: one is to implement an efficient algorithm for value-based joins, and the other is to detect when a query contains a value-based join. XQuery is syntactically a rich language, and joins can be expressed in many different ways. Thus the second task is not a trivial one. As a follow up work of our investigation and a slight deviation from the main goal of this paper, we extend the set of join queries of MBench to a micro-benchmark testing for the join detection of an engine. We present this micro-benchmark in detail, and apply it on four different engines.

What can we learn from using these benchmarks? We found that benchmarks are especially suitable for finding the *limits* of an engine. Though scientific papers rarely give error analyses, an analysis of the errors raised by an engine—syntax, out-of-memory, out-of-time—is very useful. Even on our rewritten queries, all the engines except SaxonB raised syntax errors, which indicates their non-compliance to the W3C XQuery standard. We found that the most informative manner of experimenting with and learning from benchmarks is by running them on several engines and *comparing* the results. Such comparisons are meaningful only if they are executed under the same conditions for each engine; for example, without modifying the syntax of the query when running them on different engines. Section 6 contains an analysis of the errors we found in the benchmark queries and a sample of comparison results.

All the experiments in this paper are run with the help of a testing platform, XCheck³ [9], on four XQuery engines: Galax [19], SaxonB [22], Qizx/Open [11], and MonetDB/XQuery [25].

To summarize, our main contributions are the following:

- Survey of the use of XQuery benchmarks in the scientific literature. (Section 2)
- Standardization of the current XQuery benchmark queries. (Section 3)
- Analysis of the benchmark queries: What do they measure? (Section 4)
- A micro-benchmark (extending the set of join queries of MBench) testing join detection. (Section 5)
- An example of a comparative benchmark study on four XQuery engines. (Section 6)

Our conclusions are in Section 7.

³ <http://ilps.science.uva.nl/Resources/XCheck/>

2 Survey of benchmark usage

In this section we present the results of a survey we conducted about the usage of the XQuery benchmarks. The main goal of this survey is to find out whether the 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) How many of the articles about processing XML contain 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://ilps.science.uva.nl/Resources/MemBeR/other-benchmarks/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, X Bench, and the XMT test from the W3C XML Query Test Suit (XQTS) [29]. 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.

Benchmarks	# of papers
XMark	11
X Bench	2
XQTS (XMT)	2

	# 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%)

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. The most often used algorithm for performance comparison on tree patterns is TwigStack [17].

Question 4. 33 of the papers contain (instead of or besides using the standard benchmarks) experiments on ad hoc data sets and/or queries. These are made for focused and thorough analysis of the 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 such as DBLP [1], PenntreeBank [5], SwissProt [24], and NASA [4], and synthetically generated data from the XMark and XBench benchmarks. For synthetically generated data conforming to an XML schema or DTD, authors use the ToXGene generator [12] and IBM XML data generator [2].

Real life and synthetic data sets	# of papers
XMark	22
DBLP	8
PennTreebank	6
XBench	5
SwissProt	4
NASA	3
Protein [20]	3
IMDB	2
Shakespeare [15]	2
XMach-1	1
others	9

Question 5. Out of the 33 papers containing experiments on ad hoc data sets and/or queries, 25 use XPath 1.0 queries, 3 use XQuery queries, one uses queries expressed in a modified version of XQuery, and one paper uses SQL queries. In the remaining 3 papers, the language used to express the queries is unspecified. The queries express often tree patterns and use only downwards axes.

Query language	# of papers
XPath 1.0	25
XQuery	3
modified version of XQuery	1
SQL	1
unspecified	3

3 Correcting the benchmarks queries

In the previous section, we saw that the benchmarks are not systematically used for evaluating XQuery engines. One reason could be that the benchmarks are not easy to use due to technical problems. We checked this by running the benchmarks on four open source XQuery engines: Galax, SaxonB, Qizx/Open and MonetDB/XQuery. Indeed, we discovered several issues with the benchmark queries. First, the queries of X007 and XMach-1 are written in an outdated syntax and could not be run. Next, some queries of XBench and MBench contain mistakes and also could not be used. Also, there is a minor issue in the way the benchmarks queries specify the input documents, and finally, some benchmarks are not designed to run on engines that implement static type checking. The only benchmark that we ran without any problem is XMark. This could explain why XMark is the most often used benchmark.

We corrected the queries in order to run them on the four engines. In Section 6, we will present the outcomes of these experiments. In this section, we describe the problems that we found with the benchmark queries and how we corrected them.

3.1 Outdated syntax and errors

The benchmarks consist of a set of documents and a set of queries. The queries are given in a formal language (XQuery or variants of it) together with natural language descriptions of the expected answers. All queries are designed to return non-empty sequences of items. If during the evaluation of a query on a (set of) document(s) an error is raised, then the error must be due to a bug or limitation of the implementation.

A minimal requirement for the benchmarks is that the queries are *correct*, which means that the formal representation of a query does not contain errors and that the formal XQuery semantics [32] of the query corresponds to the natural language description. There are two kinds of incorrect queries. The first kind are queries that *should* raise XQuery errors because of non-compliance to the standard, including parsing errors. Remember that the queries are not designed to raise XQuery errors. We refer to such queries as *error-raising queries*. The second kind are queries that return a (possibly empty) sequence of items that does not correspond to the natural language description of the query answer. We call such queries *semantically incorrect*.

There are three different types of XQuery errors: static errors, type errors and dynamic errors [32]. We classify the error-raising queries by the type of error they produce. *Static errors* include parsing errors. *Type errors* occur when an operator is applied to operands of wrong types. There are two kinds of type errors: static type

errors and dynamic type errors. *Static type errors* are those that are detected during query parsing by static type checking. *Static type checking* is an optional feature and not all the engines implement it. *Dynamic type errors* are those that are detected during query execution when static type checking is not used. Any dynamic type error is also a static type error, while the opposite does not hold because of automatic type casting. Finally, *dynamic errors* occur when an evaluation operation cannot be completed, e.g., division by zero.

Since static type checking is an optional feature and it was not considered when designing the benchmarks, it is fair not to consider static type errors that are not also dynamic type errors as mistakes of the benchmarks. We will discuss these errors in Section 3.3.

Checking the correctness of a given query on a given (set of) document(s) is in general a non-computable problem (note that XQuery is Turing complete). Moreover there is no XQuery reference implementation to assist us in checking the correctness “by hand”. Nevertheless we can try to detect the incorrect queries by running the benchmarks on several XQuery implementations. We might not detect all the incorrect queries, and we run the risk of confusing implementation dependent errors with XQuery errors, but this is the best we can do.

The parsing errors (included in static errors) were detected by using the XQuery parser available at the W3C XQuery Grammar Test Page (<http://www.w3.org/2005/qt-applets/xqueryApplet.html>). The rest of the errors were detected by running the benchmarks queries on the **smallest** (set of) document(s) of the corresponding benchmarks on four XQuery engines: Galax, SaxonB, Qizx/Open and MonetDB/XQuery. Note that MonetDB/XQuery implements static type checking. Thus we ignore the type errors produced by MonetDB/XQuery while checking correctness. We detect semantically incorrect queries by comparing the result of a query obtained on the **smallest** benchmark document with the natural language description of that query. Our methodology is based on the assumption that the majority of XQuery implementations (conforming to XQuery standard) cope with the evaluation of all benchmark queries on the smallest (set of) documents of the benchmarks.

In Table 1, we present the number of incorrect queries that we found. The results are grouped per benchmark and per type of error they raise. Some queries contain multiple mistakes that should raise errors of different types. We count only one error per query, namely the first one in the following order: static error, type error, dynamic error. The incorrect queries that do not raise XQuery errors are semantically incorrect.

