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 2

Background

In this chapter, we introduce the framework of this thesis and define the terminology used throughout it. Readers familiar with XML, XML query languages, XML query processing, and performance evaluation of XML query processors may prefer to skip ahead and return to this chapter only to look up a definition. A thorough study of related work comes in later chapters: we dedicate Chapter 3 to analyzing related work for Part I of the thesis, while Chapters 8 and 9 of Part II each contain discussions of relevant related work.

2.1 The Extensible Markup Language (XML)

In 1998, the W3 Consortium published its first XML Recommendation, which evolved to its fifth edition by 2008 [World Wide Web Consortium, 2008]. Within these 10 years the Extensible Markup Language (XML) has become a standard means for data exchange, presentation, and storage across the Web. The XML format has proven to be versatile enough to describe virtually any kind of information, ranging from structured to unstructured, from a couple of bytes in Web Service messages to gigabyte-sized data collections (e.g., [Georgetown Protein Information Resource, 2001]) and to serve a rich spectrum of applications.

Similar to the Hypertext Markup Language (HTML), XML is based on nested tags. But unlike the HTML tags, which are predefined and specifically designed to describe the presentation of the data, the XML tags are defined by the user and designed to describe the structure of the data. Figure 2.1 shows an example of an XML document that contains university curriculum data. The hierarchy formed by nested tags structures the content of the XML documents.

An XML document may have a schema associated to it. The most common languages are the Document Type Definition (DTD) [World Wide Web Consortium, 2008] and the XML Schema [World Wide Web Consortium, 2004b]. A schema defines the grammar for the XML tags used by the document and specifies constraints on the document structure and data value types. A DTD is
Chapter 2. Background

Figure 2.1: Example of an XML document containing university curriculum data.

```xml
<curriculum>
  <course code="c1">
    <title>Database Techniques</title>
    <lecturer>Martin Kersten</lecturer>
    <lecturer>Stefan Manegold</lecturer>
    <description>Basic principles of database design and database management. Understanding the implementation techniques underlying a relational database management system.</description>
  </course>
  <course code="c2">
    <title>Database-Supported XML Processors</title>
    <lecturer>Torsten Grust</lecturer>
    <assistant>Jan Rittinger</assistant>
    <description>How can relational database management systems (RDBMS) be transformed into the most efficient XML and XQuery processors on the market.</description>
    <prerequisites>
      <precourse ref="c1"/>
    </prerequisites>
  </course>
</curriculum>
```

Figure 2.2: Example DTD describing university curriculum data.

```xml
<!ELEMENT curriculum (course*)>
<!ELEMENT course (title, lecturer+, assistant*, description, prerequisites?)>
<!ATTLIST course code ID #REQUIRED>
<!ELEMENT title (#PCDATA)>
<!ELEMENT lecturer (#PCDATA)>
<!ELEMENT assistant (#PCDATA)>
<!ELEMENT description (#PCDATA)>
<!ELEMENT prerequisites (precourse+)>
<!ELEMENT precourse EMPTY>
<!ATTLIST precourse ref IDREF #REQUIRED>
```

a set of statements that specify: (i) the nesting of XML elements; (ii) the element occurrence constraints; (iii) the containment of attributes in XML elements; and (iv) attribute value types and default values. DTDs can specify a handful of special attribute value types, for example ID, IDREF, ENTITY, NMTOKEN, etc. However, they do not provide fine control over the format and data types of element and attribute values.

XML Schema is a more powerful schema language. It can describe more complex restrictions on element structure and number of occurrences, and it provides a much richer set of data value types than DTDs. In addition to a wide range of built-in simple types (such as string, integer, decimal, and dateTime), the schema language provides a framework for declaring custom data types, deriving new types from old types, and reusing types from other schema.

Figure 2.2 shows the DTD corresponding to the example XML document given above. The XML model differs from the relational model in that it is hierarchical as opposed to flat and it is more flexible. The XML elements can contain only child elements and no character data (element content) or they can contain char-
2.1. The Extensible Markup Language (XML)

Various tree data models were proposed for XML. W3C alone recommends a few different tree models for different purposes: the Document Object Model (DOM) \cite{1998DOM}, the XML Information Set (Infoset) \cite{2004aInfoset}, Post-Schema-Validation Infoset (PSVI) \cite{2004bInfoset}, XPath 1.0 \cite{1999XPath1}, and XPath 2.0/XQuery 1.0 Data Model (XDM) \cite{2007aXDM}. All of these models share XML's basic idea of trees of elements, attributes, and text, but have different ways of exposing that model and handling some of the details.

