Querying XML: benchmarks and recursion
Afanasiev, L.

Citation for published version (APA):

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: http://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.
Chapter 3

Analysis of XQuery Benchmarks

In this chapter, we describe and analyze five XQuery benchmarks that had been published by the database research community by 2006: XMach-1, X007, XMark, MBench, and XBench. We start by characterizing the benchmarks’ targets, measures, and workload properties and by comparing the benchmarks based on these characterizations (Section 3.2). This provides for a short summary of the benchmarks and a quick look-up resource. In order to better understand what the benchmarks measure, we conduct an analysis of the benchmark queries (Section 3.3). Next, we conduct a benchmark usage survey, in order to learn whether and how the benchmarks are used (Section 3.4). When trying to run the benchmarks on XQuery engines, we discover that a large percentage of the benchmark queries (29%) contain errors or use outdated XQuery dialects. We correct and update the benchmark queries (Section 3.5). Once the benchmarks contain syntactically correct queries, we run them on four open-source XQuery engines and we look to draw conclusions about their performance based on the benchmark results (Section 3.6). Based on the obtained benchmark results, we analyze the micro-benchmarking properties of MBench (Section 3.7). Finally, we conclude and give recommendations for future XML benchmarks based on the lessons learned (Section 3.8).

The observations made in this chapter serve as motivation for the work presented in the rest of this thesis, especially in Chapter 5, 6, and 7. All the experiments in this chapter are run with XCheck, a testing platform presented in Chapter 5.

This chapter is based on work previously published in Afanasiev and Marx, 2006, 2008. The study was conducted in 2006. Between 2006 and the time of writing this thesis (mid 2009), two more XQuery benchmarks have been proposed: an application benchmark, called TPox Nicola et al., 2007, and a repository of micro-benchmarks, called MemBeR, which we present in Chapter 6. The complete list of current XQuery benchmarks is given in Section 2.4.1.
3.1 Introduction

Benchmarks are essential to the development of DBMSs and any software system, for that matter \[Jain, 1991\]. As soon as XQuery became a W3C working draft in 2001, benchmarks for testing XQuery processors were published. Nevertheless, using benchmarks for performance evaluation of XML query processors is not (yet) common practice. At the time this work was conducted, in 2006, five XQuery benchmarks had been proposed, but there was no survey of their targets, properties, and usage. It was not clear what the benchmarks measure, how they compare, and how to help a user (i.e., a developer, a researcher, or a customer) choose between them. It was also not clear whether these benchmarks (each separately, or all together) provide a complete solution toolbox for the performance analysis of XQuery engines.

In this chapter, we provide a survey and an analysis of the XQuery benchmarks publicly available in 2006: XMach-1 \[Böhme and Rahm, 2001\], XMark \[Schmidt et al., 2002\], X007 \[Bressan et al., 2001b\], the Michigan benchmark (MBench) \[Runapongsa et al., 2002\], and XBench \[Yao et al., 2004\]. The main goal is to get an understanding of the benchmarks relative to each other and relative to the XQuery community’s need for performance evaluation tools. We believe that this analysis and survey are valuable for both the (prospective) users of the existing benchmarks and the developers of new XQuery benchmarks. Henceforth, we refer to the 5 benchmarks mentioned above as the benchmarks.

We approach our goal by addressing the three questions below.

3.1 Question. What do the benchmarks measure? or What conclusions can we draw about the performance of an engine from its benchmark results?

Every benchmark contains a workload and a measure. The benchmark results of an engine consist of the measurements obtained with the benchmark measure on the benchmark workload. First of all, the results inform us about the engine’s performance on that particular workload. Thus, the benchmark can be considered a performance test case. Test cases are very useful in discovering engines’ pitfalls. Most often, though, we want the benchmark results to indicate more general characteristics of an engine’s performance. For example, we want to predict an engine’s performance in application scenarios with similar workload characteristics. Answering Question 3.1 means understanding how to interpret the benchmark results in terms of the workload characteristics and what we can infer about an engine’s performance in more general terms.

We start tackling Question 3.1 in Section 3.2 by summarizing each benchmark, its target, workload, and performance measure. We characterize the benchmarks by using a list of important parameters of the benchmark’s target, application

\[1\]Technically, X007 is not an XQuery benchmark, since its queries are expressed in a predecessor of XQuery.
3.1. Introduction

scenario, performance measure, data and query workload, and we compare them based on these parameters. The goal of this section is to provide a general overview of the benchmarks and their parameters. This overview helps interpreting the benchmarks’ results in terms of their parameters and it helps determining what features of XQuery processing are not covered by the benchmarks.

Next, in Section 3.3, we analyze the benchmark queries to determine what language features they target and how much of the XQuery language they cover. Our goal is to understand the rationale behind the collection of queries making up each benchmark.

The benchmark queries are built to test different XQuery language features; each benchmark defines the language features it covers. In Section 3.3.1, we gather these language features in an integrated list. Firstly, as a result, a set of features that all the benchmarks designers find important and agree upon emerge. Secondly, we use the integrated list to describe the benchmark queries and obtain a map of feature coverage. In the process, we notice that the queries often use more than one language feature while labeled by the benchmark as testing only one of them. It is not always clear which of the features used influences the performance times.

In Section 3.3.2, we measure how representative the benchmark queries are of the XQuery language. The approach we take is to investigate how much of the XQuery expressive power the queries require, by checking whether the queries can be equivalently expressed in (fragments of) XPath. If we consider only the retrieval capabilities of XQuery (no XML creation), 16 of the 163 benchmark queries could not be expressed in XPath 2.0. This indicates that the benchmark queries are biased toward testing XPath, while XQuery features, such as sorting, recursive functions, etc. are less covered.

3.2. Question. How are the benchmarks used?

Answering this question helps understanding what the needs for performance evaluation tools are.

We look at the usage 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. Instead, the remaining papers use ad-hoc experiments to evaluate their research results. Section 3.4 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 3.5 describes the kind of errors we encountered and the way we corrected them. Having all queries in the same syntax made it possible to systematically analyze them.
Another reason why 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. Remember from Section 2.4.1 that 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. Out of the five benchmarks only the Michigan benchmark (MBench) was designed for micro-benchmarking [Runapongsa et al., 2002]. The rest of the benchmarks are application benchmarks. We investigate the micro-benchmarking properties of the MBench queries in Section 3.7.

3.3. QUESTION. *What can we learn from running the benchmarks?*

We gathered the five benchmarks, corrected their queries, brought them to a standard format, and analyzed their properties together. This gives us an excellent opportunity to test engines against them and analyze the results across benchmarks.

In Section 3.6 we run the benchmarks on four XQuery engines: Galax [Fernández et al., 2006], SaxonB [Kay, 2009], Qizx/Open [Axyana Software, 2006], and MonetDB/XQuery [Boncz et al., 2006b], and analyze their results. First, we found that benchmarks are suitable for finding the limits of an engine. An analysis of the errors raised by an engine—syntax, out-of-memory, out-of-time—is useful in determining its performance. For example, all the engines except SaxonB raised syntax errors, which indicates their non-compliance to the W3C XQuery standard. Next, we found that the engines’ performance differs across the benchmarks. For example, performance rankings based on the average query processing times of an engine are different per benchmark. Finally, we observed that the engines exhibit differences in performance across benchmarks on queries that test the same language feature. This means that the performance of an engine on a language feature obtained against a benchmark cannot be generalized to the other benchmarks. This could be explained by the difference in benchmark data characteristics or by poor query design.

In Section 3.7 we look at the performance of Qizx/Open on a set of MBench queries that test the performance of joins on attribute values. The goal is to investigate how adequate the MBench queries are for micro-benchmarking. The engine exhibits abnormal behavior on a subset of the join queries, but the results are not sufficient to precisely indicate what query parameter causes the problem. This is because several query parameters are varied in parallel and it is not clear which parameter influences the results. MBench did not manage to isolate the influence of different parameters, which would be necessary for conclusive results. We extend the set of queries and run a new experiment to determine which query parameter is responsible for the bad performance.
With this, our analysis ends. We conclude and describe guidelines for future work in Section 3.8.

3.2 Summary of existing XQuery benchmarks

In this section, we first introduce the benchmarks, then we compare their properties. The goal is to give a general overview of the benchmarks and provide a quick look-up table of their details and properties. This overview helps interpreting the benchmarks results and determining what features of the XQuery processing are not covered by the benchmarks.

Throughout this section, we use notions and terms introduced in Chapter 2. We refer to Section 2.4 for a description of existing types of benchmarks and their properties and to Section 2.1.2 for a list of XML document characteristics that we use to describe and compare the benchmarks. The first occurrence of a term defined in Chapter 2 is indicated in italic.

3.2.1 Introducing the benchmarks

In this subsection, we introduce the benchmarks, one by one, in the order of their respective publication years. We describe their target, workload and measure. The target is described in terms of the targeted system under test (SUT) and component under study (CUS).

XMach-1

XMach-1 was developed at the University of Leipzig in 2000 and published in the beginning of 2001 [Böhme and Rahm, 2001]. It is the first published XML database benchmark. Its objective is the evaluation of an entire DBMS in a typical XML data management scenario, which the benchmark defines to be a multi-user, multi-document, and multi-schema scenario. XMach-1 is based on a web application. The workload consists of a mix of XML queries and updates expressed over a collection of text-centric XML documents. The performance measure is the query throughput of the DBMS measured at the level of the end user of the web application. Thus, the targeted SUT is the whole web application and the CUS is the DBMS.

