Querying XML: Benchmarks and Recursion

Loredana Afanasiev

Querying XML: Benchmarks and Recursion

Academisch Proefschrift

ter verkrijging van de graad van doctor aan de
Universiteit van Amsterdam
op gezag van de Rector Magnificus
prof. dr. D. C. van den Boom
ten overstaan van een door het college voor
promoties ingestelde commissie, in het openbaar
te verdedigen in de Agnietenkapel
op vrijdag 18 december 2009, te 16.00 uur

door

Loredana Afanasiev

geboren te Drochia, Moldavië.

Promotor: Prof. dr. M. de Rijke
Co-promotor: Dr. M. J. Marx

Promotiecommissie: Prof. dr. M. L. Kersten
Prof. dr. T. Grust
Dr. I. Manolescu

Faculteit der Natuurwetenschappen, Wiskunde en Informatica

The investigations were supported by the Netherlands Organization for Scientific Research (NWO) under project number 017.001.190.

SIKS Dissertation Series No. 2009–46
The research reported in this thesis has been carried out under the auspices of SIKS, the Dutch Research School for Information and Knowledge Systems.

Cover design by Loredana Afanasiev
Printed and bound by Ipskamp Drukkers

ISBN: 978–90–9024872–1

pentru părinții mei,

Ecaterina și Vladimir,

cărora le datorez tot ce sînt

Acknowledgments

I am most thankful to the Netherlands Organisation for Scientific Research (NWO) for funding my PhD research through the Mozaïek program. I am thankful to NWO and the School for Information and Knowledge Systems (SIKS) for organizing educational workshops and courses that contributed to the improvement of my communication and research skills.

The development of this thesis was continuously guided and, at times, gently pushed by my supervisor, Maarten Marx, and my promotor, Maarten de Rijke.

There are not enough words to describe how grateful I am to Maarten Marx for the help and the effort he invested in me. Maarten played key roles in shaping not only this thesis, but also my life in the last six years. To give a few examples, Maarten led me to: my master thesis, my first scientific article, two jobs as scientific programmer at University of Amsterdam, this thesis, my husband, my internship at Google Inc., ... Maarten helped me stay in the Netherlands to finish my master thesis and bought my first pair of glasses when I could not afford them. His eagerness to help and impact are tremendous.

Though very different in character, Maarten de Rijke excels in generosity too. I will always remember Maarten for his kindness, wisdom, and professionalism. I am eternally grateful for his positive feedback when I most needed it. His superhuman powers of speed proof-reading drafts and giving invaluable comments at the nick of time were essential both for the writing of the NWO project that funded this research and for finishing this thesis.

I am very thankful to Ioana Manolescu for leading me to most of the work presented in the first part of the thesis, and to Torsten Grust, Jan Rittinger, and Jens Teubner (the Pathfinder team) for their contribution to the work presented in the second part of the thesis. I also thank Massimo Franceschet for his fruitful ideas and guidance through the first stages of my PhD.

The research presented in this thesis is based on close collaboration with the following outstanding researchers: Balder ten Cate, Massimo Franceschet, Torsten Grust, Maarten Marx, Stefan Manegold, Ioana Manolescu, Philippe

Michiels, Maarten de Rijke, Jan Rittinger, Jens Teubner, and Enrico Zimuel. I was very fortunate to work with and learn from them.

I remember with high regards Alon Halevy, Jayant Madhavan, and David Ko (the Scuba team) for the exciting time during my internship, and the MonetDB/XQuery team for their always timely support with the engine.

I am grateful to Torsten Grust, Ioana Manolescu, and Martin Kersten for serving in my committee.

Big thanks to Balder, Katja Hofmann, Edgar Meij, and Wouter Weerkamp for proof-reading parts of my thesis, and to Valentin Jijkoun and Katja for agreeing to be my paranymphs.