The basic data model for XML is a node-labeled ordered tree. This model is often used for research purposes \cite{Gou2007}. Figure 2.3 shows the tree of the XML document in Figure 2.1. The tree contains three types of nodes: (i) element nodes, corresponding to tags in XML documents, for example, course; (ii) attribute nodes, corresponding to attributes associated with tags in the XML document, for example, @ref; and (iii) text nodes (or leaf nodes), corresponding to the data values in XML documents, for example, "c1" or "Database

Figure 2.3: The node-labeled ordered tree corresponding to the university curriculum document.

acter data optionally intersected with child elements (mixed content). Elements of the same type might have different numbers and types of sub-elements. For example, in Figure 2.1, the second course, but not the first one, has an assistant sub-element; the first course has two author lecturers, but the second course only one.

One special type of constraint that a schema (DTD or XML Schema) can define is the ID/IDREF constraints: an attribute of type ID contains a unique value within the whole document, while an attribute of type IDREF contains a reference to the element with that unique ID attribute. In our example from Figures 2.1 and 2.2, the attribute ref of type IDREF points to the course element containing the attribute code of type ID with the same value.

2.1.1 The XML tree model

The basic data model for XML is a node-labeled ordered tree. This model is often used for research purposes \cite{Gou2007}. Figure 2.3 shows the tree of the XML document in Figure 2.1. The tree contains three types of nodes: (i) element nodes, corresponding to tags in XML documents, for example, course; (ii) attribute nodes, corresponding to attributes associated with tags in the XML document, for example, @ref; and (iii) text nodes (or leaf nodes), corresponding to the data values in XML documents, for example, "c1" or "Database Techniques".

Figure 2.3: The node-labeled ordered tree corresponding to the university curriculum document.
Chapter 2. Background

Techniques. Edges in the data tree represent structural relationships between elements, attributes, and text. Note that in an XML data tree, element nodes with the same name might be nested on the same path. These elements are called recursive.

2.1.2 XML data characteristics

XML documents can be described by a list of characteristics that are important when considering the functionality and performance of XML processing tools. Below, we present a list characteristics that are often used as parameters in performance evaluation tools [Schmidt et al., 2002, Runapongsa et al., 2002, Afanasiev et al., 2005a].

**Structural characteristics**

**Document size** is the total size of the benchmark document or collection of documents in bytes.

**Elements-to-size ratio** is the amount of mark-up vs the amount of data contained in documents and it is measured in number of XML elements per KB. Typically (but not necessarily), a text-centric document (e.g., a book) contains significantly more text than mark-up, while a data-centric document (e.g., a product catalog) has a rich structure and contains large amounts of mark-up to describe it.

**Tree depth** The tree depth describes the number of levels in the XML tree. The maximum number of elements on a root-leaf path, not counting the root, is called tree depth. The average tree depth is the average length of root-leaf paths, not counting the root.

**Tree fan-out** The tree fan-out describes the number of children in the XML tree. An element’s fan-out is the number of its children. The maximum of the element fan-outs in the tree is called tree fan-out. The average of the element fan-outs of the non-leaf elements in the tree is called average tree fan-out.

**Recursive elements** are elements that can be defined by a self reference. It is not unusual that the tag name of an element is the same as its direct parent or one of its ancestors, generating recursion. For example, a “section” element may have a “section” sub-element. This type of data is notoriously difficult to query.

**Mixed-content elements** are the elements that contain both children elements and text content (character data). This type of elements also poses difficulties to XML processors.
2.1. The Extensible Markup Language (XML)

**Number of element types** is the number of unique tag names in the document. It characterizes the richness of the document structure.

**Document structure type** The high flexibility of XML leads to documents with vastly different structures, from highly structured *data-centric documents* to semi-structured and unstructured *text-centric documents*. Two key features of text-centric documents are the importance of the order of elements, and the irregular element structure. Text-centric documents might contain mixed-content elements. Typical instances of such documents are Web pages. Data-centric documents have a regular structure and can be compared with relational data; the element types contain only child elements or only text content, often representing typed information. Typical instances of such documents are commerce catalogs and transactions.