The workload The benchmark data consists of a collection of small text-centric documents and one structured document describing the collection (the catalog). The schemas are described by DTDs. A distinctive feature of XMach-1 is the support of a large number of document schemas with approximately 20 documents per schema. The benchmark scales by producing collections of increasing sizes: 100 documents (adding up to 2.3MB), 1000 (adding up to 23MB), 10,000 (adding
up to 219MB), etc. The text-centric documents contain mixed-content elements, a
typical property of marked-up text. The data-centric document contains recursive
elements that add to the complexity of query processing.

The XMach-1 query set consists of 8 queries and 3 update operations. The
queries aim at covering a wide range of query language features, like navigational
queries, sorting, grouping operators, text search, etc., while remaining simple.
Update operations cover inserting and deleting of documents as well as changing
attribute values. The queries are expressed in natural language and XQuery; the
update operations are expressed in natural language only.

The benchmark defines a workload mix with firm ratios for each operation.
The mix emphasizes the retrieval of complete documents (30%) whereas update
operations only have a small share (2%). The workload simulates a real workload
in the user application scenario.

Performance measures  XMach-1 measures query throughput in XML queries
per second (Xqps). The value is measured based only on one query that tests
simple document retrieval, while running the whole workload in a multi-user
scenario during at least an hour of user time.

XMark

This benchmark was developed at the National Research Institute for Mathemat-
ics and Computer Science (CWI) in the Netherlands and made public in the
beginning of 2001 [Schmidt et al., 2001] and published at VLDB in the middle of
2002 [Schmidt et al., 2002]. The benchmark focuses on the evaluation of the query
processor (the CUS)—as is reflected by the large number of query operations—
of a DBMS (the SUT). The benchmark runs in a single-user, single-document
scenario. The benchmark data is modeled after an internet auction database.

The workload  The XMark document consists of a number of facts having a
regular structure with data-centric aspects. Some text-centric features are in-
trouced by the inclusion of textual descriptions consisting of mixed-content ele-
ments. The benchmark document scales by varying a scaling factor in a contin-
uous range from 0 to 100, or bigger. The scaling factor 0.1 produces a document of
size 10MB and the factor 100 produces a document of size 10GB. The scalability
is addressed by changing the fan-out of the XML tree. The documents conform
to a given DTD.

XMark’s query set is made up of 20 queries, no update operations are spec-
ified. The queries are designed to test different features of the query language
and are grouped into categories: (i) exact match; (ii) ordered access; (iii) type
casting; (iv) regular path expressions; (v) chasing references; (vi) construction
of complex results; (vii) joins on values; (viii) element reconstruction; (ix) full
text search; (x) path traversals; (xi) missing elements; (xii) function application;
3.2. Summary of existing XQuery benchmarks

(xiii) sorting; (xiv) aggregation. Some queries are functionally similar to test certain features of the query optimizer. The queries are expressed both in natural language and in XQuery.

**Performance measures** The performance measure consists of the query execution time of each query in the workload, measured in seconds.

**X007**

This benchmark X007 was published shortly after XMark [Bressan et al., 2001b] and it was developed at the National University of Singapore. It is derived from the object oriented database benchmark OO7 [Carey et al., 1993] with small changes in the data structure and additional operation types to better cover the XML usage patterns. In contrast to XMach-1 or XMark, no specific application domain is modeled by the data. It is based on a generic description of complex objects using component-of relationships. The database is represented by a single document. The benchmark targets the evaluation of the query processor (the CUS) of a DBMS (the SUT) in a single-user scenario.

**Workload** A X007 document has a regular and fixed structure with all values stored in attributes and it exhibits a strong data-centric character. Similar to XMark, some text-centric aspects are included using elements with mixed content. The benchmark provides 3 sets of 3 documents of varying sizes, small (4.5MB, 8.7MB, 13MB), medium (44MB, 86MB, 130MB), and large (400MB, 800MB, 1GB), for testing data scalability. The document scalability is achieved by changing the depth and the fan-out of the XML tree. The document contains recursive elements and the depth is controlled by varying the nesting of recursive elements. The width is controlled by varying the fan-out of some elements.

The workload contains 22 queries. The X007 queries are written in Kweelt [Sahuguet et al., 2000]—an enriched and implemented variant of Quilt [Chamberlin et al., 2000]. Quilt is an XML query language that predates, and is the basis of, XQuery. The rationale behind the query set is to cover the important language features, which are to query both data-centric and text-centric XML documents [Bressan et al., 2001b]. The queries fall into 3 groups: (i) traditional database queries; (ii) navigational queries; and (iii) text-search queries.

**Performance measures** The benchmark deploys the following performance measures: (i) the query execution time for each query and each group of queries [Bressan et al., 2001b] present 18 queries, while the benchmark website http://www.comp.nus.edu.sg/~ebh/X007.html presents 22 queries. The two sets of queries intersect but the first is not included in the second. We consider the queries on the website as the normative ones.
in the workload, measured in seconds; (ii) the time it takes to load the data, measured in seconds; and (iii) the space it requires to store the data, measured in MB.

The Michigan benchmark, MBench

The Michigan benchmark (MBench) was developed at the University of Michigan and published in 2002 [Runapongsa et al. 2002, Runapongsa et al. 2003]. In contrast to its predecessors (XMach-1, X007, XMark, M Bench, and XBench), it is designed as a micro-benchmark aimed at evaluating the query processor (the CUS) of a DBMS (the SUT) on individual language features. Therefore, it abstracts away from any specific application approaches defining only well-controlled data access patterns. The benchmark runs in a single-user and single-document scenario.

The workload

The MBench document has a synthetic structure created to simulate different XML data characteristics and to enable operations with predictable costs. The data structure consists of only one element that is nested with a carefully chosen fan-out at every level. With an element hierarchy of 16 and a fixed fan-out for each level most of the elements are placed at the deepest level. A second element is used to add intra-document references. The first element contains a number of numeric attributes which can be used to select a well defined number of elements within the database. With only two element types and the large number of attributes the data has clearly data-centric properties. Text-centric features are also present since every element has mixed content and the element sibling order is relevant.

The Michigan benchmark document scales in three discrete steps. The default document of size 46MB is arranged in a tree of a depth of 16 and a fan-out of 2 for all levels except levels 5, 6, 7, and 8, which have fan-outs of 13, 13, 13, 1/13 respectively. The fan-out of 1/13 at level 8 means that every 13th node at this level has a single child, and all other nodes are childless leaves. The document is scaled by varying the fan-out of the nodes at levels 5–8. For the document of size 496MB the levels 5–7 have a fan-out of 39, whereas level 8 has a fan-out of 1/39. For the document of size 4.8GB the levels 5–7 have a fan-out of 111, whereas level 8 has a fan-out of 1/111.

MBench defines 56 queries that are grouped into five categories: (i) selection queries; (ii) value-based join queries; (iii) pointer-based join queries; (iv) aggregate queries; and (v) updates. Within each group, often the queries differ only with respect to a single query parameter such as query selectivity to measure its influence on query performance. The queries are defined in natural language, SQL, XQuery, and, partially, in XPath. The update queries are defined only in natural language.
3.2. Summary of existing XQuery benchmarks

Performance measures  The performance measure is the query execution time for each query in the workload, measured in seconds.

XBench

XBench is a family of four benchmarks developed at the University of Waterloo and published in [Yao et al., 2002], at the same time as MBench. XBench characterizes database applications along the data types data-centric (DC) and text-centric (TC), and data organizations in single-document (SD) or multiple-document (MD). The result is four benchmarks that cover different application scenarios: DC/SD simulates an e-commerce catalog, DC/MD simulates transactional data, TC/SD simulates online dictionaries, and TC/MD simulates news corpora and digital libraries. The benchmarks aim at measuring the performance of the query processor (the CUS) of a DBMS (the SUT) in a single-user scenario.

The workload  The benchmark data simulate existing XML and relational data collections. The TC classes of documents use the GNU version of the Collaborative International Dictionary of English [3], the Oxford English Dictionary [4], the Reuters news corpus [5] and part of the Springer-Verlag digital library [6]. The schema and the data statistics of these collections are combined into the synthetic documents of the benchmarks. The DC classes of documents use the schema of TPC-W [7] that is a transactional web e-Commerce benchmark for relational DBMSs. For all document collections, both XML schemas and DTDs are provided. All the collections are scalable.

The common features of the TC/SD benchmark data are a big text-centric document with repeated similar entries, deep nesting and possible references between entries. The generated XML document is a single big XML document (dictionary.xml) with numerous word entries. The size of the database is controlled by the number of entries with the default value of 7333 and the corresponding document size about 100 MB.

The features of the documents in the TC/MD benchmark are many relatively small text-centric XML documents with references between them, looseness of schema and possibly recursive elements. The XML documents are articles with sizes ranging from several kilobytes to several hundred kilobytes. The size of this database is controlled by the number of articles with a default value of 266 and the default data size of around 100 MB.

XML documents belonging to the DC/SD benchmark are similar to TC/SD in terms of structure but with less text content. However, the schema is more

3
4
5
6
7
strict in the sense that there is less irregularity in DC/SD than in TC/SD—most of the XML documents in DC/SD are translated directly from the relations in TPC-W.

The documents in the DC/MD benchmark are transactional and are primarily used for data exchange. The elements contain little text content. Similar to the DC/SD data, the structure is more restricted in terms of irregularity and depth. The database scalability is achieved by controlling the number of documents with a default value of 25,920 and the default size of around 100MB.

The workload consists of a set of ca. 20 queries per benchmark. The set of queries is meant to cover a substantial part of XQuery’s features and are grouped by targeted functionality: (i) exact match; (ii) (aggregate) function application; (iii) ordered access; (iv) quantifiers; (v) regular path expressions; (vi) sorting; (vii) document construction; (viii) irregular data; (ix) retrieving individual documents; (x) text search; (xi) references and joins; and (xii) data type cast.