Out of a total of 163 benchmark queries, 48 are incorrect. XMach-1 and X007 are old benchmarks and their queries were written in older versions of XQuery. These queries raised static errors. Expressing the queries in an outdated formalism

Benchmark	static error	dyn. type error	semantically incorrect	incorrect/ total	
XMach-1	8	0	0	8/8	(100%)
X007	22	0	0	22/22	(100%)
XMark	0	0	0	0/20	(0%)
Michigan	5	0	2	7/46	(15%)
XBench TC/SD	0	1	0	1/17	(6%)
XBench DC/SD	0	1	0	1/16	(6%)
XBench TC/MD	0	3	0	3/19	(16%)
XBench DC/MD	0	6	0	6/15	(40%)
total	35	11	2	48/163	(29%)

Table 1

Number of incorrect queries grouped per benchmark and type of error

is not an error of the benchmarks; it rather indicates that they are not properly maintained. Nevertheless, the queries of XMach-1 and X007 are unusable. XBench and MBench contain queries that raise static errors, dynamic type errors and/or that are semantically incorrect. We did not find any dynamic errors in any of the benchmarks. On top of the statistics presented in Table 1, there are 14 queries that raised type errors on MonetDB/XQuery. We will discuss this errors and how we correct them in Section 3.3. XMark is the only benchmark without incorrect queries (possibly the reason why XMark is the most used benchmark).

To summarize, 29% of the total number of queries were unusable for testing Galax, SaxonB, and Qizx/Open due to diverse errors. If we consider also MonetDB/XQuery (which implements static type checking), then even more queries could not be used to test at least one of the four engines.

3.2 Correcting the queries

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

- (1) *avoid changing the semantics of the query,*
- (2) *keep the changes to the syntactical constructs in the queries to a minimum* (an XQuery query can be written in many different ways and the syntactic constructs used might influence the query performance (cf. [10] and Section 5 below)), and
- (3) *avoid using features that are not widely supported by the current XQuery engines* (for example, the `collection` feature). This guideline is meant to ensure that the benchmarks can be run on as many of the current implementa-

tions as possible.

For checking the correctness of our changes we rely on the parser available at the W3C XQuery Grammar Test Page and on SaxonB. We picked SaxonB as our reference implementation because it has a 100% score on the XML Query Test Suite (XQTS) [30]. Though XQTS is not officially meant to test for an engine's compliance to the XQuery standard, it is the best compliance test available. It consists of 14637 test cases covering the whole functionality of the language. An engine gets a 100% score if all the test cases run successfully on that engine and produce results conforming to the reference results provided in the test cases.

All the corrected queries run without raising any errors on SaxonB. On other engines errors are still raised, but they are due to engine implementation problems (see Section 6). Below we discuss the changes we made to the benchmark queries. The resulting syntactically correct benchmarks can be found on the web⁴, together with a detailed description of our changes.

3.2.1 Static errors

XMach-1, X007, and MBench contain queries that raise static errors. These errors are due to: (1) non-compliance to the current XQuery specifications, or (2) typographical errors. The XMach-1 and MBench queries are written in an older version of XQuery. They contain incorrect function definitions and incorrect FLWOR expressions and use built-in functions that were renamed or do not exist anymore. The X007 queries are written in Kweelt [27] – an enriched and implemented variant of Quilt [18]. Quilt is an XML query language that predates, and is the basis of, XQuery.

Correcting these errors is straightforward. Below we show an example of a query written in an old syntax. Consider query Q14 of X007:

```
FUNCTION year() { "2002" }
FOR $c IN document("small31.xml")
    /ComplexAssembly/ComplexAssembly
    /ComplexAssembly/ComplexAssembly
    /BaseAssembly/CompositePart
Where $c/@buildDate .>= . (year()-1)
RETURN
    <result>
        $c
    </result>
orderby (buildDate DESCENDING)
```

⁴ <http://ilps.science.uva.nl/Resources/MemBeR/other-benchmarks/queries.html>

We rewrote it to XQuery as follows:

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

for $c in doc("small31.xml")
      /ComplexAssembly/ComplexAssembly
      /ComplexAssembly/ComplexAssembly
      /BaseAssembly/CompositePart
where $c/@buildDate >= my:year()-1
order by $c/@buildDate descending
return
  <result>
    {$c}
  </result>
```

3.2.2 *Dynamic type errors*

X007, XBench and MBench contain type errors generated by: (1) applying the child step to items of atomic type, or (2) value comparisons between operands with incomparable types. These seem to be programming mistakes.

As an example of the first kind of 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 `$a` is of type `xdt:anyAtomicType`. The location step `text()` in the path expression `$a/text()` 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.

As an example of the second kind of error, consider 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 the right operand) are incomparable types a type error is raised. 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 take the latter option.

3.2.3 *Semantically incorrect queries*

We found two semantically incorrect queries, QS6 and QA2 of MBench. There might be more semantically incorrect queries that we did not detect. QS6 produced an empty sequence instead of the expected result due to a typo. QA2 contains two different programming mistakes that lead to incorrect results. We discuss this query in details below.

The natural language description of QA2 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.” [26]

The corresponding XQuery query is:

```

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

```

First of all, note that the function `my:one_level()` falls into an infinite recursion when it gets as input an empty sequence. Now, 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 avgaSixtyFour="{
  avg(for $a in $e return $a/@aSixtyFour)
}"
  aLevel="{ $e[1]/@aLevel}">
{my:one_level($e/eNest)}
</average>
```

The second error is a mismatch between structure of the resulting elements and the description of the result. When 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 Michigan benchmark authors explicitly say that the natural language description is the normative query definition. We thus picked the first option. The corrected query is below.

```
declare namespace my='my-functions';
declare function my:one_level($e as element(*)
{
if(empty($e)) then ()
else (
  <average avgaSixtyFour="{
    avg(for $a in $e return $a/@aSixtyFour)
  }"
    aLevel="{ $e[1]/@aLevel}" />,
  my:one_level($e/eNest)
)
};
<dummy>
{my:one_level(doc()/eNest/eNest)}
</dummy>
```

3.3 Other issues

There are two more issues that make the benchmarks difficult to use. One is that the benchmarks specify the input for their queries in different ways and not always

formally correctly. Another issue has to do with static type checking. The benchmark queries were not designed with this feature in mind and many queries raise static type errors when static type checking is used. We address these issues in this section and describe how we resolve them.

3.3.1 Specifying the input of a query

The benchmarks have different ways of indicating the input data. X007 and XMark queries use the `fn:doc()` function with a document URI (usually an absolute file name) as argument. MBench queries invoke the `fn:collection()` function on collection name "mbench", even though they are designed to query one document. XMach-1 queries do not contain input information and all the XPath paths are absolute. Finally, the XBench benchmark refers to the input by using a new function `input()` that is not formally defined. We changed the benchmarks so that all queries specify their input in the same way.

X007, XMark, MBench and XBench TC/SD and DC/SD are *single-document scenario* benchmarks, which means that their queries are evaluated against one document at a time. In a single-document scenario the input document should be specified, according to XQuery standard, by using the `fn:doc()` function. XMach-1 and XBench TC/MD and DC/MD are *multi-document scenario* benchmarks, i.e., their queries are evaluated against an (unbounded) collection of documents at once without explicitly invoking each document in the query via the `fn:doc()` function. XQuery has a special built-in function `fn:collection()` to deal with this scenario.

We changed the queries of X007, XMark, MBench and XBench (TC/SD and DC/SD) to access their input data by invoking the `fn:doc()` function. The document URI is left out to be filled in at query execution. 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.

For the queries of XMach-1 and XBench TC/MD and DC/MD we should use the `fn:collection()` function. Unfortunately, this function is not yet supported by all the engines. In order to run this scenario in a uniform way on all the current engines, we create an XML document `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>
```

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

Table 2
Language analysis of the benchmarks

We then 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 $collection :=
  for $docURI in doc("collection.xml")
    //doc/text()
  return doc($docURI)
for $a in $collection//tagname
return $a
```

3.3.2 Static type errors

Some engines, e.g., MonetDB/XQuery, implement static type checking feature of XQuery. This feature requires implementations to detect and report static type errors during the static analysis phase of the query processing model [31]. During static type checking the engine tries to assign a static type to the query and it raises a type error if it fails. In order to run the benchmarks on the engines that implement static type checking, we ensure that the benchmark queries do not raise static type errors.