**Content characteristics**

The data values found in the text nodes and attribute values can be described by the characteristics of the underlying value domains, such as *range*, *distribution*, etc.

**Characteristics in the presence of a schema**

The presence of a schema (DTD, XML Schema, or other) enriches XML documents with more properties.

**Elements with ID/IDREF attributes** Attributes of type *ID* and *IDREF* define unique key and key-reference constraints. These attributes define a link relation between their elements. This relation extends the tree structure of the XML document into a graph structure.

**Number of data value types** Schema languages, such as XML Schema, provide a rich set of simple value types. The element and the attribute content can be of type string, integer, float, date, etc. By default, the content of XML documents is considered to be of type string. When a schema is not provided, the value type can be determined by successful typecasting from string data.

**Data set characteristics**

The XML specification gives an important place to the notion of document, which can be seen as a logical and/or physical segment of an XML data set. XML documents can be combined in *collections* of documents that share a common schema and/or characteristics. An XML *data set* is a set of XML documents or collections. The document characteristics discussed above can also be used
Chapter 2. Background

to describe data sets. Below, we present two characteristics that are data-set specific.

**Number of documents** XML data sets can be *single-document*, i.e., consisting of a single large document (e.g., e-commerce catalogs), or *multi-document*, i.e., consisting of a collection of many smaller (and usually more shallow) documents (e.g., a digital library).

**Number of documents per schema** The data set can be characterized by one or more schemas. Each schema can cover one or more documents in the data set.

Since data set characteristics have an impact on the functionality and performance of XML processing tools, they are typically considered as parameters in performance evaluation tools, such as benchmarks. We use them throughout this thesis for the same purposes: in Chapter 3 we use the characteristics to describe the data sets of five XML query benchmarks; in Chapter 5 we propose an automatic tool for the execution of benchmarks that also records the *data set size*; in Chapter 6 we use the characteristics to organize a repository of XML query benchmarks; in Chapter 7 and 8 we measure the performance of XML query processors with respect to varying values of document parameters.

## 2.2 XML query languages

The amount of data available in XML format raises an increasing demand to store, process, and query XML data in an effective and efficient manner. To address this demand, the W3 Consortium has recommended today’s mainstream query languages: XML Path Language (XPath), version 1.0 and 2.0 [World Wide Web Consortium, 1999a, 2007], XML Query Language (XQuery), Version 1.0 [World Wide Web Consortium, 2007], and Extensible Stylesheet Language Transformations (XSLT), version 1.0 and 2.0 [World Wide Web Consortium, 1999b, 2007c].

Each query language has different language features that make it suitable for a particular query scenario. A *language feature* can be a functionality of the language, a syntactic construct, a property of the language, etc. The main functionalities of XPath are tree navigation. XQuery is particularly good at operations such as joins and sorts. XSLT is meant for transformations of XML trees into other XML trees, into text, or into other textual formats.

In this thesis, we will only address XPath and XQuery.

### 2.2.1 XPath

XPath is the basic XML query language, it selects nodes from existing XML documents. It introduces a convenient syntax for describing paths from the root
or from a given context node in the document tree to the nodes to be selected. The language draws its name from this property.

A simple XPath query is formulated as a sequence of path steps separated by a slash sign, /, as in URLs. A simple path step is composed of an axis and a tag. Two most commonly used axes are the child axis, where child::a selects all elements labeled with a that are children of the context node, and the descendant axis, where descendant::a selects all elements labeled with a that are descendants of the context node. These two axes are often omitted creating the popular short-hands / and //, where a/b denotes selecting all elements labeled with b that are children of all elements labeled with a that are children of the context node, and where a//b denotes selecting b-elements that are descendants of a-elements that are children of the context node. The XPath query //course/lecturer selects all lecturer-elements that are children of all course-elements descendants of the root element in the curriculum document from Figure 2.1. In addition to the child axis and the descendant axis, XPath defines 11 other axes to facilitate the XML tree navigation [World Wide Web Consortium, 1999a].

An XPath query can specify more complex path patterns by using predicates. One example is //course[@code="c1"]/lecturer, in which //course/lecturer is the main path of the query, while the content between square brackets is a predicate expressing an extra condition on the course elements on the path. This query selects all lecturer children of the course element whose code is "c1". Generally, an XPath query might involve multiple predicates composed of arbitrary path expressions.