**Performance measures** The performance measure of XBench is the query execution time for each query in the workload, measured in seconds.

### 3.2.2 Characterizing and comparing the benchmarks

In this section, we compare the benchmarks introduced above based on the their key features such as evaluation targets, application scenarios, performance measures, and workload properties. This comparison is intended to give a general but complete picture of the existing benchmarks. It can serve as a resource for choosing among the benchmarks for a particular evaluation purpose or application domain. It also helps spotting which features of the XQuery processing are not covered by the benchmarks.

**Lining up the benchmarks**

Tables 3.1 and 3.2 summarize the main benchmark features that we discuss below. For a precise definition of the data parameters listed in these tables, we refer to Section 2.1.2.

A fundamental difference between the benchmarks lies in their evaluation target. With its aim of evaluating a database system in a multi-user scenario XMach-1 covers the user view on the system as a whole. All components of the DBMS like document and query processing, handling updates, caching, locking, etc. are included in the evaluation. The evaluation is done at the client level in a client-server scenario. The other benchmarks restrict themselves to the evaluation of the query processor in a single-user scenario.

Another fundamental difference is the benchmark type: XMach-1, XMark, X007, and XBench are application benchmarks, while MBench is a micro-bench-
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>targeted CUS</td>
<td>a web application</td>
<td>DBMS, query processor</td>
<td>DBMS, query processor</td>
<td>DBMS, query processor</td>
</tr>
<tr>
<td>benchmark type</td>
<td>DBMS</td>
<td>appl. benchmark</td>
<td>appl. benchmark</td>
<td>query processor</td>
</tr>
<tr>
<td># users</td>
<td>multi-user</td>
<td>single-user</td>
<td>single-user</td>
<td>single-user</td>
</tr>
<tr>
<td>performance measure</td>
<td>query throughput (Xqps)</td>
<td>query execution time (sec)</td>
<td>query execution time (sec)</td>
<td>query execution time (sec)</td>
</tr>
<tr>
<td># queries</td>
<td>8</td>
<td>20</td>
<td>23</td>
<td>49</td>
</tr>
<tr>
<td>query language</td>
<td>NL, XQuery</td>
<td>NL, Kweelt</td>
<td>NL, XQuery</td>
<td>NL, XQuery, XPath, SQL</td>
</tr>
<tr>
<td># update operations</td>
<td>3</td>
<td>–</td>
<td>–</td>
<td>7</td>
</tr>
<tr>
<td>language</td>
<td>NL</td>
<td>–</td>
<td>–</td>
<td>NL</td>
</tr>
<tr>
<td>main data type</td>
<td>text-centric</td>
<td>data-centric</td>
<td>data-centric</td>
<td>data-centric</td>
</tr>
<tr>
<td># schemas</td>
<td>ca. #docs/20 DTDs</td>
<td>1 DTD</td>
<td>1 DTD</td>
<td>1 XML Schema</td>
</tr>
<tr>
<td># element types</td>
<td>4*schemas+7</td>
<td>9</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td># recursive types</td>
<td>2</td>
<td>–</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td># mixed-content types</td>
<td>4</td>
<td>1</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td># ID/IDREF types</td>
<td>–</td>
<td>4 ID, 9 IDREF</td>
<td>–</td>
<td>1 ID, 1 IDREF</td>
</tr>
<tr>
<td># data value types</td>
<td>2 (string, datetime)</td>
<td>4 (string, integer, float, date)</td>
<td>4 (string, integer, year)</td>
<td>2 (xs:string, xs:integer)</td>
</tr>
<tr>
<td>tree depth</td>
<td>11, 11, 14</td>
<td>11</td>
<td>8, 8, 10</td>
<td>16</td>
</tr>
<tr>
<td>avg depth</td>
<td>3.8</td>
<td>4.6</td>
<td>7.9, 7.9, 9.9</td>
<td>14.5</td>
</tr>
<tr>
<td>tree fan-out</td>
<td>38, 46, 100</td>
<td>2550–2550000</td>
<td>61, 601, 601</td>
<td>13, 39, 111</td>
</tr>
<tr>
<td>avg fan-out (no leaves)</td>
<td>3.6</td>
<td>3.7</td>
<td>3</td>
<td>2</td>
</tr>
<tr>
<td># documents</td>
<td>100, 1000, 10000, etc.</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>document size</td>
<td>ca. 16KB</td>
<td>11MB–11GB</td>
<td>4.5MB–400MB</td>
<td>46MB, 496MB, 4.8GB</td>
</tr>
<tr>
<td>total dataset size</td>
<td>2.3MB, 23MB, 219MB, 11MB–11GB</td>
<td>4.5MB–400MB</td>
<td>46MB, 496MB, 4.8GB</td>
<td></td>
</tr>
<tr>
<td># elements/KB</td>
<td>10</td>
<td>ca. 14</td>
<td>ca. 9.7</td>
<td>1.4</td>
</tr>
</tbody>
</table>

Table 3.1: Characteristics of the standard XML benchmarks: XMach-1, XMark, X007, and MBench. Here, NL means natural language. For the definition of the data parameters see Section 2.1.2.
Table 3.2: Characteristics of XBench, a family of four benchmarks. As before, NL means natural language.

<table>
<thead>
<tr>
<th>3.2.8</th>
<th>2002</th>
</tr>
</thead>
<tbody>
<tr>
<td>XBench (2002)</td>
<td></td>
</tr>
</tbody>
</table>

| target | SUT | benchmark type | performance measure | # users | # queries | query language | # update operations | main data type | # schemas | # element types | # recursive types | # mixed-content types | # ID/IDREF types | # data value types | tree depth | avg depth | tree fan-out (no leaves) | avg fan-out (no leaves) | # documents | # documents | total dataset size | # elements/KB |
| 3.1.3.6 | 100M. 1000M. 1GB | DBMS, query processor | query execution time (sec) | single-user | 17 19 16 15 | NL, XQuery | – | text-centric | text-centric text-centric data-centric data-centric | – | 2 | 1 ID, 1 IDREF – 1 ID, 1 IDREF – | 1 (xs:string) 3 (xs:string, xs:byte, xs:date) 5 (xs:string, xs:byte, xs:decimal, xs:short, xs:date) 8 (xs:string, xs:byte, xs:decimal, xs:short, xs:int, xs:long, xs:dateTime, xs:date) | 7 6 7 3 9 4 1 6 | 4.3 10.7, 11.3, 20.7 3.7 3.9 4.1 3.7 3.9 4.1 2 576 \(n \langle 1 \), 2 \(n \langle 2 \), 3 \(n \langle 4 \), 4 \(n \langle 576 \) | 10MB, 100MB, 1GB, 10GB | 25.8 3.1, 3.5, 6 ca. 21 ca. 7.6 |
| 3.1.3.6 | 100M. 1000M. 1GB | CUS | query execution time (sec) | single-user | 26–26666 | XML Schema, DTD | – | text-centric | text-centric data-centric data-centric | 1 XML Schema, DTD 6 XML Schemas and DTDs | 2 | | | 733, 7333, 73333, 733333, 7333333, 7333333 | 6 | 9 | 75, 264, 2665 250 | 2KB–1500KB | 1KB–3KB |

Chapter 3. Analysis of XQuery Benchmarks
3.2. Summary of existing XQuery benchmarks

mark. Application benchmarks test performance in particular application scenarios simulated by their workloads. They differ in the performance measure used: XMach-1 measures the query throughput on a workload of a small number of simple and complex queries, while XBench, XMark, and X007 measure the query execution time of twice as many, fairly complex queries stressing various features of the query language. The benchmark results are valid only for the tested or similar workloads.

MBench, on the other hand, targets the performance of a query engine on core language features independent of the user application scenario. Its workload consists of a large number of simple queries often differing only in one query parameter. In this way, MBench performs a systematic evaluation and aims at characterizing the performance of the tested language features in terms of the tested query parameters.

Only two benchmarks, XMach-1 and MBench, consider update operations although they can substantially impact DBMS performance.

The benchmarks aim to accommodate the characteristics of both data types: data-centric and text-centric. XMach-1 and XBench TC/SD and TC/MD emphasize the text-centric aspect the most, while the other benchmarks focus on data-centric properties. Nevertheless, all but the XBench TC/MD and DC/MD benchmarks contain mixed-content elements. Note also that the elements-to-size ratios show big differences of markup density from benchmark to benchmark, and they do not correlate with the main type of data: e.g., XBench TC/SD has text-centric documents with the biggest elements-to-size ratio, while MBench has data-centric documents with the smallest elements-to-size ratio. Together, the benchmarks have a good coverage of both types of data.

With respect to the number of different data value types, XBench DC/MD and DC/SD are the richest with 8 and 5 types, respectively, defined by their XML Schemas. XMach-1, XMark, and X007 rely on DTDs to describe their document structure, thus they defined only character data. Nevertheless, the documents contain string data that can be successfully typecast to other types, e.g., XMark contains dates, integers, and floats besides strings.

Each of the benchmarks supports a parameterized scaling of the data set from kilobytes to gigabytes. They all use synthetic data and provide a data generator for compactness and scalability. XMach-1, XBench TC/MD, and DC/MD use collections of many (100–100000) small documents (average 10KB each). This allows easy scalability of the database and gives flexibility to the database system for data allocation, locking, caching etc. The other benchmarks require the whole database to be a single document. This might pose a significant challenge to XML database systems that perform document-level locking etc., and would make it difficult to use these benchmarks for multi-user processing.