The Information and Language Processing Systems (ILPS) group was my adopted family for the last six years. Over the years the group welcomed new members and parted with some, but it always retained its magical unity and warmth.

Thanks to my colleagues and friends for my daily required portion of interaction and fun: David Ahn, Krisztian Balog, Wouter Bolsterlee, Marc Bron, Simon Carter, Breyten Ernsting, Gaëlle Fontaine, Amélie Gheerbrant, Jiyin He, Katja, Bouke Huurnink, Valentin, Sofiya Katrenko, Martha Larson, Edgar, Gilad Mishne, Börkur Sigurbjörnsson, Erik Tjong Kim Sang, Manos Tsagias, Wouter, Shimon Whiteson, and Victoria Yanulevskaya.

Special thanks to my dear friends: David, for tutoring me in English, for infecting me with the joy of jogging, for factuality and eloquence, for his endless energy; Katja, for most interesting discussions, for positivity and action-driven personality, for inspiration; Valentin and Victoria, for their warmth, for sharing the nostalgic moments of our cultural past; Govert van Drimmelen, for persistence and encouragement; Valeriu Savcenco, for the good and bad times and for remaining a truthful friend.

I am grateful to Olle and Ineke ten Cate, for their continuous and unconditional support. I am infinitely grateful to Ineke for her care and love. Ineke nurtured me through the most difficult times of writing.

The ultimate support I got from my better half, my love and husband, Balder. Balder inspired me to my best times and carried me through my worst times.

Îi mulțumesc din sufletul surorii mele, Angela, și familiei ei, Sergiu, Felicia, și David, pentru toată veselia și căldura pe care mi-o aduc. Deși suntem departe una de alta, căldura ta mă ajută să trec prin momentele grele.

Tot binele pe care l-am făcut sau îl voi face vreodată îl dedic părinților mei. Dragii mei părinți, dragostea și susținerea voastră stau la baza indrăznelei mele de a merge atît de departe pe calea academică și a vieții. Vă mulțumesc din suflet pentru viața frumoasă pe care mi-ați dat-o.

October 26, 2009
Amsterdam

Contents

Chapter 1

Introduction

The Extensible Markup Language (XML) [World Wide Web Consortium, 2008] is a textual format for representing data. Since its introduction in 1998 as a World Wide Web Consortium (W3C) recommendation, it has become a widely accepted standard for storage and exchange of data. XML was specifically designed to be able to handle semi-structured data: data that is not sufficiently structured, or whose structure is not sufficiently stable, for convenient representation in a traditional relational database.

The increasing amount of data available in the XML format raises a great demand to store, process, and query XML data in an effective and efficient manner. To address this demand, the W3C has developed various accompanying languages that facilitate, e.g., typing, querying, and transformation of XML data.

A large part of this thesis is devoted to XQuery, the XML Query language [World Wide Web Consortium, 2007]. XQuery has been developed since 2000 and has been published as a W3C recommendation in 2007. Some of the main ingredients of the XQuery language are XML document navigation, data typing, filtering, joining, ordering, new XML document construction, and recursive user-defined functions. The fragment of XQuery that is concerned with XML document navigation is known as XPath. XPath is an important language that was developed independently of XQuery by the W3C and proposed as a recommendation in 1999 [World Wide Web Consortium, 1999a]. It lies at the heart of XQuery as well as of other XML-related technologies such as Extensible Stylesheet Language Transformations (XSLT) and XML Schema.

In this thesis, we are concerned with the *query processing* problem for XQuery: given an XML document or collection of documents, and a query, to compute the answer to the query.[1] This is a difficult problem. A long line of research addresses the problem of efficient query processing. One strategy is to employ query optimization. *Query optimization* refers to deciding which query processing

[1] This problem is also known as query evaluation. In this thesis, though, we reserve the term "evaluation" for the empirical evaluation of systems or techniques.

technique to use at run time, in order to gain efficiency.