XPath version 1.0 is a relatively simple language. For example, it does not have variables or namespace bindings, it has a very simple type system, it cannot select parts of an XML elements, or create new XML documents. To address some of these issues, XPath version 2.0 was introduced. The main difference is that the new version supports a new query model, shared with XQuery (XQuery 1.0 and XPath 2.0 Data Model (XDM) [World Wide Web Consortium, 2007a]), and that it supports all the simple data value types built into XML Schema, such as xs:date, xs:time, xs:decimal, xs:string, etc.

### 2.2.2 XQuery

XQuery is a strongly typed functional language built on top of XPath 2.0. The core features of XQuery are (i) For-Let-Where-Order by-Return (FLWOR) clauses (pronounced “flower”); (ii) XML node constructors; and (iii) (recursive) user-defined functions. The latter functionality, full recursion at a user’s disposal, turns XQuery into a Turing-complete language [Kepser, 2004].

FLWOR clauses, which can be freely nested and composed, are the main building blocks of XQuery queries. Every FLWOR expression begins with one or more For and/or Let clauses (in any order), followed by an optional Where clause, an optional Order by clause, and finally one Return clause. The For and
for $lecturer in doc("curriculum.xml")//lecturer
let $courses := doc("curriculum.xml")
    //course[lecturer = $lecturer]
    where not(empty($courses))
order by $lecturer/text() ascending
return
<lecturer>
    <name>{$lecturer/text()}</name>
    <course>{$courses/title/text()}</course>
</lecturer>

(a) Query.

<lecturer>
    <name>Martin Kersten</name>
    <course>Database Techniques</course>
</lecturer>
<lecturer>
    <name>Stefan Manegold</name>
    <course>Database Techniques</course>
</lecturer>
<lecturer>
    <name>Torsten Grust</name>
    <course>Database-Supported XML Processors</course>
</lecturer>

(b) Result.

Figure 2.4: An XQuery query that groups the courses given by “curriculum.xml” in Figure 2.1 by lecturer. The results the query produces on “curriculum.xml”.

Let clauses bind a variable to a sequence of items selected by an XPath/XQuery expression—the For clause binds the variable to every item in the sequence in turn, while the Let clause binds the variable to the whole sequence. The Where clause specifies a selection or a join predicate over the bound variables. The Order by clause specifies an order condition on the resulting bounded sequences. Finally, the Return clause constructs the result based on an expression optionally using the bounded variables. Often the expression in the Return clause formats the results in XML format. Figure 2.4 shows an example: the query groups the courses given in the curriculum example in Figure 2.1 by lecturer. Through its FLWOR clauses, XQuery bares similarities to SQL, the well-known query language for relational databases [Gulutzan and Pelzer, 1999].

Constructing XML is useful for several purposes, for example, for organizing the data in a new structure (transformation) and representing temporary intermediate data structures (composition). XQuery has expressions for constructing every XML node kind. For every node kind, it supports two construction expressions: one with a syntax similar to XML and an alternative XQuery syntax primarily used for nodes whose names or contents are computed from XQuery.
2.3 XML query processing

The problem of querying XML is to find in an XML document or an XML collection all matches that satisfy a given query formulated in a query language (e.g., XPath, XQuery). We call this problem XML query processing.

XML query processing has different application scenarios. A common application scenario is the basic scenario, namely retrieving parts of a fully specified document or collection of documents that satisfy a query. This is also referred to as the database scenario. Another application scenarios is selective dissemination of information (SDI) [Altinel and Franklin 2000]. A core component of SDI is a document filter that matches each incoming XML document from publishers with a collection of queries from subscribers, to determine which subscribed queries have at least one match in the document rather than finding all matches.
of the subscribed queries, as required by the basic scenario. In most SDI applications, the data arrives in the form of data streams, i.e., in its original sequential document format, and the queries are evaluated as the data arrives, often on incomplete documents.

In this thesis, we are only concerned with the basic scenario for XML query processing.