Since one of the strengths of XML lies in flexible schema handling, an XML database can be required to easily handle multiple schemas. This feature is tested in XMach-1. The other benchmarks use a single or a small set of fixed
schema. As a result, the number of element types remains unchanged for different database sizes. Note that MBench has an unrealistically small number of element types (2). This might lead to artificial storage patterns in the systems with an element-determined database organization such as some XML to relational mapping approaches.

### 3.2.3 Conclusions

Our benchmark summary and comparison shows that XMach-1 and MBench have a clear focus and are distinct from the rest. XMach-1 tests the overall performance of an XML DBMS in a real application scenario. MBench tests an XML query processor on five language features on a document with artificial schema and properties, but it allows for systematic control of the document and query parameters. This way, MBench characterizes the performance of the query processor in terms of the tested document and query parameters.

XMark, XOO7, and XBench are similar in many respects. The key difference of the latter are single-document vs multi-document scenarios and different schema characteristics. While the documents for all three benchmarks are designed to simulate real application scenarios, the rationale behind the query set is to pose natural queries that cover important query language features. The main performance measure is the execution time on each query.

Together, the benchmarks have a good coverage of testing the main properties of the XML query processing. We refer to Tables 3.1 and 3.2 for an overview.

One of the properties that is not well covered is the data value types. Most benchmark documents contain string data and integer data, but do not cover the whole range of value types defined by XML Schema, for example. Another missing feature is namespaces. Thus, the benchmarks do not cover more advanced XML features.

### 3.3 Benchmark query analysis

In this section, we analyze the benchmark queries to determine what language features they test and how much of the XQuery language they cover.

The benchmark queries are built to test different language features. Each benchmark defines the language features it covers. In Section 3.3.1, we gather these language features in a unified list and use it to describe the benchmark queries.

In Section 3.3.2, we measure how representative the benchmark queries are of the XQuery language. The approach we take is to investigate how much of XQuery’s expressive power the queries require, by checking whether the queries can be equivalently expressed in (fragments of) XPath.
3.3 Benchmark query analysis

3.3.1 Language feature coverage

The rationale behind the query set of each benchmark is to test important language features. Recall from Section 2.2 that a language feature is a logical operation on the data that can be expressed in the considered query language, e.g., tree navigation, value filters, value joins, etc. The benchmarks cover language features that are often used in the benchmark application scenario and/or that are challenging for the query engines. The queries are categorized according to the language features they test; each query is presented under one language feature.

There is no standard list of important XQuery language features; each benchmark defines the features it tests. Though the lists are different, there is quite a bit of semantic overlap between the defined features.

The goal of this section is to determine what language features the benchmarks test. For this purpose, we reconcile the features that are defined by more than one benchmark and gather them in a duplicate-free list. Then, we categorize the benchmark queries using this new list. This allows us to see what language features the benchmarks target, in a unified way. Further, we observe that, with the exception of the MBench queries, the benchmark queries contain usually more than one language feature that might dominate the query processing times. It is not clear why these queries are presented only under one category.

Below, we present the unified list of language features and their definitions. For each feature, we indicate which benchmark defines it and the terms they use. If no term is indicated, then the term used by the benchmark coincides with the given name. The features are ordered by the number of occurrences in the benchmarks.

**Text search** Expressing text search with the help of the \texttt{fn:contains()} function. (XMach-1, X007: text data handling, XMark: full text search, XBench, MBench: element content selection, string distance selection)

**Ordered access** Expressing conditions on the position of elements in a sequence. (X007: element ordering/order preservation, XMark, XBench, MBench: order sensitive parent-child selection)

**Regular path expressions** Expressing paths with one or more element names unknown, usually with the help of the descendant axis. (X007, XMark, MBench: ancestor-descendant selection, ancestor-nesting in ancestor-descendant selection, ancestor-descendant complex pattern selection)

**Aggregates** Expressing aggregation, such as count, sum, minimum, maximum, and average. (X007: aggregate functions, XMark: aggregation, XBench: function application, MBench: aggregation)

**Element construction** Expressing construction of new elements. (X007: reconstruct new structure, element transformation, XMark: element recon-
struction, construction of complex results, XBench: document construction, MBench: returned structure)

**Value-based joins** Expressing joins on the basis of attribute values or element content. (XMach-1, X007: join, XMark: joins on values, MBench)

**Pointer-based joins** Expressing joins on the basis of attribute values of type ID. (XMark: chasing references, XBench: references and joins, MBench)

**Exact match** Expressing simple string lookups with a fully specified path. (XMach-1, XMark, XBench: exact match, retrieve individual documents)

**Value filters** Setting condition on attribute value or element content. (XMach-1, X007: simple selection and number comparison, string comparison, value filter range query, MBench: exact match attribute value selection)

**Sorting** Expressing sorting a sequence of items by their (string or non-string) values. (X007, XMark, XBench)

**Path traversal** Expressing explicit paths (no wildcards). (XMark, MBench: parent-child selection)

**Missing elements** Testing for empty sequences. (XMark, XBench: irregular data)

**Negation** Expressing boolean negation. (X007, MBench: negated selection)

**Type casting** Casting string values to numeric or other type of values. (XMark, XBench: datatype cast)

**Twig patterns** Expressing twig-like structural conditions. (MBench: parent-child complex pattern selection, ancestor-descendant complex pattern selection)

**Quantifiers** Expressing existential or universal conditions on elements of a sequence. (XBench)

**User-defined functions** Expressing user-defined functions. (XMach-1, XMark: function application)

**Updates** Expressing updates. (XMach-1, MBench)

Note that these language features have different levels of abstraction, some features are logically included in the others. For example, when executing a *value filter*, often *type casting* is applied; queries that contain *aggregate* functions also contain *joins* and *group-by* expressions; or, testing for *missing elements* can be done with the help of the *existential quantifier*. 
### 3.3. Benchmark query analysis

<table>
<thead>
<tr>
<th>Feature</th>
<th>XMach-1</th>
<th>X007</th>
<th>XMark</th>
<th>XBench</th>
<th>MBench</th>
</tr>
</thead>
<tbody>
<tr>
<td>Text search</td>
<td>Q2</td>
<td>Q7</td>
<td>Q14</td>
<td>Q17, Q18</td>
<td>QS11–QS14</td>
</tr>
<tr>
<td>Ordered access</td>
<td>–</td>
<td>Q11, Q12, Q20–Q23</td>
<td>Q2, Q3, Q4</td>
<td>Q4, Q5</td>
<td>QS9, QS10, QS15–QS17</td>
</tr>
<tr>
<td>Regular path expressions</td>
<td>–</td>
<td>Q10</td>
<td>Q6, Q7</td>
<td>Q8, Q9</td>
<td>QS8, QS21–QS27</td>
</tr>
<tr>
<td>Aggregates</td>
<td>–</td>
<td>Q13, Q17</td>
<td>Q20</td>
<td>Q3</td>
<td>QA1-QA6</td>
</tr>
<tr>
<td>Element construction</td>
<td>–</td>
<td>Q9, Q16, Q18</td>
<td>Q10, Q13</td>
<td>Q12, Q13</td>
<td>QR1–QR4</td>
</tr>
<tr>
<td>Value-based joins</td>
<td>Q7, Q8</td>
<td>Q5</td>
<td>Q11, Q12</td>
<td>–</td>
<td>QJ1, QJ2</td>
</tr>
<tr>
<td>Pointer-based joins</td>
<td>–</td>
<td>–</td>
<td>Q8, Q9</td>
<td>Q19</td>
<td>QJ3, QJ4</td>
</tr>
<tr>
<td>Exact match</td>
<td>Q1, Q4, Q5</td>
<td>–</td>
<td>Q1</td>
<td>Q1, Q2, Q16</td>
<td>–</td>
</tr>
<tr>
<td>Value filters</td>
<td>Q6</td>
<td>Q1–Q4, Q6</td>
<td>–</td>
<td>–</td>
<td>QS1–QS7</td>
</tr>
<tr>
<td>Sorting</td>
<td>–</td>
<td>Q14</td>
<td>Q19</td>
<td>Q10, Q11</td>
<td>–</td>
</tr>
<tr>
<td>Path traversal</td>
<td>–</td>
<td>–</td>
<td>Q15, Q16</td>
<td>–</td>
<td>QS18–QS20</td>
</tr>
<tr>
<td>Missing elements</td>
<td>–</td>
<td>–</td>
<td>Q17</td>
<td>Q14, Q15</td>
<td>–</td>
</tr>
<tr>
<td>Negation</td>
<td>–</td>
<td>Q15</td>
<td>–</td>
<td>–</td>
<td>QS35</td>
</tr>
<tr>
<td>Type casting</td>
<td>–</td>
<td>–</td>
<td>Q5</td>
<td>Q20</td>
<td>–</td>
</tr>
<tr>
<td>Twig patterns</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>Q6, Q7</td>
<td>QS28–QS34</td>
</tr>
<tr>
<td>Quantifiers</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>–</td>
</tr>
<tr>
<td>User-defined functions</td>
<td>Q3</td>
<td>–</td>
<td>Q18</td>
<td>–</td>
<td>QU1–QU7</td>
</tr>
<tr>
<td>Updates</td>
<td>M1–M3</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>–</td>
</tr>
</tbody>
</table>

Table 3.3: The language features under which the benchmark queries are listed.
As a by-product of this study we obtain a list of important language features that at least 3 benchmarks are covering: from text search to sorting.

In Table 3.3 we give the correspondence between the benchmark queries and the unified feature list from above. This correspondence is the composition of the old-feature-to-new-feature mapping from above and the correspondence of queries to old features of each benchmark. Note that this table shows the way the queries are listed by the benchmarks, not necessarily their coverage of these features. For example, there is no query that does not contain path traversal expressions or regular path expressions, nevertheless there are benchmarks that do not mention these features in their categories.