Our research falls within two topics: (*i*) developing optimization techniques for XQuery, and (*ii*) performance evaluation for XQuery processing. These topics are tightly connected, since performance evaluation is aimed at measuring the success of optimization techniques. More specifically, we pursue *optimization techniques for recursion* in XQuery, on the one hand, and *benchmarking as a performance evaluation technique*, on the other hand.

Recursion in XQuery

The backbone of the XML data model, namely *ordered, unranked trees*, is inherently recursive and it is natural to equip the associated query languages with constructs that can query such recursive structures. XQuery provides two mechanisms for expressing recursive queries: *recursive axes*, such as `descendant` and `ancestor` [World Wide Web Consortium, 1999a] and *recursive user-defined functions*. The recursive axes of XPath provide a very restricted form of recursion, while recursive user-defined functions in XQuery allow for arbitrary types of recursion. Indeed, they are the key ingredient of the Turing completeness of the language [Kepser, 2004]. Thus, in terms of expressive power, the designers of the language took a giant leap from recursive axes to recursive user-defined functions.

Recursive user-defined functions are *procedural* in nature, they describe *how* to get the answer rather than *what* the answer is. Language constructs that describe the query answer are called *declarative* [Brundage, 2004]. Procedural constructs put the burden of optimization on the user's shoulders. Recursive user-defined functions largely evade automatic optimization approaches beyond improvements like tail-recursion elimination or unfolding [Park *et al.*, 2002, Grinev and Lizorkin, 2004].

In this thesis, we investigate the possibility of adding a declarative recursion operator to the XQuery language. While less expressive than recursive user-defined functions, the declarative operation puts the query processor in control of query optimizations.

Benchmarking

Performance is a key criterion in the design and use of XML processing systems. Performance evaluation helps to determine how well a system performs (possibly in comparison with alternative systems), whether any improvements need to be made and, if so, to which (bottleneck) components of the system, what the optimal values of the system's parameters are, and how well the system will perform in the future, etc. This makes performance evaluation a key tool to achieving good performance.

Benchmarking is a performance measurement technique that has a well established and acknowledged role in the development of DBMSs [Jain, 1991]. A

benchmark refers to a set of performance measures defined on a workload consisting of XML data and queries. The process of performance evaluation of a system by measurements of a benchmark is called benchmarking.

In this thesis, we investigate existing XQuery benchmarks and develop new ones. Our focus is on developing so-called micro-benchmarks and related methodology. *Micro-benchmarks* are narrowly-scoped benchmarks; they test the performance of individual components of a system on particular system operations or functionalities. We also develop software tools for facilitating the process of benchmarking as well as test suites and are concerned with how to ensure the repeatability of experimental evaluations.

1.1 Research questions

Our research has two main goals: one is to develop benchmarking methodology and tools for performance evaluation of XML query engines, the other is to analyze and develop optimization techniques for declarative recursion operators for XML query languages. Correspondingly, the research questions that we address fall into two groups.

Developing benchmarking methodology and tools

The general research question of the first part of the thesis is: *How to evaluate the performance of XML query processing?* To answer this question, we first investigate the tools previously proposed in the literature, namely five benchmarks for performance evaluation of XQuery engines. The following questions are driving our investigation:

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

Question 3.2: How are existing benchmarks used?

Question 3.3: What can we learn from using the five benchmarks?

As a result of our investigation, we learn that there is a need for performance tools and methodology for precise and comprehensive experimental studies and that there are no tools to meet this need. Hence, the question we address next is:

Question 6.1: What is a suitable methodology for precise and comprehensive performance evaluations of XML query processing techniques and systems?

Our answer to this question involves micro-benchmarks and a methodology for micro-benchmarking. To illustrate, and put to work, the methodology we propose, we undertake a concrete performance evaluation task driven by the following question:

Question 7.1: How to measure the performance of value-based joins expressed in XQuery? What is a suitable measure and which parameters are important to consider?