2.3.1 Approaches and implementations

Within the basic scenario for XML query processing, we distinguish two types of implementations (aka engines), main-memory and persistent storage. The main-memory implementations load the XML data from secondary memory into main memory, process the data in an internal format (usually a tree), and then perform the XML queries over this format. Among main-memory implementations are for XPath 1.0, XMLElements [Gottlob et al., 2005, Koch, 2004, Xalan, 2002], and for XQuery, Galax [Fernández et al., 2006], Saxon [Kay, 2009], Qizx/Open Xaya Software [2009]. These implementations are convenient query tools for lightweight application scenarios, such as XML editors.

When we are dealing with large amounts of XML data, however, the main-memory approach becomes infeasible. Besides the problem of efficient querying of data, other problems, such as data manipulation (updates), transaction management, and security issues, have suggested the use of database technology for processing XML data. Motivated by this, in recent years, significant effort has been devoted to developing high-performance XML Database Management Systems (XML DBMSs). Database systems use a persistent storage implementation to query processing that is characterized by the fact that the XML data is pre-processed and diverse structural and value indices are built and stored in secondary memory for the purpose of improving query processing performance.

Further, the persistent storage engines fall into two classes: the native approach and the relational approach. The native approach is characterized by the fact that specialized storage and query processing systems tailored for XML data are developed from scratch. Native XML DBMSs for XPath 1.0 include Natix [Fiebig et al., 2002]; and for XQuery, Timber [Jagadish et al., 2002], eXist [Meier, 2006], X-Hive [X-Hive/DB, 2005], IBM DB2 Viper [Nicola and van der Linden, 2005], and Berkley XML DB [Berkley XML DB, 2009].

The relational approach directly utilizes existing relational database systems to store and query XML data. Among the relational-based XML DBMSs for XQuery are: MonetDB/XQuery [Boncz et al, 2006a,b], IBM System RX [Beyer et al., 2005], Microsoft SQL Server 2005 [Pal et al., 2005], Oracle DB [Murthy et al., 2005].

A detailed survey of different approaches to XML query processing can be found in Krishnamurthy et al., 2003 and Gou and Chirkova, 2007.

Throughout the thesis, we use a selection of the implementations presented
above to validate our research: in Chapters 3, 6, and 7 we use Galax, Saxon, Qizx/Open, and MonetDB/XQuery to analyze a set of existing XQuery benchmarks and validate performance benchmarks and methodology developed by us; and in Chapter 5 we propose a tool that automates the execution of performance benchmarks and that comes with adapters for a large set of implementations. In Chapter 8 we briefly present MonetDB/XQuery and its approach to query processing, as well as a new optimization technique for a particular type of recursive queries implemented on top of it. Further, we evaluate the proposed technique in comparison with Saxon. Our choice of engines is based on availability and ease of use, and it covers both main-memory and persistent storage implementations.

2.4 Performance evaluation of XML query processors

As pointed out in the introduction, performance is a key criterion in the design and use of XML query processing systems. Users, system administrators, and developers of query processing techniques and engines are all interested in performance evaluation since their goal is to obtain or provide the highest performance in a given setting. Performance evaluation helps determine how well a system is performing (possibly in comparison with alternative systems), whether any improvements need to be made and to which (bottleneck) components of the system, what the optimal values of the system’s parameters are, how well the system will perform in the future, etc. This makes performance evaluation a key tool to achieving good performance.

In [Jain, 1991], the author lists three general techniques for performance evaluation: analytical modeling, simulation, and measurement. The main consideration in selecting an evaluation technique is the life-cycle stage in which the system is. Measurements are preferred when a system already exists and evaluation is done for optimization purposes, while benchmarking is a measurement technique that has a well established and acknowledged role in the development of DBMSs [Jain, 1991]. Part I of this thesis is dedicated to analyzing, facilitating the use of, and developing benchmarks for performance evaluation of XML query processors.

2.4.1 Benchmarking

In performance studies, the term benchmark is used in many meanings. One common use of the term is synonymous to workload. Another use of the term refers to the performance measurements of a system against a standard workload, often presented relative to other systems. In this thesis, a benchmark refers to a set of performance measures defined on a workload. Often, benchmarks also contain detailed rules on how to apply the measures and obtain the benchmark
measurements, and on how to interpret the results. The process of performance evaluation of a system by measurements of a benchmark is called benchmarking.