3.3.1. Remark. The benchmark queries usually combine more than one language feature. It is not always clear which feature has the biggest impact on the performance times.

As an example, consider Q9 of X007 that is listed under the element construction feature:

```
for $a in doc()/ComplexAssembly/ComplexAssembly/ComplexAssembly/
    ComplexAssembly/BaseAssembly/CompositePart/
    Connection/AtomicPart
return
    <AtomicPart>
    {<$a/@*} 
    {$a/../..}
</AtomicPart>
```

The total performance time of this query depends on the performance time of the long child path traversal, on the ancestor axis implementation, and on the element construction operation. Which time has the biggest share of the total time is not clear.

Next, we measure how much of the XQuery language the queries cover.

### 3.3.2 Query language 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. Since XPath is a well studied language and a fragment of XQuery, it forms a good point of reference for our investigations. We consider four flavors of XPath:
3.3. Benchmark query analysis

**XPath 2.0** [World Wide Web Consortium, 2007] This language is less expressive than XQuery; for example, it does not contain sorting features and user-defined functions. XPath 2.0 and XQuery share the same built-in functions.

**Navigational XPath 2.0** [ten Cate and Marx, 2009] 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** [World Wide Web Consortium, 1999a] 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** [Gottlob and Koch, 2002] This fragment is the navigational fragment of XPath 1.0. 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 over 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 features are not used by the queries.

**Assumptions and query rewriting conventions** Many benchmark queries create new XML content based on the information retrieved 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 require the expressive power of XQuery.

Below, we explain the procedure that we follow for rewriting XQuery queries into XPath queries. Often, XQuery queries use element construction to produce new XML output, while XPath just retrieves parts of the input document(s). We remove the element construction from the generation of the output and return only the content of the retrieved elements. Note that the query semantics changes in this case. The following example illustrates the process. Consider query QR2 from MBench:
Chapter 3. Analysis of XQuery Benchmarks

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, in 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 attribute nodes from the source document and outputs them in the following order:

$e1/@aUnique1,
$e1/eNext[1]/@aUnique1, ..., $e1/eNext[last()]/@aUnique1,
$e2/@aUnique1,
...

where the $e1$, $e2$, etc., are eNest elements given in document order. Thus, the order in which the information is presented is preserved, but the structure of the output elements is changed. The difference between the two queries is that the XPath query outputs a sequence of nodes retrieved from the input document, while the original query uses these nodes to construct new XML elements. Thus the structure and the type of the items in the result sequence changes.

Results Table 3.4 contains the distribution of the benchmark queries over the language fragments defined above. 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.

3.3.3 Conclusions

Based on the observations made in this section, we draw three conclusions:
3.4. Survey of benchmark usage

In this section, we present the results of a survey 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 XML query processors. If yes, how are they used? And if not, what does the community use?

For this survey, we consider the 2004 and 2005 conference proceedings of ICDE, SIGMOD and VLDB. First, we select from the pool of the published articles, those articles that are about XML processing, i.e., articles that are about

### Table 3.4: Query language analysis of the benchmarks.

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

1. There are language features that are covered by all, or all but one, benchmarks, i.e., there is agreement among the benchmark authors about which logical operations a query engine should handle well. These features are at the top of the list in Table 3.3.

2. The benchmark queries, with the exception of those of MBench, combine many language features that might impact the query processing times. As a result, the benchmark queries have an exploratory nature rather than a diagnostic nature: if an engine performs badly on one language feature this might reflect on the total performance time of all queries that cover it, while if an engine performs badly on a query listed under a language feature it does not necessarily mean that the engine performs badly on that language feature.

3. The benchmark query sets are biased towards testing XPath features. This is well argued, since XPath is a large and important fragment of XQuery. Still, some important XQuery features, like sorting and recursion, are not well covered. This can be attributed to the fact that the benchmarks are old relative to the language standard.
Chapter 3. Analysis of XQuery Benchmarks

Table 3.5: Benchmark usage survey statistics.

<table>
<thead>
<tr>
<th>Conference</th>
<th># of papers</th>
<th>with standard benchmarks</th>
<th>Benchmark</th>
<th># of papers</th>
<th>with derivation</th>
</tr>
</thead>
<tbody>
<tr>
<td>VLDB</td>
<td>22</td>
<td>7</td>
<td>XMark</td>
<td>11</td>
<td>5</td>
</tr>
<tr>
<td>SIGMOD</td>
<td>9</td>
<td>4</td>
<td>XBench</td>
<td>2</td>
<td>0</td>
</tr>
<tr>
<td>ICDE</td>
<td>10</td>
<td>2</td>
<td>XQTS (XMT)</td>
<td>2</td>
<td>0</td>
</tr>
<tr>
<td>total</td>
<td>41</td>
<td>13 (31%)</td>
<td>total</td>
<td>13</td>
<td>5</td>
</tr>
</tbody>
</table>

XQuery, XPath, XSLT or other closely related XML query languages, and about query processing. There are 51 such articles. Then, we filter out those articles that do not contain an experimental evaluation. We are left with a pool of 41 papers that are about XML processing and contain experimental evaluations. Note that 80% (41 out of 51) of the articles on XML processing evaluate their research experimentally. We examine the experimental results in these papers and gather information about the data sets and the queries used in these experiments.

The detailed questions that we seek to answer are:

1. How many articles use standard benchmarks and which ones?
2. How many of these articles follow the benchmark methodology and how many deviate from it?
3. What data sets and queries are used in experiments that do not use the standard benchmarks? Characterize the queries in terms of the query languages used.

Detailed statistics including references to the papers we examined can be found on the web: [http://ilps.science.uva.nl/Resources/MemBeR/other-benchmarks/survey.html](http://ilps.science.uva.nl/Resources/MemBeR/other-benchmarks/survey.html). Below, we briefly list the answers to our questions.

Questions 1 and 2. Table 3.5 contains statistics about the benchmarks usage. Out of 41 papers on XML processing containing experiments, 13 use the standard benchmarks: XMark, XBench, and the XMT test from the W3C XML Query Test Suit (XQTS) [World Wide Web Consortium, 2006a]. XMark is the absolute winner with 11 articles referring to it. Out of these, 5 contain experiments that were run only on a few selected benchmark queries, those that contain language features relevant to the research presented. Otherwise, the experiments were run in compliance with the methodology of the respective benchmarks.

Question 3. Table 3.6 contains the statistics about data sets and query languages used. Out of 41 surveyed papers, 33 (80%) contain (instead of or besides using the standard benchmarks) experiments on ad hoc data sets and/or queries. These are conducted to facilitate a thorough analysis of the techniques presented.
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 [Michael Ley, 2006], PennTreeBank [Treebank, 2002], SwissProt [Swiss-Prot and TrEMBL, 1998], NASA [NASA, 2001], and Protein [Georgetown Protein Information Resource, 2001], 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 [Barbosa et al., 2002] and the IBM XML data generator [Diaz and Lovell, 1999].

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 often express tree patterns and only use downwards axes.

**Conclusions**  This survey of benchmark usage in the database scientific community leads to the following conclusions: (i) with the exception of XMark, standard benchmarks are not systematically used for evaluating XML query processors; (ii) instead, the authors design specific experiments to analyze in details the proposed research; (iii) the majority of these experiments are based on fragments of XPath 1.0 and on synthetic data provided by the benchmarks.

We see two possible reasons for the lack of popularity from which the benchmarks suffer. One reason might be that the benchmarks are not easy to use. In the next section, we present more evidence to support this hypothesis. Never-
theless, based on the benchmark analysis presented in the previous sections, we believe that the main reason might be the fact that the benchmarks’ workload and measures are not suitable for the type of experimental analysis conducted in the research papers. Note that the majority of articles use the benchmark data but not the whole benchmark workload including the queries. To take XMark as an example, 6 articles use the full benchmark workload, 5 articles use only a subset of the query set, and 22 articles use only the XMark data set with a custom-made set of queries.

### 3.5 Correcting and standardizing the benchmark queries

In order to check how easy it is to run the benchmarks, we ran them on four open source XQuery engines: Galax, SaxonB, Qizx/Open, and MonetDB/XQuery, and 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 parsed by the engines. Second, some queries of XBench and MBench contain mistakes and raised errors or gave wrong results. And third, no benchmark but XMark is designed to run on engines that implement static type checking, and thus their queries raise errors on those engines. The only benchmark that we ran without any problem is XMark. This could explain why XMark is the benchmark that is most often used.

We corrected the benchmark queries and standardized the way the queries specify the input documents. As a result, we could run the benchmarks on the four engines. In Section 3.6, 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.

#### 3.5.1 Detecting outdated syntax and errors

The benchmarks consist of a set of documents and a set of queries. Recall from Tables 3.1 and 3.2 that 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 [World Wide Web Consortium, 2007b] 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; 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 error: static errors, type errors and dynamic errors [World Wide Web Consortium 2007b]. 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 3.5.3.

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 realistically do.

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 the XQuery standard) cope with the evaluation of all benchmark queries on the smallest (set of) documents of the benchmarks.

In Table 3.7, we present the number of incorrect queries that we found. The results are grouped by benchmark and by the 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 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 3.7, there are 14 queries that raise type errors on MonetDB/XQuery. We will discuss these errors and how we correct them in Section 3.5.3. 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 Qizzx/Open due to diverse errors. If we also consider MonetDB/XQuery (which implements static type checking), then even more queries could not be used to test at least one of the four engines.