Besides building benchmarking tools and methodology, there are two other main issues that need to be addressed when pursuing performance evaluation. The following questions address these issues:

Question 4.1: How to ensure the repeatability of experimental studies of database systems? This question incorporates two sub-questions: What is a proper methodology for designing and reporting on experimental studies that facilitates their repeatability? and What is a proper mechanism for evaluating the repeatability of experimental studies presented in scientific research?

Question 5.1: Is it possible to build a generic tool for automating the following three tasks: (*i*) running a performance benchmark, (*ii*) documenting the benchmark experiment, and (*iii*) analyzing the benchmark results? What are the design choices that need to be made?

Inflationary fixed points in XQuery

Having focused on developing benchmarking methodology in the first part of the thesis, we pursue the following general question in the second part of the thesis: *How to obtain declarative means of recursion in XQuery?* This boils down to a more concrete question:

Question 8.1: What is a suitable declarative recursive operator in XQuery that is rich enough to cover interesting cases of recursion query needs and that allows for (algebraic) automatic optimizations?

As an answer to this question we consider an *inflationary fixed point* (IFP) operator added to XQuery and investigate an efficient evaluation technique for it. The next question addresses the theoretical properties of this operator in the setting of XQuery.

Question 9.1: How feasible it is to do static analysis for recursive queries specified by means of the fixed point operator? Specifically, are there substantial fragments of XQuery with the fixed point operator for which static analysis tasks such as satisfiability are decidable?

The connection between our two main concerns in this thesis—performance evaluation and optimization—is established in Chapter 8 where we use the benchmarking tools developed in Part I to evaluate the performance of the optimizations proposed in Part II.

1.2 Main contributions

Following the research questions listed above, our contributions fall within the two topics of the thesis. The main contributions of Part I relate to developing tools for performance evaluation of XML query processors. The main contributions of Part II relate to analyzing and developing optimization techniques for a declarative recursion operator in XML query languages. A detailed list of contributions follows.

Developing benchmarking methodology and tools

1. A survey of experimental evaluations of XML query engines published by the database scientific community; a survey of XQuery benchmarks; an analysis of XQuery benchmarks with the focus on the benchmark queries and measures; a corrected and standardized list of benchmark queries.

2. A micro-benchmarking methodology for systematic, precise, and comprehensive performance analysis of XML query engines; a framework for collecting and storing micro-benchmarks. Both the methodology and the framework collectively go under the name of *MemBeR*.

3. A micro-benchmark for testing value-based joins expressed in XQuery; an analysis of four XQuery engines on value-based joins based on this micro-benchmark.

4. An attempt at defining a standard for reporting experimental studies with the goal of assuring their repeatability; a report on the review of repeatability of experimental studies conducted for the research articles submitted to SIGMOD 2008.

5. A tool for automatic execution of benchmarks on several XML query engines, called *XCheck*. XCheck includes statistical and visualization tools that help the performance analysis.

Inflationary fixed points in XQuery

1. An analysis of an inflationary fixed point operator for expressing recursion in XQuery; a distributivity property of IFP expressions that allows for efficient query processing; implementation of the IFP operator into an XQuery engine (MonetDB/XQuery); a performance evaluation of our approach.

2. An analysis of theoretical properties of the logical core of the XPath language (Core XPath) enhanced with an inflationary fixed point operator; a proof of the undecidability of this language; a proof of the fact that this language is strictly more expressive than the language obtained by adding a Transitive Closure operator to Core XPath.

1.3 Organization of the thesis

This thesis is organized in two parts, each focused on a different general research question. Before these parts begin, in Chapter 2, we set the background and introduce the terminology used throughout the thesis. After these two parts end, we conclude the thesis in Chapter 10. Figure 1.1 suggests reading paths through the thesis.