The term test workload denotes any workload used in performance evaluation. Usually, a workload is composed of a data set and a set of operations over it. A test workload can be real or synthetic. A real workload is one observed on a system being used in normal conditions. A synthetic workload is developed and used for performance evaluation, it has characteristics similar to the ones of the real workload under study, and it can be applied repeatedly and in a controlled manner. The main reason for using a synthetic workload is that it models a real workload when no real-world data is available. Other reasons are that the workload can easily be modified in a controlled manner and that it might have built-in performance measuring capabilities [Jain, 1991].

The workload is the crucial part of a benchmark. It is possible to reach misleading conclusions if the workload is not properly selected. A workload should be representative of the tested scenario. In most cases, the evaluation targets a component of the system rather than entire system under test. In such cases, the workload should be focused by exercising the component under study while reducing the influence of other components or external factors. Another necessary condition for workloads is reproducibility. The workloads should be such that the results can be easily reproduced without too much variance.

The term System Under Test (SUT) refers to the complete set of components that define the system whose performance we want to assess and improve. Sometimes there is one specific component in the SUT whose performance is being considered. This component is called the Component Under Study (CUS). For example, a researcher wants to understand the impact of a new storage index on the performance of an XML DBMS system. In this case, the DBMS system is the SUT and the new index is the CUS.

A performance measure is a function defined on (subsets of) a test workload. The result of a performance measure on a particular input is called performance measurement. For example, the CPU time and the elapsed time measured for each operation in the workload are performance measures. The CPU time and the elapsed time, in this case, are called units of measure.

With respect to their scope, benchmarks fall into two categories: application benchmarks and micro-benchmarks. Application benchmarks focus on performance assessment, while micro-benchmarks are tools for explaining performance.

Application benchmarks

Application benchmarks test the overall performance of a system in a real-life application scenario, for example banking, airline reservations, etc. The workload consists of a data set and operations over it that are representative of the application under test. A typical workload contains a limited set of simple and complex operations that mimic the operation load in a user scenario. The benchmark
measures target the end-to-end performance of a system.

Application benchmarks are useful tools both for system developers and users. They are not suitable for testing different system components in isolation, since the workload and the measures are designed to cover system functionalities as a whole.

The Transaction Processing Performance Council (TPC) has a long history of developing and publishing benchmarks for testing relational database systems. For example, the TPC has proposed benchmarks for Online Transaction Processing applications (TPC-C and TPC-E), for Decision Support applications (TPC-H), and for an Application Server setting (TPC-App).

In the domain of XML databases and query processing systems, five application benchmarks have been proposed: XMach-1 [Böhm and Rahm 2001], XMark [Schmidt et al. 2002], X007 [Bressan et al. 2001b], XBench [Yao et al. 2004], and TPoX [Nicola et al. 2007]. The first four benchmarks are developed in academia. The last benchmark, TPoX (Transaction Processing over XML), was developed jointly by IBM and Intel. It simulates a financial application in a multi-user environment with concurrent access to the XML data. TPoX targets the performance of a relational-based XML storage and processing system.

In Chapter 3, we discuss and analyze the first four XQuery benchmarks and their properties in detail. TPoX is not included in this discussion and analysis, since it was published after this work had been conducted.

Micro-benchmarks

Micro-benchmarks test the performance of individual components of a system on particular system operations or functionalities. For example, a micro-benchmark might target the performance of an optimization technique for recursive queries. The workload and the measures are designed to allow for testing the targeted system component and operation in isolation. Usually, the performance measures are parametrized to allow for a systematic evaluation of the benchmark target with respect to interesting parameters.

Unlike application benchmarks, micro-benchmarks do not directly help in determining the best performing system in a particular application scenario. Rather, they help assessing the performance of particular components of a system and are most useful to system developers and researchers.

In the domain of databases, micro-benchmarks were first introduced for object-oriented databases [Carey et al. 1993]. The first benchmark with micro-benchmark features targeting XML databases is the Michigan benchmark (MBench) [Runapongsa et al. 2002].

In Chapter 3, we analyze the MBench benchmark and its features. In Chapter 6, we propose and discuss a repository of micro-benchmarks for testing XML query processing techniques and engines. And in Chapter 7, we propose a micro-benchmark for testing processing techniques for value-based join expressed in
XQuery.

Having completed an outline of the core concepts and terminology used in the thesis, we get to work on benchmarking methodology and tools next.