### 3.5.2 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. [Afanasiev et al., 2005a] and Section 3.7 below), and

---

**Table 3.7: Number of incorrect queries grouped per benchmark and type of error.**

<table>
<thead>
<tr>
<th></th>
<th>static error</th>
<th>dyn. type error</th>
<th>semantically incorrect</th>
<th>incorrect/total</th>
</tr>
</thead>
<tbody>
<tr>
<td>XMark</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0/20 (0%)</td>
</tr>
<tr>
<td>X007</td>
<td>22</td>
<td>0</td>
<td>0</td>
<td>22/22 (100%)</td>
</tr>
<tr>
<td>XMach-1</td>
<td>8</td>
<td>0</td>
<td>0</td>
<td>8/8 (100%)</td>
</tr>
<tr>
<td>Michigan</td>
<td>5</td>
<td>0</td>
<td>2</td>
<td>7/46 (15%)</td>
</tr>
<tr>
<td>XBench TC/SD</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1/17 (6%)</td>
</tr>
<tr>
<td>XBench DC/SD</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1/16 (6%)</td>
</tr>
<tr>
<td>XBench TC/MD</td>
<td>0</td>
<td>3</td>
<td>0</td>
<td>3/19 (16%)</td>
</tr>
<tr>
<td>XBench DC/MD</td>
<td>0</td>
<td>6</td>
<td>0</td>
<td>6/15 (40%)</td>
</tr>
<tr>
<td><strong>total</strong></td>
<td><strong>35</strong></td>
<td><strong>11</strong></td>
<td><strong>2</strong></td>
<td><strong>48/163 (29%)</strong></td>
</tr>
</tbody>
</table>
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 implementations 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) [World Wide Web Consortium, 2006b]. 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 14,637 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 3.6). Below we discuss the changes we made to the benchmark queries. The resulting syntactically correct benchmarks can be found on the web[,] together with a detailed description of our changes.

Correcting static errors

XMach-1, X007, and MBench contain queries that raise static errors. These errors are due to: (i) non-compliance to the current XQuery specifications, or (ii) 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 [Sahuguet et al, 2000] — an enriched and implemented variant of Quilt [Chamberlin et al, 2000]. 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>

[8http://ilps.science.uva.nl/Resources/MemBeR/other-benchmarks/queries.html]
Chapter 3. Analysis of XQuery Benchmarks

We rewrote it to XQuery as follows:

```xml
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>
```

Correcting dynamic type errors

X007, XBench, and MBench contain type errors generated by: (i) applying the child step to items of atomic type, or (ii) 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:

```xml
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:
3.5. Correcting and standardizing the benchmark queries

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.

Correcting semantically incorrect queries

We found two semantically incorrect queries, namely QS6 and QA2 of MBench. 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 detail 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." [Runapongs et al., 2002]

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() gets into an infinite recursion when it receives 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:

```xml
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.

```xml
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>
```

### 3.5.3 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. The other 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.

Specifying the input of a query

The benchmarks have different ways of indicating the input data. X007 and XMark queries use the \( \text{fn:doc()} \) function with a document URI (usually an absolute file name) as argument. MBench queries invoke the \( \text{fn:collection()} \) 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 X Bench benchmark refers to the input by using a new function \( \text{input()} \) 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 the XQuery standard, by using the \( \text{fn:doc()} \) 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 \( \text{fn:doc()} \) function. XQuery has a special built-in function \( \text{fn:collection()} \) 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 \( \text{fn:doc()} \) 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 \( \text{fn:collection()} \) 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 \textit{collection.xml} that contains the list of documents in the collection and their absolute URIs:

\[
\begin{verbatim}
<collection>
  <doc>/path/doc1.xml</doc>
  <doc>/path/doc2.xml</doc>
  ...
  <doc>/path/docn.xml</doc>
</collection>
\end{verbatim}

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 \texttt{fn:collection()} function call. So the query:

\begin{verbatim}
for $a$ in \texttt{fn:collection()//tagname}
return $a$
\end{verbatim}

becomes:

\begin{verbatim}
let $collection :=$
  for $docURI$ in \texttt{doc("collection.xml")}
    \texttt{//doc/text()}
  return \texttt{doc($docURI)}
for $a$ in $collection//tagname
return $a$
\end{verbatim}

Correcting static type errors

Some engines, e.g., MonetDB/XQuery, implement the 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 \cite{WorldWideWebConsortium2007}. 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,

\begin{verbatim}
for $word$ in \texttt{doc()//dictionary/e}
where some $item$ in $word/ss/s/qp/q$
  satisfies $item/qd$ \texttt{eq} "1900"
return $word$
\end{verbatim}

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

\begin{verbatim}
zero-or-one($item/qd) eq "1900"
\end{verbatim}

The adjusted query passes the static type checker of MonetDB/XQuery.
3.5.4 Conclusion

The main conclusion we draw in this section is that the benchmarks, with the exception of XMark, are not maintained and have become outdated very quickly and thereby unusable. The benchmarks were published in 2001 and 2002, while XQuery became a W3C recommendation only in 2007. The changes that were made to the language in the meantime are not accounted for in the benchmarks. Besides this, we found queries that were incorrect but we could not attribute the reason to outdated syntax, thus we consider them as simply errors. The fact that these errors were not corrected by now is again an indication that these benchmarks are not maintained or used.

Since XQuery became a W3C recommendation in 2007, we expect our corrections to the benchmark queries to last as long as the benchmarks are relevant.

3.6 Running the benchmarks

In the previous sections, we introduced and analyzed the benchmarks themselves; in this section, we discuss what we can learn from using them. 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 ran each query 4 times and we took 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 benchmark data of different sizes:
Chapter 3. Analysis of XQuery Benchmarks

The last two columns indicate the sizes of the benchmark data, whether it is a document or a collection of documents. We picked the largest data sizes so that three out of four engines would manage to process the data and produce an answer on our machine, i.e., the benchmark queries would be doable for the majority of the engines. For MBench, only the data of size 46MB satisfied this condition, hence we consider only one data size for MBench.

Figures 3.1 and 3.2 contain the results of running the benchmarks on these two data sizes on the four engines. The individual benchmark queries are given on the x-axis and the total execution times on the y-axis. For more detailed results, see [http://ilps.science.uva.nl/Resources/MemBeR/other-benchmarks/results.html](http://ilps.science.uva.nl/Resources/MemBeR/other-benchmarks/results.html).

This experiment was first published in [Afanasiev and Marx, 2006]. Independently, a similar experiment that covers more engines and benchmark data sizes was conducted in [Manegold, 2008]. Manegold's findings are in line with ours.

In the following sections, we briefly go through the lessons we learned based on these results.

### 3.6.1 Failed measurements

<table>
<thead>
<tr>
<th></th>
<th>Galax</th>
<th>Qizx/Open</th>
<th>MonetDB/XQuery</th>
</tr>
</thead>
<tbody>
<tr>
<td>XMach-1</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>X007</td>
<td>0</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>XMark</td>
<td>0</td>
<td>4</td>
<td>0</td>
</tr>
<tr>
<td>MBench</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>XBench TC/SD</td>
<td>1</td>
<td>0</td>
<td>2</td>
</tr>
<tr>
<td>XBench DC/SD</td>
<td>1</td>
<td>2</td>
<td>0</td>
</tr>
<tr>
<td>XBench TC/MD</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>XBench DC/MD</td>
<td>0</td>
<td>2</td>
<td>1</td>
</tr>
</tbody>
</table>

Table 3.8: Number of syntax errors raised by the engines.

The first piece of information the benchmark results provide us with is the failed measurements. They occur due to syntax errors and engine crash errors. The
3.6. Running the benchmarks

Figure 3.1: XMach-1, X007, XMark, and XBench on Galax, SaxonB, Qizx/Open, and MonetDB/XQuery.
### Figure 3.2: XBench TC/SD, DC/SD, TC/MD, and DC/MD on Galax, SaxonB, Qizx/Open, and MonetDB/XQuery.

<table>
<thead>
<tr>
<th>Query</th>
<th>Galax</th>
<th>SaxonB</th>
<th>Qizx/Open</th>
<th>MonetDB/XQuery</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>9</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>10</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>11</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>12</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>13</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>14</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>15</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>16</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>17</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>18</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>19</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>20</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Total execution time (sec), XBench TC/SD, doc 10MB

Total execution time (sec), XBench TC/SD, doc 104MB

Total execution time (sec), XBench DC/SD, doc 10MB

Total execution time (sec), XBench DC/SD, doc 100MB

Total execution time (sec), XBench TC/MD, doc 8MB

Total execution time (sec), XBench DC/MD, doc 16MB

Total execution time (sec), XBench DC/MD, doc 160MB

Total execution time (sec), XBench TC/MD, doc 112MB

Total execution time (sec), XBench DC/MD, doc 112MB
latter can have several different causes: out of memory, out of Java heap space, materialization out of bounds, segmentation fault, etc. Table 3.8 lists the number of syntax errors raised by the engines: for Qizx/Open all the errors are related to the \texttt{fn:zero-or-one()} function; the two errors of Galax are caused by the fact that it does not implement the \texttt{preceding} axis; the MonetDB/XQuery errors are diverse, for details see the results web page. SaxonB did not raise syntax errors. Table 3.9 lists the engine crash errors obtained: for Galax, the errors are “materialization out of bounds”; for MonetDB/XQuery, the errors are caused by the large intermediate results that do not fit in main-memory nor in virtual memory. SaxonB and Qizx/Open did not produce crash errors.