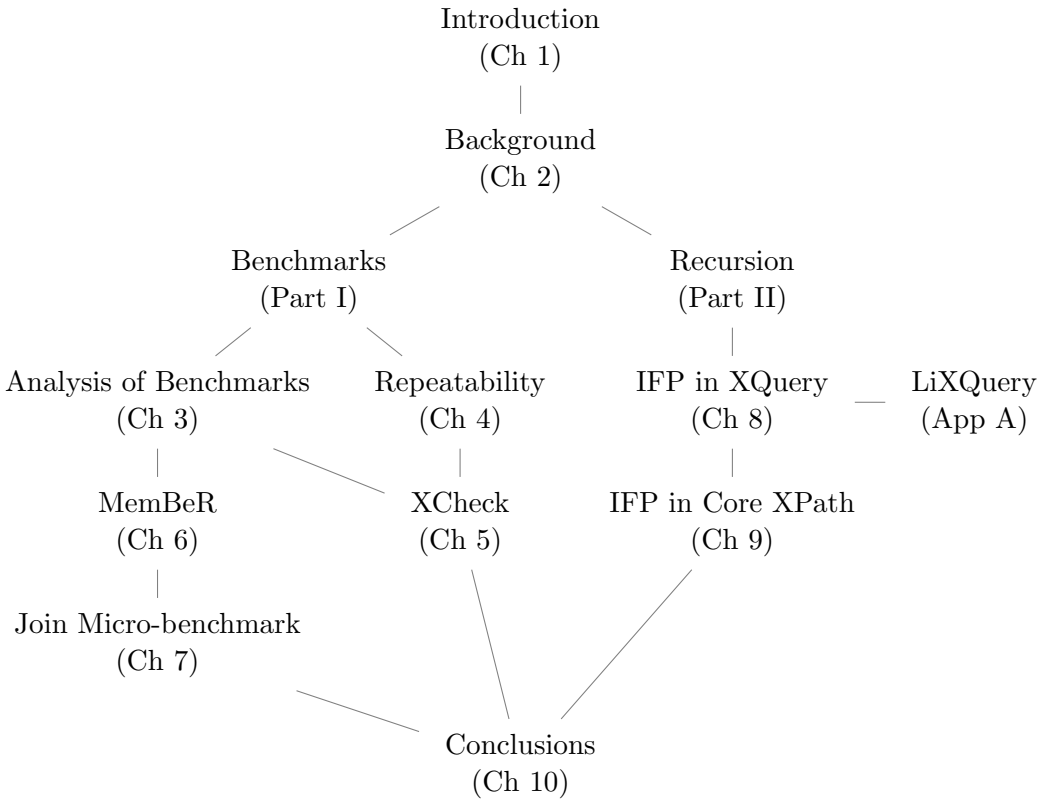


Figure 1.1: Suggested reading paths.

Part I (Benchmarks) is composed of five chapters addressing the research questions dealing with performance evaluation of XML query processing. In Chapter 3, we analyze existing XQuery benchmarks and survey their usage. This chapter serves as a detail discussion of related work for the whole part. In Chapter 4, we discuss how to achieve the repeatability of experimental evaluations of computer systems in the database domain. As part of setting up a methodology for repeatability, we perform a review of articles submitted to the research conference SIGMOD 2008 and measure the repeatability of the presented experimental evaluations. In Chapter 5, we discuss the problems and challenges of automating the execution of performance evaluation benchmarks on many XML query engines and comparison of their performance. We present a software tool, XCheck, as a

solution to these problems. In Chapter 6, we present a methodology for micro-benchmarking XML query engines, which we refer to as MemBeR. MemBeR also comprises a framework for collecting and storing micro-benchmarks. Finally, in Chapter 7, we present a MemBeR-style micro-benchmark for testing performance of value-based joins expressed in XQuery. We validate the micro-benchmark by analyzing the performance of four XQuery engines.