All the benchmarks except XMark contain queries that raise static type errors on MonetDB/XQuery. All these errors were caused by applying operators and functions on sequences that could have multiple items while only a singleton or empty sequence is allowed. For example, Q6 of XBench TC/SD,

```
for $word in doc()/dictionary/e
where some $item in $word/ss/s/qp/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 passes the static type checker of MonetDB/XQuery.

4 Benchmark query analysis

Our main goal in this section is to see how representative the benchmark queries are of XQuery as a complete language. As a first step in this direction, we investigate whether the queries could equivalently be expressed in (fragments of) XPath. We end this section with a practical exercise: we look at each benchmark as a whole and determine whether all queries are needed in the benchmark. We take a utilitarian perspective: if two queries yield the same benchmark results (in our case, execution times) on a reasonable number of engines, then one of them is superfluous.

4.1 Functional coverage

XQuery queries can retrieve parts of the input document(s) and can create new XML content on the basis of them. We focus on the first of these two tasks. We investigate how much of the expressive power of XQuery is used by the benchmark queries for retrieving parts of the input. This is the main functionality of a query language and we have at hand a methodology for investigating it.

We classify the benchmark queries in terms of the XPath fragments in which they can be expressed. In this way we can see how much expressive power each query uses. XPath is a well known and popular language meant to retrieve parts of input documents. It is also a fragment of XQuery. This makes it a good point of reference for our investigation. We consider four flavors of XPath:

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

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

XPath 2.0 This language is less expressive than XQuery; for example, it does not contain sorting functionalities and user-defined functions. XPath 2.0 and XQuery share the same built-in functions.

Navigational XPath 2.0 This fragment excludes the use of position information, aggregation and built-in functions. Value comparisons are allowed. Navigational XPath 2.0 is used for navigating in the XML tree and testing value comparisons.

XPath 1.0 This language is less expressive than XPath 2.0, for example, it does not contain variables or iterations. It also contains fewer built-in functions.

Core XPath This fragment is the navigational fragment of XPath 1.0, as defined in [21]. It excludes the use of position information, built-in functions and comparison operators. Core XPath is used only for navigating in the tree.

The distribution of the benchmark queries in these fragments serves as an indication of how much of the expressive power of XQuery is used, and by how many queries. Analyzing the fragments that are not covered, we can determine which XQuery functionalities are not used by the queries.

Many benchmark queries create new XML content based on the information re-

trieved from the input documents. For example, some queries use element construction to group the retrieved results. Most of the queries use element construction only to change the tag names of the retrieved elements. Since XPath queries cannot create new XML and since we investigate just the “retrieval” part of a query, we ignore this operation when analyzing the benchmark queries. We will give more details on this below. The remainder of the query we rewrite, if possible, into one of the XPath fragments described above. We try to express each query first in Core XPath. If we fail we try to express it in XPath 1.0. If we fail again we take Navigational XPath 2.0 and in the end XPath 2.0. The queries that cannot be expressed in XPath 2.0 are genuine XQuery queries.

Table 2 contains our results. Out of 163 queries, 47 (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 those use sorting and the other 3 use recursive functions.

We explain below the procedure we followed for rewriting XQuery queries into XPath queries. Often, XQuery queries use element construction to produce the output, while XPath just copies the retrieved parts of the input document(s) to the output. So how can we rewrite an XQuery query into an XPath query? We remove the element construction from the generation of the output and just output the sequence of items retrieved by the query. Note that the query semantics changes in such case. The following example illustrates the process. Consider query QR2 from MBench:

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

For each eNest element that satisfies the condition @aSixtyFour=2 and that is given in the document order, this query creates a new eNest element containing a new element child for each child given in the document order of the original eNest element. If we strip away the element construction the query becomes:

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

This query retrieves attributes from the source document and outputs them in the

following order:

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

Where the e_i are given in document order, and their children as well. Thus the order in which the information is presented is maintained, but the structure of the output elements is changed. The difference between two queries is that the XPath query outputs a sequence of atomic values retrieved from the input document, while the original query uses these atomic values to construct new XML elements. Thus the structure and the type of the items in the result sequence changes.

Note that this query belongs to Navigational XPath 2.0 and cannot be rewritten in XPath 1.0.

4.2 Behavior analysis

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
XMark	20	12	9,2,1,1,1,1,1,1,1
MBench	46	17	28,2,2,1,1,1,1,1,1,1,1,1,1,1,1
XBenchTCSD	17	11	5,2,2,1,1,1,1,1,1,1
XBenchDCSD	16	9	6, 3,1,1,1,1,1,1,1
XBenchTCMD	19	4	16,1,1,1
XBenchDCMD	15	7	1,1,1,1,1,1,1,1,1,1,1,1,1,1,1

Table 3
Discriminative cores of the studied benchmarks

Benchmarks usually consist of a mix of simple and more challenging queries. With time the engines get more efficient and the benchmarks grow old. Some queries might become redundant as part of a benchmark. For example, the queries testing for downwards navigation in the XML tree are simple for the current implementations and all the engines have the same performance on them. Running and interpreting a benchmark is costly; thus we want the set of queries of a benchmark to be as concise as possible. That is why a benchmark should be modified with time and the set of queries cleaned of queries that (1) do not discriminate between engines; (2) perform very well on all engines in “all” situations; (3) are redundant (i.e., another query in the same benchmark measures the same thing).

The notion of minimal or cost effective benchmark is highly dependent on its design and goals. In this section we give an example of an ad-hoc measure that can be applied when the benchmark goal is unknown, or that can be applied as a pilot investigation of the benchmark queries. We view running a set of queries on one document on different engines as one experiment and ask what differences in engines' behavior does an individual query show? Consider Figure 1 which describes the total execution times of running Xbench DC/SD on a 10MB and a 100 MB document on four different engines. (Note the log scale on the y-axis for the 100MB document). The execution times of queries 12 and 14 is almost the same on all four engines, on both documents. Thus after running query 12 we gain no more additional insight from running query 14. So we can say that these queries are performance similar for these engines. Query 20 is different because of the missing value for one of the engines. 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.

We define a measure of performance similarity of two queries as follows: queries q_i and q_j are *performance similar* 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. Formally,

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

We do not have any empirical evidence to support the limit of 15% used in this formula; it was chosen as a “rule of thumb”. Also notice that this measure is highly dependent on the set of engines that the queries are executed on. For example, queries 12, 14, 19, and 20 in Figure 1 are performance similar if we consider only SaxonB and MonetDB/XQuery as reference engines. If we include Qizx/Open in calculation, only 12, 14 and 19 are performance similar, due to the missing value for Qizx/Open on query 20.

Based on the performance similarity measure obtained from running Xbench DC/SD on the four engines and on the two documents displayed in Figure 1, we obtained the following groups of performance similar queries:

$$\begin{array}{l}
Q1, Q8 \quad Q2, Q5, Q7, \quad Q3 \quad Q4 \quad Q6 \quad Q10 \quad Q11 \quad Q17 \quad Q20 \\
Q9 \quad Q12, Q14 \\
Q19
\end{array}$$

where:

```

for $a in A, $b in B
where $a/@att1 = $b/@att2
return
($a/@att1, $b/@att2)

```

if:

```

for $a in A, $b in B
return
if( $a/@att1 = $b/@att2 )
then ($a/@att1, $b/@att2)
else ()

```

pred:

```

for $a in A,
  $b in B[$a/@att1 = ./@att2]
return
($a/@att1, $b/@att2)

```

filter:

```

for $a in A, $b in B
return
($a/@att1, $b/@att2)
[$a/@att1 = $b/@att2]

```

A, B: path expressions; att1, att2: attribute names

Fig. 3. Four logically equivalent ways of expressing an equi-join.

The groupings are determined as follows: each group contains one query (the “representative query” of the group) which is performance similar to all other queries in the group.

In the same way we determined a *discriminative query core* of the studied benchmarks. Table 3 presents our results. 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 queries that have the same performance on four different engines. The actual groups can be found at <http://ilps.science.uva.nl/Resources/MemBeR/other-benchmarks/groupings.txt>.

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

5 Micro-benchmarking

Out of the five benchmarks we consider, four benchmarks, XMach-1, XMark, X007, and XBench, are defined as application benchmarks and the Michigan benchmark (MBench) is defined as a micro-benchmark. *Application benchmarks* test the overall performance of an engine in a real-life user scenario, while *micro-benchmarks* focus on testing particular aspects of query evaluation [10]. For example, a micro-benchmark can test the performance of basic query operations, like selection, sorting, joins, etc. Other targets for micro-benchmarking can be the efficiency of a stor-

Varying query parameters	Expected results
QJ1 $\Rightarrow$ QJ2 query selectivity: 1.6% $\Rightarrow$ 6.3%	evaluation time grows
QJ3 $\Rightarrow$ QJ4 query selectivity: 0.02% $\Rightarrow$ 0.4%	evaluation time grows
QJ1,QJ2 $\Rightarrow$ QJ3,QJ4 average query selectivity: 3.95% $\Rightarrow$ 0.21% join type: value-based $\Rightarrow$ id/idref syntactic form: <i>where</i> form $\Rightarrow$ <i>if</i> form	average evaluation time decreases

Table 4

Varying the query parameters of the four join queries of the Michigan benchmark and the expected results

age scheme, of an index, or of a query optimization technique like query rewriting. Most importantly, experimenting with micro-benchmarks should lead to precise conclusions about the engine’s performance on the micro-benchmark’s target [10].

In this section we investigate the micro-benchmarking properties of the Michigan benchmark query set. More precisely, we analyze the set of four queries which test the performance of *value-based joins*. Our main goal is to check whether these queries allow for precise conclusions regarding the engine’s performance on value-based joins, i.e. whether the target of the micro-benchmark is reached.

We first run these join queries on Qizx/Open. The recorded evaluation times show that: (1) it is difficult to interpret the results, because the queries vary several parameters at once; and (2) from this query set we cannot draw conclusions about the performance of an engine on value-based joins. We then extend the join query set to a micro-benchmark testing for join recognition techniques. We run this new micro-benchmark on four XQuery engines, Galax, SaxonB, Qizx/Open, and MonetDB/XQuery, and analyze the results.

Summary and outline of this section. The join queries of the Michigan benchmark are insufficient to draw conclusions about the performance of an engine on value-based joins. But the set of join queries can be extended to overcome this problem. We extended the query set to allow for testing a more focused target, the performance of join recognition techniques, and present the results for this new micro-benchmark. Section 5.1 contains the description the join query set of the Michigan benchmark and the experiment with Qizx/Open. In Section 5.2 we describe the new micro-benchmark and in Section 5.3 we present the results of running it on four XQuery engines.

5.1 The join queries of the Michigan benchmark

The Michigan benchmark consists of 46 XQuery queries and 4 documents of increasing size (approximately 45MB – 40GB). The documents in the benchmark have parameters like number of elements, fan-out, and document depth. Most (99%) elements are of the same type and are called `eNest`. Each `eNest` element has 6 numeric attributes with precise value distributions. For example, the attribute `aUnique2` of an `eNest` element contains a unique integer generated randomly; the attribute `aSixtyFour` of an `eNest` element contains an integer equal to the value of its `aUnique2` attribute modulo 64. The remainder (1%) of the elements are called `eOccasional` and contain only one attribute of type `IDREF`.

The queries are designed to test the performance of basic query operations. There are 5 categories of queries, testing for *simple selection*, *structural selection*, *serialization*, *value-based joins*, and *aggregates*.⁵ Each query has two variants, one selecting a small number of elements from the input document and returning these in the output, and the other selecting a large number of elements. *Query selectivity* is the percentage of elements of the queried document retrieved (selected) by the query. The selectivity of a query is controlled by filtering the `eNest` elements with a particular attribute value. For example, the query `//eNest[@aSixtyFour=0]` returns approximately 1/64th (1.6%) of all `eNest` elements. By varying the selectivity of a query one can test the influence of the result size on the evaluation times. In summary, the Michigan benchmark is a family of 5 micro-benchmarks each focusing on a basic query operation and allowing for document size scalability and query result size scalability tests.

To exemplify the potential and the limitations of the Michigan benchmark, we consider the category of join queries as a case study. They are designed to test how an engine deals with joins on attribute values. The performance of engines is measured in two dimensions: *join type* and *query selectivity*. There are two types of joins: joins on *simple attributes* and *id/idref* joins. The distinction was made in order to test possible performance advantages of the *id/idref* joins in the presence of an *id*-based index. Between the queries of the same join type the query selectivity is varied, in order to test for the influence of the query result size on the join evaluation algorithms. The four join queries of the Michigan benchmark, QJ1–QJ4, are created by varying these two parameters.

Queries QJ1 and QJ2 are joins on simple attributes; QJ3 and QJ4 are *id/idref* joins. QJ2 returns roughly 4 times more elements than QJ1, and QJ4 returns around 20 times more elements than QJ3. The actual queries are below.

The query QJ1 is:

⁵ The benchmark considers an extra query category that falls outside the scope of this paper, namely update queries.

```

for $e1 in doc()//eNest[@aSixtyFour=2],
    $e2 in doc()//eNest[@aSixtyFour=2]
where $e2/@aUnique1=$e1/@aUnique1
return
    <eNest1 aUnique1="{ $e1/@aUnique1} "
        aSixtyFour="{ $e1/@aSixtyFour} "
        aLevel="{ $e1/@aLevel} ">

    <eNest2 aUnique1="{ $e2/@aUnique1} "
        aSixtyFour="{ $e2/@aSixtyFour} "
        aLevel="{ $e2/@aLevel} "/>
</eNest1>

```

QJ2 is obtained from QJ1 by replacing all the occurrences of the attribute name `aSixtyFour` with `aSixteen`. Thus we expect that QJ2 returns 4 (=64/16) times more elements. The selectivities of QJ1 and QJ2 are approximately 1.6% and 6.3%, respectively.

The query QJ3 is:

```

for $e1 in doc()//eOccasional,
    $e2 in doc()//eNest[@aSixtyFour=3]
return
if ($e2/@aUnique1=$e1/@aRef) then
    <eOccasional aRef="{ $e1/@aRef} ">
        <eNest aUnique1="{ $e2/@aUnique1} "
            aSixtyFour="{ $e2/@aSixtyFour} "/>
    </eOccasional>
else()

```

QJ4 is obtained from QJ3 by replacing all the occurrences of the attribute name `aSixtyFour` with `aFour`. The selectivities of QJ3 and QJ4 are approximately 0.4% and 0.02%.⁶

Remark. Besides the two parameters described above, the queries vary in another parameter, namely the *syntactic form* used to express the joins. QJ1-QJ2 use the `where` clause to express the join, while queries QJ3-QJ4 use the `if then else` construct. Abstracting away from these specific instances the syntactic patterns of the four queries are as shown in the top row of Figure 3. Clearly, these two patterns are equivalent. Moreover, both variants have the same normal form in XQuery Core [32], which is a complete fragment of XQuery that is used to specify the

⁶ Note that even though the selectivity of the subexpression `//eNest[@aFour=3]` is 16 times larger than the selectivity of the subexpression `//eNest[@aSixtyFour=3]`, the selectivity of QJ4 is 20 times larger than the selectivity of QJ3. The difference is due to the influence of the `eOccasional` elements on the join outcome. For more information about the `eOccasional` elements and their attribute values see [26].

formal semantics. Unfortunately, the benchmark authors do not explain why this parameter is varied and how it influences the target of the micro-benchmark: the performance of value based joins.

In [26] the join queries are evaluated on a document of fixed size and the results are interpreted on the four evaluation times measurements. When analyzing these evaluation times, one should look at the effect of the query selectivity on the performance for each join type. If a simple, unoptimized nested loop join algorithm is implemented to evaluate the joins, the query complexity is $O(n^2)$ and the selectivity factor has large impact on the performance times. On the other hand, optimized algorithms should scale better with respect to query selectivity. One expects that the id/idref joins scale up better than the simple joins, when, for example, an id-based index is used. And finally, one would expect that the average evaluation time of the id/idref joins is smaller than the average evaluation time of the simple joins, due to the optimization opportunities of the id/idref joins and also due to the fact that the query selectivity of the former queries is smaller than the query selectivity of the later queries. Unfortunately, the comparison of the simple joins and the id/idref joins cannot be made precise because the syntactic form of the query is varied at the same time and it is not clear how this parameter influences the evaluation times. In Table 4, we list the parameters that vary between the four queries, their values and the expected influence on the results.

We now show that the seemingly harmless and unexplained variation in syntactic form leads to results which are hard to interpret.

Experiment

We ran the four queries on the smallest (46MB, 728K nodes) Michigan benchmark document using Qizx/Open. For more details about the experimental set-up see Section 6. The evaluation times in seconds are presented in Table 5. As expected, the evaluation times for QJ2 and QJ4 are larger than those for QJ1 and QJ3, respectively. But the average evaluation time for QJ3-QJ4 is 2 orders of magnitude larger than the average time for QJ1-QJ2, while we expected the evaluation time to decrease. Does this indicate an abnormality with the Qizx/Open implementation of id/idref joins? Or is the difference in the evaluation times maybe due to the variance in syntactic form?

Query (selectivity)	Evaluation time (sec)
QJ1 (1.6%)	3.6
QJ2 (6.3%)	3.8
QJ3 (.02%)	338.8
QJ4 (.4%)	396.1
avg(QJ1,QJ2)	3.7
avg(QJ3,QJ4)	367.45

Table 5. Qizx/Open on the four join queries of MBench

The latter hypothesis sounded more likely. Thus we extended the query set with the `where` and `if` variants for all four queries. The execution times presented in Table 4 show that our supposition was right. Note that if we fix the syntactic form (i.e. consider one column of Table 4), then the results correspond to our initial expectations: the evaluation times increase within a join type when the query selectivity increases, and the average evaluation time of `id/idref` joins is smaller than the average evaluation time of value-based joins. But the processing times for the `if` variant are much larger than the performance times for the `where` variant. The

algorithm that Qizx/Open applies for the joins expressed in the `where` form is efficient, it seems to scale sub-linear with respect to the query selectivity, but it shows no difference between the two types of joins. The algorithm applied to the joins expressed in the `if` form is less efficient: it seems to scale super-linear with respect to query selectivity for the simple joins, but scales better for the `id/idref` joins.

By separating the influence of the query’s syntactic form from the influence of the other parameters, we could interpret the results and learn more about the engine’s join evaluation strategies. Moreover, our extended experiment indicated a problem with the join detection mechanism of the engine. Still, this query set is rather simple and we cannot say much about the performance of the join detection mechanism in Qizx/Open. For example, we cannot say whether the joins in the `where` form will always perform better over the ones in the `if` form.

Since in XQuery joins can be expressed syntactically in many different ways, the join evaluation problem is two fold: first a join has to be recognized and then the efficient algorithm can be applied. In order to test for join performance one needs to test first the join recognition techniques and then, assuring that one algorithm can be isolated for a particular case, to test for the performance of that join evaluation algorithm. Thus these two subproblems should be tested separately to avoid misinterpretation of the results. Moreover, joins are complex operations and there are more parameters that might influence the performance of a join evaluation algorithm, for example the number of join conditions.

We took this challenge and created a micro-benchmark to thoroughly test the join detection mechanisms for simple joins. As future work, we would like to extend this micro-benchmark to also test the performance of join evaluation algorithms. We ran our new micro-benchmark on Qizx/Open (among other engines) and the results showed that the engine has implemented a weak detection algorithm, probably

Query (selectivity)	where variant	if variant
QJ1 (1.6%)	3.6	330.4
QJ2 (6.3%)	3.8	1405.6
QJ3 (.02%)	3.3	338.8
QJ4 (.4%)	3.5	396.1
avg(QJ1,QJ2)	3.7	868
avg(QJ3,QJ4)	3.4	367.45

Table 6. Qizx/Open on the MBench join query set extended to cover two syntactic variants (evaluation times in seconds).

parameter / query:	Aa	Ab	Ab&	Abv	Ab&v	Ab&&	Ba	Ba&	Ba&v	Bref	Bref&
number of join conditions	1	1	2	2	3	3	1	2	3	1	2
Boolean connective(s)	–	–	and	or	and, or	and, and	–	and	and, or	–	and
join type	self	self	self	self	self	self	simple	simple	simple	id/idref	id/idref
equivalence class	$\widetilde{Aa}$	$\widetilde{Ab}$	$\widetilde{Ab}$	$\widetilde{Ab}$	$\widetilde{Ab}$	$\widetilde{Ab}$	$\widetilde{Ba}$	$\widetilde{Ba\&}$	$\widetilde{Ba}$	$\widetilde{Bref}$	$\widetilde{Bref}$

Fig. 4. Join recognition micro-benchmark queries and their values for the varying parameters

based on syntactic patterns, and that not all the joins expressed in the `where` form perform well (see the results on Qizx/Open in Figure 6). We describe the micro-benchmark in details in the next subsection. In Section 5.3 we apply the benchmark on four engines and analyze the results.

5.2 A micro-benchmark testing join recognition

In our experiment from the previous section we spotted a problem with the join recognition mechanism of Qizx/Open, but the join queries of the Michigan benchmark were not enough for us to draw any conclusion about the join recognition mechanism of this engine. In this section we present a micro-benchmark inspired by the join queries of the Michigan benchmark. Our goal is to show how these join queries can be transformed into a micro-benchmark testing for the join recognition mechanism of an engine.

Target

Joins are expensive operations. In the settings of relational databases and SQL, there was a considerable amount of effort invested into implementing joins efficiently [8]. In the settings of XML and XQuery the same problem stands. Moreover, since in XQuery joins can be expressed syntactically in many different ways, the problem is two fold: first a join has to be recognized and then the efficient algorithm can be applied. Fortunately, the join evaluation algorithms employed by the relational databases can be transferred to the XQuery settings and many XQuery engines have implemented efficient algorithms for joins (e.g. Qizx/Open, MonetDB/XQuery, etc.). Still, recognizing a join in XQuery is not an easy task due to the complexity of the language.

We present a micro-benchmark that targets the join recognition mechanism of XQuery engines. We assume that the tested engine implements an efficient algorithm for evaluating value-based joins and also that it has a mechanism for determining when a query involves a join and then applying that algorithm for its evaluation.

Measure

The idea behind the benchmark is to compare the performance of different syntactic variants of the same join. The difference between the evaluation time of each variant serves as a measure for determining whether the engine incorporates a robust technique for join detection.

Data

The micro-benchmark data set is based on queries similar to QJ1-QJ4 of the Michigan benchmark, using the same benchmark documents.

Query parameters

The set of queries is built by varying several parameters that might challenge the join recognition mechanism. We only consider value-based equi-joins, i.e., the join condition is an equality of attribute values. All the attributes used in the queries are of numeric type⁷ (the documents are provided with an XML Schema). Whenever possible, we design the queries to have the same intermediate and final results on the benchmark documents. In this way we can compare the performance of equivalently behaving (though differently written) queries. The parameters that are varied are:

- **syntactic patterns** – in addition to the syntactic variants `where` and `if`, we consider two more equivalent syntactic variants, `pred` and `filter` (Figure 3);
- **number of join conditions** – we consider queries with up to 3 join conditions;
- **Boolean connectives** – we use *conjunctions* and *disjunctions* to combine multiple join conditions;
- **join types** – we consider three types of joins: *simple*, *id/idref chasing* and *self-join*;
- **document-level equivalence classes** – besides the fact that each query is represented in four equivalent syntactic variants, the query set is divided into several subsets containing queries that are equivalent (have the same answer) with respect to the MBench documents.

We now describe each parameter.

Syntactic patterns. Each query is expressed in four different syntactic variants: `where`, `if`, `pred`, and `filter` shown in Figure 3, where `A` and `B` are path expressions, while `att1` and `att2` are attribute names. The most common way of expressing joins in XQuery is by using the `where` clause in a FLWOR expression.

⁷ For the future, it would be interesting to test different conditions like `<` or the text function `contains()` expressed on attributes of different types.

All five benchmarks have joins expressed in this way. The origin of the `where` clause in XQuery lies in the `where` clause in SQL. The `where` and the `if` pattern have the same normal form in XQuery Core [32]. This is a complete fragment of XQuery that is used to specify the formal semantics. In the pattern `pred`, the join is expressed in a predicate. Some benchmarks use this pattern for expressing their joins (XMach-1, XBench). In the `filter` pattern, the same predicate condition appears in the return clause as a filter to the sequence construction. Though this variant cannot be found in any of the five benchmarks, it is a simple pattern likely to be used.

Join conditions and types. Further, the queries can have one, two or three different join conditions combined with conjunction and/or disjunction between them, for example:

```
for $a in A, $b in B
where
($a/@att1 = $b/@att2 and
 $a/@att3 = $b/@att4)
return ($a, $b)
```

where `att3` and `att4` are attribute names.

The join types *simple* and *id/idref* are determined by the types of the attributes involved in the joins, *simple* (`xs:integer`) or *id/idref* (`xs:ID`, `xs:IDREF`). A join in which elements from a sequence are combined with other elements from the same sequence when there are matching values in the joined attributes, is called a *self-join*.

Equivalence classes. The query set is divided into five equivalence classes. Two queries belong to the same equivalence class if they have the same intermediate and final results on the MBench documents. By intermediate results we mean the results of the path expressions A and B in the query patterns described above. The attribute values of the MBench documents are strongly correlated (see Figure 9 for details) and this correlation causes the equivalence of the queries. Note that the queries are only equivalent for the MBench documents because of the particular way they are generated. The queries are not equivalent on all documents conforming to the MBench DTD.

Query set

Following the patterns described above, we arrived at the query set consisting of 11 join queries (44 syntactically different queries) described in Figure 5. The name of a query is an encoding of the query properties. If in the query patterns described above, the path expression B is the same as the path expression A, then the query is a self-join and the query name contains the capital letter A, otherwise it contains

the capital letter B. If any join condition in a query (any query can have one, two or three join conditions) is expressed between two attributes with the same name, then the query contains the small letter a, otherwise b. For example query Aa is:

```
for $a in A, $b in A
where $a/@att1 = $b/@att1
return ($a,$b)
```

Further, $\vee$ stands for a disjunct and $\&$ for a conjunct, thus the query $Ab\&\vee$ contains three join conditions connected by a conjunction and a disjunction, as shown below.