<table>
<thead>
<tr>
<th></th>
<th>Galax</th>
<th>MonetDB/XQuery</th>
</tr>
</thead>
<tbody>
<tr>
<td>XMach-1</td>
<td>–</td>
<td>all queries on data 2</td>
</tr>
<tr>
<td>X007</td>
<td>Q18 on data 2</td>
<td>Q5</td>
</tr>
<tr>
<td>XMark</td>
<td>Q11, Q12 on data 2</td>
<td>–</td>
</tr>
<tr>
<td>MBench</td>
<td>–</td>
<td>QJ1, QJ2, QJ3 QA4</td>
</tr>
<tr>
<td>XBench DC/SD</td>
<td>–</td>
<td>Q6</td>
</tr>
<tr>
<td>XBench DC/MD</td>
<td>–</td>
<td>all queries on data 2</td>
</tr>
</tbody>
</table>

Table 3.9: Engine crash errors produced by the engines.

### 3.6.2 Comparing the performance of different engines

It is useful to analyze the benchmark results of an engine in comparison with the results of the other engines. The comparison can be on absolute times, but also on the \textit{shape} of the graphs. For instance, the shapes of the graphs obtained on XMach-1 (Figure 3.1) indicate that Galax has difficulties with Q2 while the other engines do not. Another example is comparing the constant and robust behavior of MonetDB/XQuery on X007 and XMark (Figure 3.1) with the often faster, but unstable performance of Saxon and Qizx/Open. Thus, by comparing the performance of several engines, it is easier to spot problems and bottlenecks.

Often, the goal of running a benchmark is to select the best performing engine among two or more candidates. Thus, the benchmark results are used for ranking the engines. For example, \cite{Boncz2006a} rank 16 engines based on the results obtained on XMark. We want to perform a similar tournament between the four engines using all five benchmarks.

Ranking the engines based on all benchmark results shown in Figures 3.1 and 3.2 is difficult. There are many parameters, such as database size and benchmark type, that influence the engines’ performance and that make it hard to create an informative general ranking of engines. For example, let us consider a ranking based on the average query processing time. It is easy to see from Figure 3.2 that on XBench TC/MD, the engine rankings computed on different data
sizes are almost reversed. The rankings differ also per benchmark: as noticed also by [Boncz et al., 2006a], MonetDB/XQuery outperforms all other engines on XMark, but it has great difficulties with XMach-1 and Xbench DC/MD. For a ranking based on average performance times to be informative there should be a small standard deviation of the performance times across all the measurements for a particular engine. Thus, a ranking based on this measure and all benchmark results is not informative.

3.6.3 Performance on language features

In Section 3.3.1, we categorized the benchmark queries according to the language feature they test. Then we observed that most of the queries, with the exception of those of MBench, express more than one language features at once. Since the benchmark authors do not explicitly argue why the query is considered to test one language feature and not the others that it contains, we concluded that one has to be careful when interpreting the benchmark results and not necessarily attribute the results to the feature that the query is said to test. We are now going to see how robust an engine’s performance is on the queries from the same language feature category among different benchmarks.

As expected, there are engines that exhibit different performance patterns on queries that target the same language feature but are from different benchmarks. Below, we provide three examples based on Figures 3.1 and 3.2. In parenthesis we show the queries that were mapped into the respective language feature category (see Table 3.3).

- **Galax on text search** (Q2 of XMach-1, Q7 of X007, Q14 of XMark, Q17, Q18 of XBench, and QS11, QS12 of MBench): the graphs show that the engine has difficulties with the text search queries on XMach-1, XMark, XBench TC/SD, TC/MD, and DC/SD, but not on X007, XBench DC/MD, and MBench.

- **Qizx/Open on join** (Q7, Q8 of XMach-1, Q5 of X007, Q11, Q12 of XMark, and QJ1, QJ2 of MBench): the engine has difficulties on the join queries of XMach-1 and XMark, but not on X007 and MBench.

- **Qizx/Open on pointer-based join** (Q8, Q9 of XMark, Q19 of all four XBench, and QJ3, QJ4 of MBench): the engine has difficulties on the joins of MBench, but not on the rest of the benchmarks.

The performance variance can be attributed to the difference in benchmark data properties or to poor query design. There are many parameters that can influence the performance of an engine and there is a need for further investigation to determine the cause of the differences in the engines’ behavior.
3.7. Micro-benchmarking with MBench

3.6.4 Conclusions

In this section, we made three observations:

- The five benchmarks are useful for finding the limits of the tested engines.

- Comparing the performance of several engines allows for a quick discovery of performance problems. Ranking engines based on their performance on all the benchmarks is not informative due to large variance in performance along different benchmarks and data sizes.

- The engines’ performance on one language feature on one benchmark cannot be generalized to the rest of the benchmarks that test the same feature. We believe the reason lies not only in the difference in the data properties among the benchmarks, but in the query design. In Section 3.7, we present further arguments to support this claim.

All three observations indicate that the benchmarks are a powerful tool for exploratory performance evaluation. The succinct query sets and the availability of data generators that can vary data size and other data parameters present an important advantage over the W3C XML Query Test Suit (XQTS) [World Wide Web Consortium, 2006a], for example. They also allow for easy expansion of the benchmark design for further investigations of the performance of engines.

3.7 Micro-benchmarking with MBench

In this section, we investigate the micro-benchmarking properties of MBench. Our main goal is to check whether the benchmark queries allow for precise conclusions regarding an engine’s performance on the tested language features.

MBench targets several language features (see Section 3.2). In Section 3.6.3, we observed that one of the engines, namely Qizx/Open, has difficulties on the pointer-based join queries of MBench and not on the queries of other benchmarks testing the same language feature. Following our curiosity about the reason for this behavior, we chose the join queries of MBench for our investigation. We hope this investigation will give us some insights into MBench and micro-benchmarking in general.

First, we describe the join queries of MBench in Section 3.7.1. Then, we analyze Qizx/Open’s query processing times on these queries obtained in previous section and discover that it is difficult to interpret the results. The reason is that the queries vary several parameters at the same time and it is not clear which parameter influences the query execution time. We extend the benchmark query set in order to find out the answer. We present our experiment with Qizx/Open in Section 3.7.2. We conclude in Section 3.7.3.
3.7.1 MBench join queries

In this section, we describe the MBench data and the join queries in detail.

Most (99%) of the elements of the MBench data are of the same type and are called eNest. Each eNest element has numeric attributes with precise value distributions. For example, the attribute aUnique2 of type ID contains a unique integer generated randomly; the attribute aSixtyFour 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, aRef, of type IDREF.

Each query in the MBench query set has two variants, one selecting a small number of elements of the input document 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 query processing times.

The join query set is designed to test how a query processor 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 (value-based) and id/idref (pointer-based) 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 given below.

The query QJ1 is:

```xquery
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>
```
3.7. Micro-benchmarking with MBench

<table>
<thead>
<tr>
<th>Varying query parameters</th>
<th>Expected results</th>
</tr>
</thead>
<tbody>
<tr>
<td>QJ1 (\Rightarrow) QJ2</td>
<td>query selectivity: (1.6% \Rightarrow 6.3%) query processing time grows</td>
</tr>
<tr>
<td>QJ3 (\Rightarrow) QJ4</td>
<td>query selectivity: (0.02% \Rightarrow 0.4%) query processing time grows</td>
</tr>
<tr>
<td>QJ1,QJ2 (\Rightarrow) QJ3,QJ4</td>
<td>average query selectivity: (3.95% \Rightarrow 0.21%) average query processing time decreases</td>
</tr>
<tr>
<td>join type: value-based (\Rightarrow) (id/idref)</td>
<td>syntactic form: (\text{where}) form (\Rightarrow) (\text{if}) form</td>
</tr>
</tbody>
</table>

Table 3.10: Varying the query parameters of the four join queries of the Michigan benchmark and the expected results.

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

The query QJ3 is:

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

QJ4 is obtained from QJ3 by replacing all the occurrences of the attribute name \(\text{aSixtyFour}\) with \(\text{aFour}\). The query selectivity of QJ3 and QJ4 is approximately \(0.4\%\) and \(0.02\%\).

Remark Besides the two parameters described above, namely join type and query selectivity, the queries vary in another parameter, the \(\text{syntactic form}\) used to express the joins. QJ1-QJ2 use the \(\text{where}\) clause to express the join, while queries QJ3-QJ4 use the \(\text{if then else}\) construct. Clearly, these two patterns are equivalent. Moreover, both variants have the same normal form in XQuery.

\(^9\text{Note that even though the selectivity of the subexpression } //\text{eNest[@aFour=3]} \text{ is 16 times larger than the selectivity of the subexpression } //\text{eNest[@aSixtyFour=3]}, \text{ the selectivity of QJ4 is 20 times larger than the selectivity of QJ3. The difference is due to the influence of the \text{eOccasional} elements on the join outcome. For more information about the \text{eOccasional} elements and their attribute values, see [Runapongsa et al., 2002].}\)
### Chapter 3. Analysis of XQuery Benchmarks

<table>
<thead>
<tr>
<th>Query (selectivity)</th>
<th>Query execution time (sec)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>original query</td>
</tr>
<tr>
<td>QJ1 (1.6%)</td>
<td>3.6</td>
</tr>
<tr>
<td>QJ2 (6.3%)</td>
<td>3.8</td>
</tr>
<tr>
<td>QJ3 (.02%)</td>
<td>338.8</td>
</tr>
<tr>
<td>QJ4 (.4%)</td>
<td>396.1</td>
</tr>
<tr>
<td>avg(QJ1,QJ2)</td>
<td>3.7</td>
</tr>
<tr>
<td>avg(QJ3,QJ4)</td>
<td>367.45</td>
</tr>
</tbody>
</table>

Table 3.11: Qizx/Open on the original and modified join queries of MBench.