Part II (Recursion) is composed of two chapters addressing the research questions referring to recursion in XQuery. In Chapter 8, we consider adding an inflationary fixed point operator to XQuery. We develop an optimization technique for processing this operator. Further, we implement this technique on top of MonetDB/XQuery, and evaluate its performance. In Chapter 9, we study the theoretical properties, decidability and expressivity, of this inflationary fixed point operator in the setting of Core XPath, the XML tree navigational core of XPath and XQuery.

The work presented in Chapter 8 relies on a logical fragment of XQuery called LiXQuery. For a quick reference to the language constructs, we present the definition of the syntax of LiXQuery in Appendix A.

1.4 Publications underlying the thesis

The material in this thesis builds on a number of previously published papers that we list below. Full details of these publications can be found in the Bibliography.

Part I builds on work presented in:

1. "An analysis of XQuery benchmarks" [Afanasiev and Marx, 2006] and [Afanasiev and Marx, 2008];
2. "The repeatability experiment of SIGMOD 2008" [Manolescu *et al.*, 2008a];
3. "XCheck: a Platform for Benchmarking XQuery Engines" [Afanasiev *et al.*, 2006]; and
4. "MemBeR: a micro-benchmark repository for XQuery" [Afanasiev *et al.*, 2005a].

Part II builds on work presented in:

1. "On Core XPath with Inflationary Fixed Points" [Afanasiev and ten Cate, 2009]; a journal version of this article is currently in preparation;
2. "Recursion in XQuery: Put Your Distributivity Safety Belt On" [Afanasiev *et al.*, 2009];
3. "An Inflationary Fixed Point Operator in XQuery" [Afanasiev *et al.*, 2008];
4. "Lekker bomen" [Afanasiev *et al.*, 2007]; and
5. "PDL for Ordered Trees" [Afanasiev *et al.*, 2005b].

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 to 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 schema 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

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

```
<!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>
```

Figure 2.2: Example DTD describing university curriculum data.

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-

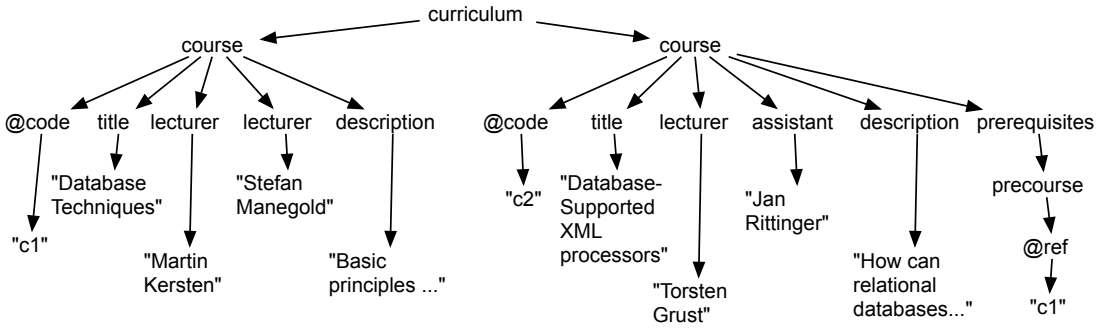


Figure 2.3: The node-labeled ordered tree corresponding to the university curriculum document.

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

Various tree data models were proposed for XML. W3C alone recommends a few different tree models for different purposes: the Document Object Model (DOM) [World Wide Web Consortium, 1998], the XML Information Set (Infoset) [World Wide Web Consortium, 2004a], Post-Schema-Validation Infoset (PSVI) [World Wide Web Consortium, 2004b], XPath 1.0 [World Wide Web Consortium, 1999a], and XPath 2.0/XQuery 1.0 Data Model (XDM) [World Wide Web Consortium, 2007a]. 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 [Gou and Chirkova, 2007]. 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"`. 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.

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

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

```
declare namespace my='my-functions';
declare function my:descendants($n as node()*) as node()* {
  if (empty($n))
  then ()
  else ($n/*, my:descendants($n/*))
};
```

Figure 2.5: Recursively computing the descendants of a sequence of nodes.

expressions. The query given in Figure 2.4 uses the XML based syntax.