```
for $a in A, $b in A
where
($a/@att1 = $b/@att2 and
$a/@att3 = $b/@att4 or
$a/@att5 = $b/@att6)
return ($a,$b)
```

Queries whose name contains `ref` are id/idref chasing joins. Since the attribute names are necessarily different, the corresponding `b` letter is redundant. Each query belongs to an equivalence class. The classes are defined by one of their query members. For example, $Ab\&$ is equivalent to Ab , thus it falls into the $\tilde{A}b$ equivalence class. The `where` variant of the actual queries can be found in Figure 9. The whole set of queries can be found online at <http://ilps.science.uva.nl/Resources/MemBeR/mb-joins/output/outcome.html>.

Running scenario

The running scenario of this micro-benchmark is to fix a document size and for each query to evaluate all four syntactic variants.

Interpreting benchmark results

Recall that the results of the benchmark are interpreted on the assumption that the tested engine implements an efficient algorithm for evaluating value-based joins and also that it has a mechanism for determining when a query involves a join and it applies that algorithm. The robustness of the join recognition mechanism is then measured by varying the query parameters one at a time. The main dimension is the syntactic formulation of the joins. A robust engine should recognize the equivalence of the different syntactic variants and apply the same join evaluation algorithm on them. By varying the join type, the number of join conditions, and the boolean connectives between join conditions, we can determine whether different algorithms are applied for different variants and learn their behavior with respect to these parameters. The classes of equivalent queries further help us to isolate the

Fig. 5. Join recognition micro-benchmark results on 4 engines. The gray background indicates the equivalence class $\tilde{Ab}$.

influence of the above parameters.

A robust recognition technique should show in:

- A. similar performance times for as many syntactic variants of a query as possible;
- B. similar performance times for all the queries within an equivalence class expressed in a particular syntactic variant, meaning that varying the join condition and boolean connectives does not influence the recognition;
- C. similar performance times for all the queries of the same join type expressed in a particular syntactic variant, meaning that varying the join condition and the boolean connectives does not influence the recognition.

The next subsection presents the results of our experiments.

5.3 Join recognition micro-benchmarking results

We ran our join recognition micro-benchmark on four XQuery engines: SaxonB v8.8 [22], Qizx/Open v1.0 [11], Galax v0.5.0 [19], and MonetDB/XQuery v4.15.0/v0.12.0 [25]. For this experiment we picked the smallest documentnt size avail-

	A. comparing the syntactic variants	B. query performance within the equivalence class $\widetilde{Ab}$
Qizx/Open	<ul style="list-style-type: none"> - The <code>if</code>, <code>pred</code>, and <code>filter</code> forms perform the same, though the performance times are bad. - The <code>where</code> form performs 2 orders of magnitude better than the other forms, with the exception of queries containing disjunctions. Ranking: 1. <code>where</code> 2. <code>if</code> , <code>pred</code> , <code>filter</code>	<ul style="list-style-type: none"> - <code>where</code> form: the queries containing <i>disjunction</i> perform 2 orders of magnitude worse than the other queries. - <code>if</code>, <code>pred</code>, and <code>filter</code> forms: all queries perform similar.
SaxonB	<ul style="list-style-type: none"> - The <code>where</code> and <code>if</code> forms perform the same. - The <code>filter</code> form performs one order of magnitude better than the rest of the forms. Ranking: 1. <code>filter</code> 2. <code>where</code> , <code>if</code> 3. <code>pred</code>	<ul style="list-style-type: none"> - <code>where</code> and <code>if</code> forms: the queries containing <i>disjunction</i> perform 2 orders of magnitude better than the other queries. - <code>pred</code> and <code>filter</code> forms: all queries perform similar.
Galax	<ul style="list-style-type: none"> - All four forms perform different: the <code>where</code> form performs best; the <code>pred</code> form is 2 orders of magnitude slower than the <code>where</code> form. Ranking: 1. <code>where</code> 2. <code>if</code> 3. <code>filter</code> 4. <code>pred</code>	<ul style="list-style-type: none"> - All forms: all queries perform similar.
MonetDB/ XQuery	<ul style="list-style-type: none"> - The <code>where</code> and <code>if</code> forms perform the same. - The <code>pred</code> form is one order of magnitude slower than the rest. Ranking: 1. <code>where</code> , <code>if</code> 2. <code>filter</code> 3. <code>pred</code>	<ul style="list-style-type: none"> - <code>pred</code> form: the performance times increase with the <i>number of join conditions</i>. - <code>where</code>, <code>if</code>, and <code>filter</code> forms: all queries perform similar.

Table 7

Join recognition micro-benchmark results on four engines. Checking the robustness conditions A. and B. of the join detection.

able, namely 45MB. MonetDB/XQuery was not able to complete the evaluation on this document, so we generated a smaller one by cutting the XML hierarchy at the 11th level. The resulting document is of size 2MB. All the experiments were done

with XCheck [9] and all the experimental data and details are at <http://ilps.science.uva.nl/Resources/MemBeR/mb-joins/output/outcome.html>. The engines were run with their default settings.

Figure 6 contains the result plots per engine. Each plot presents the evaluation times in seconds of the respective engine obtained for the four different syntactic variants. On the X axis, the 11 join queries are placed. With a gray background we indicate the equivalence class $\widetilde{Ab}$. We comparing engines, note that the scale on the Y-axis varies from plot to plot, and recall that the MonetDB experiment was performed with a much smaller input document.

Above we described three conditions, A, B, and C, that a robust join detection mechanism should satisfy. In Table 6 we present the outcomes of checking condition A and B. In Column A, we check how many of the four syntactic forms perform the same. We also rank the forms based on the average performance times of all queries written in this form—the best performing on top. In column B contains results for equivalence class $\widetilde{Ab}$. The results for the other classes do not reveal more information. Note that the last condition, C, asks whether the queries of the same join type perform similarly. Since each join type is the union of one or several equivalence classes of queries, the results for B imply also the results for C, thus we do not discuss them.

The results indicate that all four engines have implemented smarter join evaluation techniques than just taking the cartesian product and filtering afterwards. Nevertheless, the engines do not manage to recognize the different syntactic forms as the same query. As a result, the execution times of different equivalent variants of the same query differ with up to two orders of magnitudes. The robust engines with respect to recognizing the equivalence of `where` and `if` variants are MonetDB/XQuery and SaxonB. However, for SaxonB, the `filter` variant is the best performing and the engine misses the equivalence between this variant and the rest. The worst performing join recognition technique is implemented in Qizx/Open. From the results we conclude that Qizx/Open recognizes the joins based on a weak syntactic pattern that does not cover the disjunctive joins. Below, we analyze the results for each engine presented in Table 6.

Qizx/Open. The results for Qizx/Open indicate that the engine’s join recognition is based on a syntactic pattern using the `where` clause. The three other forms are evaluated in the same way, we believe, by materializing the cartesian product as intermediate result. Moreover it seems that the `where` pattern misses the disjunctive joins, since these queries perform as bad as the queries written in the other syntactic forms.

SaxonB. We were surprised to observe the `filter` variant to be the winner in the case of SaxonB, since this is a less common way of writing joins in XQuery. The semantics of this variant suggests that each resulting pair of the cartesian product is

constructed and then the join is applied. We asked the developer’s opinion on this matter. The implementor of SaxonB, Michael Kay, confirmed that `filter` is the only variant of the join queries in which the subexpression `B` (see the query patterns in Figure 3) is being regularly pulled out of the nested for loop (so it’s evaluated once and the results held in memory). In the other cases this opportunity not being recognized, query plans for all the variants have the same complexity.

The `where` and `if` variants perform the same. This is caused by the fact that both variants have the same normal form in XQuery Core. SaxonB first rewrites a query into its normal form and then executes it. Observe that the disjunctive joins expressed in the `where` and `if` form are evaluated much faster than the other queries from the same equivalence class $\widetilde{Ab}$. For these queries the same evaluation strategy as for the `filter` queries is applied.

Galax. The results on Galax show that all four variants have different implementations. The best performing variant is `where` and the least performing is `pred`. Even though the Galax implementation pipeline indicates that all the queries are normalized to XQuery Core before evaluation, the differences between `where` and `if` indicate that this is not always the case.

MonetDB/XQuery. First we have to remark that MonetDB/XQuery (including the version we test v4.15.0/v0.12.0) is based on two different underlying implementations of XQuery language. The first implementation, we call it *milprint-summer*, has a good coverage of XQuery language but relies on ad-hoc query optimizations. The second implementation, we call it *algebra-based*, uses an extended relational algebra and relies on algebraic optimizations, though it does not cover the whole XQuery language. When executing the engine, one can specify which implementation should be used. The results that we present here use the *milprint-summer* implementation. The best performing variants are `where` and `if`, which perform the same. This is explained by the engine’s compilation pipeline: both syntactic variants are translated into the same XQuery Core normal form. The `filter` variant seems to be evaluated in a different manner, nevertheless retaining comparable performance. The worst performing variant is `pred`. Note that in the case of queries `Aa` and `Ref`, the performance time points overlap. We believe that the developers might have included simple syntactic patterns to recognize the simple joins in the `pred` and `filter` variants, and those patterns do not cover the conjunctive and disjunctive joins.

In conclusion, none of the tested engines was able to recognize all four join variants. Nevertheless, two of the engines, SaxonB and MonetDB/XQuery, manage to evaluate the `where` and `if` variants in the same way, we believe due to normalizing XQuery queries to XQuery Core. An advice for XQuery users, based on these results, is to use the `where` clause for expressing joins and stay away from the `pred` version.

6 Running the benchmarks

In the previous sections we analyzed the benchmarks themselves, in this section we show some examples of how they can be used to analyze query engines. We report on 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 7 and 8 contain the results of running the benchmarks on these two documents on the four engines. The individual queries in the benchmarks 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 7) 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 7 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.⁸ Table 8 lists the engine crash errors for Galax and MonetDB. For Galax, these were “materialization out of bounds” errors. For MonetDB/XQuery they differed: large intermediate results do not fit in the main-memory nor on the virtual memory; it cannot deal with a large 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 8
Number of syntax errors for each benchmark

Ranking the engines? It seemed a nice idea to organize a tournament between engines, like reported in [14]. They consider 16 engines and one benchmark: XMark. The data for comparison is gathered from different sources in the literature, and

⁸ <http://ilps.science.uva.nl/Resources/MemBeR/other-benchmarks/results.html>

	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 9

Engine crash errors given by Galax and MonetDB.

the different settings under which the tests were performed made the comparison hard. We wanted to perform a better controlled tournament, now using all available benchmarks. The data in Figures 7 and 8 show that evaluating the performance of an engine is not so simple as ranking them in a list. 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 8. 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 [14], 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 and of the engines, performance rankings of engines are not informative. On the other hand, performance comparison of engines can be very useful for developers. For instance, consider Qizx on MBench in Figure 7. It shows very nice almost constant behavior with two peaks on queries QJ3 and QJ4. Qizx developers can compare their times with those of Saxon and Galax which give much more stable behavior on the related queries QJ1–QJ4. They would easily see that a small investment in query optimization (rewrite an if clause into a where clause, as described in Section 4.2) has great performance benefits.

A more detailed performance analysis of these four engines and others, based on the benchmarks, is presented in [23].

7 Conclusions

We analyzed the XQuery benchmarks publicly available in 2006 by asking three questions: How are they used? What do they measure? What can we learn from using them? The main conclusion that we draw is that the benchmarks are useful

but not totally adequate for evaluating XQuery engines.

Concerning the actual usage of the benchmarks, our literature survey shows that, with the exception of XMark, the benchmarks are *not* used for evaluating XQuery processors. One reason might be that the benchmarks do not comply with the W3C XQuery standard. We found that 29% of the benchmark queries cannot be run on current XQuery engines due to diverse errors, including syntax errors. We fixed these errors and rewrote the queries in a uniform format for all the benchmarks. A second reason might be that many of the papers contain an in-depth analysis of a particular XPath/XQuery processing technique and the benchmarks are not suitable for this. In such cases, specialized micro-benchmarks are more appropriate [10].

Concerning what the benchmarks measure, our analysis of the benchmark queries showed that they test mostly XPath functionality. Note that we were concerned only with retrieving information from the input document(s), and we ignored the XML construction functionality of XQuery. Only 10% of all queries use genuine XQuery properties, which are properties that cannot be expressed in XPath 1.0 or XPath 2.0, such as sorting and user-defined recursive functions.

MBench was designed for micro-benchmarking, and we tested if it really can be used for this purpose. We conclude that the set of 4 queries designed for micro-benchmarking value-based joins is insufficient for drawing sound conclusions. As a result of our investigation, we identified a new sub-problem of the XQuery join evaluation problem: detecting the joins expressed in XQuery. We extended the set of 4 MBench join queries to a micro-benchmark testing join detection.

So, what can we learn from using the 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 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 make the following recommendations:

- The XQuery community will benefit from new benchmarks—both application benchmarks and micro-benchmarks—that are well-designed, stable, scalable, extensible. Both types of benchmarks should have a good coverage of XQuery functionality.
- Application benchmarks could be extensions of XMark and thus benefit from its good properties. Among the good properties of XMark, we note especially the ease of running it on documents of increasing size and the fact that it is often used in the scientific papers and thus serves as a reference for comparison.
- The micro-benchmarks should consist of clear, well-described categories of queries,

in the spirit of MBench, as advocated in [10].

- Each of these categories should be as small as possible in the number of “information needs” it contains.
- Each information need should be formalized as an XQuery query which focuses on the structure of the query and the desired output. When testing for a particular functionality, the use of other features of the language should be avoided. It is also desirable to program the information need in as many different natural ways as possible, using different (every possible) syntactic constructs of XQuery.
- The use of standardized benchmarks (or standardized parts of them) is strongly encouraged. Experiments must be well documented and reproducible. A standardized testing platform like XCheck [9] can help in running the benchmarks and making experiments comparable.

We hope that this study will facilitate the use of the benchmarks.

Acknowledgments

We are grateful to Ioana Manolescu for extensive comments.

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] Penn Treebank: A corpus of parsed sentences. <http://www.cs.washington.edu/research/xmldatasets/data/treebank>.
- [6] Xalan: an implementation of XPath 1.0. <http://xalan.apache.org/>, 2002.
- [7] X-Hive/DB: An XML Database System Management. <http://www.x-hive.com/products/db/index.html>, 2005.
- [8] S. Abiteboul, R. Hull, and V. Vianu. *Foundations of Databases*. Addison Wesley, 1995.
- [9] L. Afanasiev, M. Franceschet, M. Marx, and E. Zimuel. XCheck: a Platform for Benchmarking XQuery Engines. In *Proceedings of VLDB, Demo*, pages 1247–1250, Seoul, Korea, September 2006. ACM Press.

- [10] 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.
- [11] Axyana software. Qizx/Open: An open-source Java implementation of XQuery. <http://www.axyana.com/qizxopen>, 2006.
- [12] 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.
- [13] T. Böhme and E. Rahm. XMach-1: A benchmark for XML data management. In *Proceedings of BTW2001*, Oldenburg, 2001.
- [14] 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 *Proceedings of SIGMOD*, pages 479–490, 2006.
- [15] J. Bosak. Shakespeare. <http://www.ibiblio.org/xml/examples/shakespeare/>.
- [16] 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.
- [17] N. Bruno, N. Koudas, and D. Srivastava. Holistic twig joins: optimal XML pattern matching. In *SIGMOD Conference*, pages 310–321, 2002.
- [18] D. Chamberlin, J. Robie, and D. Florescu. Quilt: An XML Query Language for Heterogeneous Data Sources. In *Proceedings of WebDB*, LNCS. Springer-Verlag, 2000.
- [19] 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: An implementation of XQuery. <http://www.galaxquery.org>, 2006.
- [20] Georgetown Protein Information Resource. Protein Sequence Database. <http://www.cs.washington.edu/research/xmldatasets/>, 2001.
- [21] G. Gottlob and C. Koch. Monadic Queries over Tree-Structured Data. In *Logic in Computer Science*, pages 189–202, Los Alamitos, CA, USA, July22–25 2002. IEEE Computer Society.
- [22] M. H. Kay. SaxonB. An XSLT and XQuery processor. <http://saxon.sourceforge.net>, 2006.
- [23] S. Manegold. An Empirical Evaluation of XQuery Processors. In *In proceedings of ExpDB 2006*, pages 40–53, Chicago, Illinois, USA, June 2006. ACM Press.
- [24] G. Miklau. UW XML Repository. <http://www.cs.washington.edu/research/xmldatasets>.
- [25] MonetDB/XQuery. An XQuery Implementation. <http://monetdb.cwi.nl/XQuery>, 2006.

- [26] 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.
- [27] A. Sahuguet, L. Dupont, and T.-L. Nguyen. Kweelt: a framework to query XML data. <http://kweelt.sourceforge.net/>.
- [28] 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.
- [29] World Wide Web Consortium. XQuery Test Suite. <http://www.w3.org/XML/Query/test-suite/>, 2006.
- [30] World Wide Web Consortium. XQuery Test Suite Result Summary. <http://www.w3.org/XML/Query/test-suite/XQTSReport.html>, 2006.
- [31] World Wide Web Consortium. XQuery 1.0: An XML Query Language. <http://www.w3.org/TR/xquery>, January 2007.
- [32] World Wide Web Consortium. XQuery 1.0 and XPath 2.0 Formal Semantics. <http://www.w3.org/TR/xquery-semantics/>, January 2007.
- [33] B. Yao, T. Özsu, and N. Khandelwal. XBench benchmark and performance testing of XML DBMSs. In *Proceedings of ICDE*, pages 621–633, 2004.

Fig. 6. XMach-1, X007, XMark, and XBench on Galax, SaxonB, Qizx/Open, and MonetDB/XQuery.

Fig. 7. Xbench TC/SD, DC/SD, TC/MD, and DC/MD on Galax, SaxonB, Qizx/Open, and MonetDB/XQuery.