Core World Wide Web Consortium [2007b], which is a complete fragment of XQuery that is used to specify the formal semantics. The benchmark authors do not explain why this parameter is varied and how it influences the target of the micro-benchmark.

Measure In Runapongsa et al. [2002] the join queries are evaluated on a document of fixed size and the results consist of four query processing time measurements. When analyzing the results, the authors 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 a large impact on the performance times. On the other hand, optimized algorithms should scale better with respect to query selectivity. The authors expect that the id/idref joins scale up better than the simple joins, when, for example, an id-based index is used. And finally, the expectation is that the average query processing time of the id/idref joins is smaller than the average query processing 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 latter queries. The influence of the varying syntactic form is not taken into account in the benchmark measure. In Table 3.10 we list the parameters that vary between the four queries, their values and the expected influence on the results.

In the next section, we analyze Qizx/Open’s results on these queries.

### 3.7.2 Evaluating Qizx/Open on the MBench join queries

In Section 3.6 we ran Qizx/Open on the MBench data of size 46MB (728K nodes). The engine’s execution times on the four join queries, QJ1–QJ4, are presented in the second column of Table 3.11. As expected, the query processing times for QJ2 and QJ4 are larger than those for QJ1 and QJ3, respectively. But the average query processing time for QJ3-QJ4 is 2 orders of magnitude larger than...
the average time for QJ1–QJ2, while we expected the query processing time to decrease.

Does this indicate an abnormality with the Qizx/Open implementation of id/idref joins? Or is the difference in the query processing times maybe due to the variance in the syntactic form? The latter hypothesis sounds more plausible.

We extended the query set with the where and if variants for all four queries and ran the engine on the new queries. The execution times presented in the third and the fourth column of Table 3.11 show that our hypothesis was right. Note that if we fix the syntactic form (i.e., consider one column of Table 3.11), then the results correspond to our initial expectations: the query processing times increase within a join type when the query selectivity increases, and the average query processing time of id/idref joins is smaller than the average query processing time of value-based joins. But the processing times for the if variant are much larger than the performance times for the where variant. Note that the algorithm that Qizx/Open applies for the joins expressed in the where form is efficient—it seems to scale sub-linearly 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-linearly with respect to query selectivity for the simple joins—but scales better for the id/idref joins.

Since in XQuery joins can be expressed syntactically in many different ways, the join processing problem is two-fold: first a join has to be recognized and then the efficient algorithm can be applied. Our extended experiment indicates a problem with the join detection mechanism of Qizx/Open. By separating the influence of the query’s syntactic form from the influence of other parameters, we could interpret the results and learn more about the engine’s join evaluation strategies.

The extended join query set only tests the influence of 3 parameters. Joins are complex operations and there are more parameters that might influence the performance of a join processing technique, for example the number of join conditions. As follow-up work, we further extend this micro-benchmark to thoroughly test the join detection mechanisms on more query parameters. We present this micro-benchmark and experiments on four query processors in Chapter 7.

3.7.3 Conclusions

The application benchmarks, XMach-1, X007, XMark, and XBench, are not suitable for a thorough analysis of a query processing technique. MBench, on the other hand, is a micro-benchmark and it is meant for such an analysis. We investigated the MBench queries that test for value-based and pointer-based joins and found that they fail to isolate the impact of two parameters, namely the join type and the syntactic form, on the performance evaluation and lead to inconclusive results. Not surprisingly, there is an engine for which the benchmark does not
behave as expected and for this engine the results cannot be interpreted.

3.8 Conclusions

In this chapter, we described and studied five standard XQuery benchmarks publicly available in 2006: XMach-1, XMark, X007, MBench, and XBench. The questions we pursued are: Question 3.1 “What do the benchmarks measure?”, Question 3.2 “How are the benchmarks used?”, and Question 3.3 “What can we learn from using them?”. The main conclusion we draw is that the benchmarks are very useful for exploratory performance studies, but not adequate for rigorous performance evaluations of XML query processors. Below, we summarize our answers to each question.

Answering Question 3.1 The benchmark summaries and comparison given in Section 3.2 show that XMach-1 and MBench have a distinct and clear focus, while X007, XMark, and XBench, have a more diffuse focus and are similar in many respects. The key difference between XMach-1, MBench and the rest is the target and performance measure. XMach-1 is an application benchmark that tests the overall performance of an XML DBMS in a real application scenario; the benchmark measure is the query throughput. MBench a micro-benchmark that tests the performance of an XML query processor on five language features on an artificial document; the benchmark measure is the query processing time. X007, XMark, and XBench are application benchmark that test the performance of an XML query processor on a (small) set of (complex) queries. The key difference between them is the document scenario they test: X007, XMark, and XBench TC/SD and DC/SD test single-document scenario, while XBench TC/MD and DC/MD test multi-codument scenario. Tables 3.1 and 3.2 contain a detailed description of benchmark parameters and their values for reference and comparison.

The queries of each benchmark were designed to test important language features. Each query is labeled with one language feature. Table 3.3 contains a mapping of the benchmark queries into the language features they are designed to test. In Section 3.3 we observe that a query usually contains more than one language feature and an engine’s performance on that query should not necessarily be attributed to the language feature with which it is labeled. Thus, the queries have an exploratory nature rather than a diagnostic nature.

Further, also in Section 3.3, we show that 90% of all queries can be expressed in XPath 1.0 or 2.0, if we consider only the element retrieval functionality and ignore the XML construction functionality of XQuery. The remaining 10% of the queries test two XQuery properties: sorting and user-defined recursive functions. Thus the benchmarks measure mainly the performance of XPath features.

When considered together, as a family, the benchmarks have a good coverage of the main characteristics of XML documents and of the important XQuery language features. Nevertheless, they do not cover the whole space of XML query
3.8. Conclusions

processing scenarios and parameters. For example, more advanced XML/XQuery
features, such as typed data, namespaces, recursion, etc., are poorly covered.

Answering Question 3.2 In Section 3.4 we conducted a survey of scientific
articles reported in the 2004 and 2005 proceedings of the ICDE, SIGMOD and
VLDB conferences. The survey shows that fewer than 1/3 of the articles on XML
query processing that provide experimental results use benchmarks (11 papers use
XMark and 2 papers use XBench). The remaining articles use ad-hoc experiments
to evaluate their research results. The majority of these (73%) use benchmark
data sets or real data and ad-hoc query sets. Thus, with the exception of XMark
and XBench, the benchmarks are not used.

One reason for the limited usage of the benchmarks might be that their data
and query set are outdated. For example, the benchmark queries (with the ex-
ception of XMark) do not comply with the W3C XQuery standard that was
finalized five years after the benchmarks were developed. 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 for the limited usage of the benchmarks 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 kind of analysis. In
such cases, specialized micro-benchmarks are more appropriate [Afanasiev et al.,
2005a]. Since MBench was designed for micro-benchmarking, we will comment
on its properties below.

Answering Question 3.3 In Section 3.6 we ran the benchmarks on four
XQuery engines: Galax, SaxonB, Qizx/Open, and MonetDB/XQuery and com-
pared their performance. A first observation we make is that the engines produce
errors and suffer from crashes, which makes the comparison difficult. Next, no
engine can be crowned as a winner. The relative performance of the engines
varies on different benchmarks, which indicates that the engines are tuned for a
particular user or data scenario. The last observation we made is that the en-
gines’ performance on queries testing a specific language feature might differ per
benchmark. This again might be the reason for the difference between data and
user scenario or it might be an indication of poorly designed queries. Thus, it
is important to check an engine on several benchmarks, instead of only one, in
order to get a more complete picture of its performance.

In Section 3.7 we tested whether MBench is suitable for a rigorous analy-
sis of a language feature it targets, namely attribute-value joins. Based on the
benchmark results obtained on an XQuery engine we conclude that the set of four
queries designed for micro-benchmarking joins is insufficient for drawing sound
conclusions about its performance. We conclude that MBench, even though it
provides a good starting point for micro-benchmarking, is incomplete, which leads
to inconclusive results. In Chapter 7 we extend the set of MBench join queries to
a micro-benchmark testing the impact of seven query and document parameters
on join processing techniques.

To summarize, the benchmarks have an exploratory nature and are a good 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. Nevertheless, they are not suitable for a detailed analysis of query processing techniques.

3.8.1 Recommendations and next steps

Based on the study presented in this chapter, we make the following recommendations for future XML benchmarking:

• The XQuery community will benefit from new benchmarks—both application benchmarks and micro-benchmarks—that have a good coverage of XQuery features. A serious investment should be made for maintaining these benchmarks at the same pace as the development of the XQuery engines themselves.

At the time of writing this thesis, another XQuery application benchmark has been proposed, TPox [Nicola et al. 2007], while we present a repository of micro-benchmarks, MemBeR, in Chapter 6 and a join-detection micro-benchmark in Chapter 7.

• 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 (relatively) often used in the scientific papers and thus serves as a reference for comparison.

• Micro-benchmarks should consist of clear, well-described categories of queries, in the spirit of MBench and as advocated in Afanasiev et al. [2005a]. When testing for a particular language feature, the use of other language features should be avoided. It is also desirable to test different ways of expressing the query functionality using different (every possible) syntactic constructs of XQuery.

We follow this recommendation when organizing the micro-benchmarks in MemBeR (the repository of micro-benchmarks presented in Chapter 6) and designing the join-detection micro-benchmark (presented in Chapter 7).

• The use of standardized benchmarks (or standardized parts of them) is strongly encouraged. Experiments must be well documented and reproducible. A testing platform can help in running standardized benchmarks and making experiments comparable.

In Chapter 5, we make an effort in this direction by presenting a platform for running standardized benchmarks, XCheck.