Besides providing a library of built-in functions, XQuery allows for *(recursive) user-defined functions ((R)UDFs)*. Users can write custom functions containing any number of typed parameters and any XQuery expression in the function body, including recursive function calls. Function parameters can be of any type allowed by the XQuery data model. They have distinct names and are in scope for the entire function body.

Recursion is commonly used when processing XML, due to its tree-like structure. XQuery provides two recursive functionalities: (*i*) recursive axes, such as `descendant` and `ancestor`; and (*ii*) recursive user-defined functions. RUDFs in XQuery admit arbitrary types of recursion, which makes recursion in the language procedural by nature. Figure 2.5 contains a recursive user-defined function that computes the descendants of a given sequence of nodes. Note that UDFs need to be defined in a custom namespace, using the `namespace` definition clause.

In Chapter 8, we argue for the usefulness of declarative recursive operators as opposed to procedural recursion provided by RUDFs. We propose adding to XQuery a declarative operator, namely an inflationary fixed point operator, and present an efficient processing technique for it. In Chapter 9, we discuss the theoretical properties of this operator when added to the core navigational fragment of XPath.

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, XMLTaskForce [Gottlob *et al.*, 2005, Koch, 2004], Xalan [Xalan, 2002], and for XQuery, Galax [Fernández *et al.*, 2006], Saxon [Kay, 2009], Qizx/Open [Axyana 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 [Berkeley 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) [TPC, 2009] 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 [Böhme 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.

Part I

Benchmarks

In this part of the thesis, we address the research questions dealing with performance evaluation of XML query processing. In Chapter 3, we analyze existing XQuery benchmarks and survey their usage. This chapter serves as a detailed discussion of related work for the whole part. In Chapter 4, we discuss how to achieve the repeatability of experimental evaluations of computer systems in the database domain. As part of setting up a methodology for repeatability, we perform a review of articles submitted to the SIGMOD 2008 research conference and measure the repeatability of the presented experimental evaluations. In Chapter 5, we discuss the problems and challenges of automating the execution of performance evaluation benchmarks on many XML query engines and on comparing their performance. We present a software tool, XCheck, as a solution to these problems. In Chapter 6, we present a methodology for micro-benchmarking XML query engines, which we refer to as MemBeR. MemBeR also comprises a framework for collecting and storing micro-benchmarks. Finally, in Chapter 7, we present a MemBeR-style micro-benchmark for testing performance of value-based joins expressed in XQuery. We validate the micro-benchmark by analyzing the performance of four XQuery engines.

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].[1] 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.

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. XMarch-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 Mathematics 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 introduced by the inclusion of textual descriptions consisting of mixed-content elements. The benchmark document scales by varying a scaling factor in a continuous 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 specified. 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;

(*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,a] 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 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.[2] 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

[2]Bressan *et al.* [2001b] present 18 queries, while the benchmark website `http://www.comp.nus.edu.sg/~ebh/XOO7.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, MBench, 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.

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 scaleable.

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 100MB.

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] `http:www.ibiblio.org/webster`
[4] `http://www.oed.com/`
[5] `http://about.reuters.com/researchandstandards/corpus/`
[6] `www.springerlink.com/`
[7] `http://www.tpc.org/tpcw/`