```

Aa:
for $e1 in doc()//eNest[@aSixtyFour=0],
  $e2 in doc()//eNest[@aSixtyFour=0]
where $e2/@aUnique1=$e1/@aUnique1
return
(data($e1/@aUnique1),data($e2/@aUnique1))

Ab&:
for $e1 in doc()//eNest[@aSixtyFour=0],
  $e2 in doc()//eNest[@aSixtyFour=0]
where $e1/@aFour=$e2/@aSixteen and
  $e1/@aSixtyFour=$e2/@aSixteen
return
(data($e1/@aUnique1),data($e2/@aUnique1))

Ab&&:
for $e1 in doc()//eNest[@aSixtyFour=0],
  $e2 in doc()//eNest[@aSixtyFour=0]
where $e1/@aFour=$e2/@aSixteen and
  $e1/@aSixtyFour=$e2/@aSixteen and
  $e1/@aSixtyFour=$e2/@aFour
return
(data($e1/@aUnique1),data($e2/@aUnique1))

Ba:
for $e1 in doc()//eNest[@aSixtyFour=1],
  $e2 in doc()//eNest[@aSixtyFour=0]
where $e1/@aLevel=$e2/@aLevel
return
(data($e1/@aUnique1),data($e2/@aUnique1))

Ba&v:
for $e1 in doc()//eNest[@aSixtyFour=1],
  $e2 in doc()//eNest[@aSixtyFour=0]
where $e1/@aFour=$e2/@aFour and
  $e1/@aSixteen=$e2/@aSixteen or
  $e1/@aLevel=$e2/@aLevel
return
(data($e1/@aUnique1),data($e2/@aUnique1))

Bref&:
for $e1 in doc()//eOccasional,
  $e2 in doc()//eNest[@aSixtyFour=0]
where $e2/@aUnique1=$e1/@aRef and
  $e2/@aSixtyFour=$e1/parent::*/@aSixtyFour
return
(data($e1/@aUnique1),data($e2/@aUnique1))

Ab:
for $e1 in doc()//eNest[@aSixtyFour=0],
  $e2 in doc()//eNest[@aSixtyFour=0]
where $e1/@aFour=$e2/@aSixteen
return
(data($e1/@aUnique1),data($e2/@aUnique1))

Abv:
for $e1 in doc()//eNest[@aSixtyFour=0],
  $e2 in doc()//eNest[@aSixtyFour=0]
where $e1/@aFour=$e2/@aSixteen or
  $e1/@aSixtyFour=$e2/@aSixteen
return
(data($e1/@aUnique1),data($e2/@aUnique1))

Ab&v:
for $e1 in doc()//eNest[@aSixtyFour=0],
  $e2 in doc()//eNest[@aSixtyFour=0]
where $e1/@aSixtyFour=$e2/@aSixteen and
  $e1/@aSixtyFour=$e2/@aFour or
  $e1/@aFour=$e2/@aSixteen
return
(data($e1/@aUnique1),data($e2/@aUnique1))

Ba&:
for $e1 in doc()//eNest[@aSixtyFour=1],
  $e2 in doc()//eNest[@aSixtyFour=0]
where $e1/@aLevel=$e2/@aLevel and
  $e1/@aSixteen=$e2/@aSixteen
return
(data($e1/@aUnique1),data($e2/@aUnique1))

Bref:
for $e1 in doc()//eOccasional,
  $e2 in doc()//eNest[@aSixtyFour=0]
where $e2/@aUnique1=$e1/@aRef
return
(data($e1/@aUnique1),data($e2/@aUnique1))

```

Note: the document name must be given as an argument to the doc() function.

The attributes of the eNest elements, their values and the correlations between them:

aUnique1: contains the number of the element in the document order;

aUnique2: contains a randomly generated number;

aFour: equals aUnique2 mod 4;

aSixteen: equals (aUnique1 + aUnique2) mod 4;

aSixtyFour: equals aUnique2 mod 64.

This implies that the queries Ab, Ab&, Abv, Ab&&, Ab&v are equivalent, and Ba, Ba& are equivalent on the Michigan documents.

The document-level equivalence of Bref and Bref& is based on the properties of the eOccasional elements.

For more details see the Michigan benchmark document generator.

Fig. 8. The where variant of the equi-join micro-benchmark query set, based on the documents of the Michigan benchmark.