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-

	XMach-1 (2001)	**XMark (2001)**	**X007 (2001)**	**MBench (2002)**
targeted SUT	a web application	DBMS, query processor	DBMS, query processor	DBMS, query processor
targeted CUS	DBMS	query processor	query processor	query processor
benchmark type	appl. benchmark	appl. benchmark	appl. benchmark	micro-benchmark
# users	multi-user	single-user	single-user	single-user
performance measure	query throughput (Xqps)	query execution time (sec)	query execution time (sec)	query execution time (sec)
# queries	8	20	23	49
query language	NL, XQuery	NL, Kweelt	NL, XQuery	NL, XQuery, XPath, SQL
# update operations	3	–	–	7
language	NL	–	–	NL
main data type	text-centric	data-centric	data-centric	data-centric
# schemas	ca. #docs/20 DTDs	1 DTD	1 DTD	1 XML Schema
# element types	4∗#schemas+7	76	9	2
# recursive types	2	–	1	1
# mixed-content types	1	4	1	1
# ID/IDREF types	–	4 ID, 9 IDREF	–	1 ID, 1 IDREF
# data value types	2 (string, datetime)	4 (string, integer, float, date)	4 (string, integer, year)	2 (xs:string, xs:integer)
tree depth	11, 11, 14	11	8, 8, 10	16
avg depth	3.8	4.6	7.9, 7.9, 9.9	14.5
tree fan-out	38, 46, 100	2550–2550000	61, 601, 601	13, 39, 111
avg fan-out (no leaves)	3.6	3.7	3	2
# documents	100, 1000, 10000, etc.	1	1	1
document size	ca. 16KB	11MB–11GB	4.5MB–400MB	46MB, 496MB, 4.8GB
total dataset size	2.3MB, 23MB, 219MB, etc. (#docs∗16 KB)	11MB–11GB	4.5MB–400MB	46MB, 496MB, 4.8GB
# elements/KB	10	ca. 14	ca. 9.7	1.4

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.

	XBench (2002)			
	TC/SD	**TC/MD**	**DC/SD**	**DC/MD**
targeted SUT	DBMS, query processor			
targeted CUS	query processor			
benchmark type	application benchmark			
performance measure	query execution time (sec)			
# users	single-user			
# queries	17	19	16	15
query language	NL, XQuery			
# update operations	–			
language	–			
main data type	text-centric	text-centric	data-centric	data-centric
# schemas	1 XML Schema, DTD	1 XML Schema, DTD	1 XML Schema, DTD	6 XML Schemas and DTDs
# element types	24	26	50	82
# recursive types	–	1	–	–
# mixed-content types	2	–	1	–
# ID/IDREF types	1 ID, 1 IDREF	–	1 ID, 1 IDREF	–
# data value types	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)
tree depth	7	6	7	3
avg depth	6	3.5, 3.9, 4.1	4.6	2
tree fan-out	733, 7333, 73333, 733333	75, 264, 2665	$250 * 10^n, n = 1, 2, 3, 4$	$576 * 10^n, n = 1, 2, 3, 4$
avg fan-out (no leaves)	4.3	10.7, 11.3, 20.7	3.1	8
# documents	1	26–26666	1	2592–2592000
document size	10MB–10GB	2KB–1500KB	10MB–10GB	1KB–3KB
total dataset size	10MB, 100MB, 1GB, 10GB			
# elements/KB	25.8	3.1, 3.5, 6	ca. 21	ca. 7.6

Table 3.2: Characteristics of XBench, a family of four benchmarks. As before, NL means natural language.

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 performe 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 characteritics. 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.

3.3.1 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 `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*.

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

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:

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:

```
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:

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

Table 3.4: Query language analysis of the benchmarks.

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.

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

Conference	# of papers	with standard benchmarks
VLDB	22	7
SIGMOD	9	4
ICDE	10	2
total	41	13 (31%)

Benchmark	# of papers	with derivation
XMark	11	5
XBench	2	0
XQTS (XMT)	2	0
total	13	5

Table 3.5: Benchmark usage survey statistics.

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`. 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.

Real life and synthetic data sets	# of papers
XMark	22
DBLP	8
PennTreebank	6
XBench	5
SwissProt	4
NASA	3
Protein	3
IMDB	2
Shakespeare [Bosak, 1999]	2
XMach-1	1
others	9
total	33

Query language	# of papers
XPath 1.0	25
XQuery	3
modified version of XQuery	1
SQL	1
unspecified	3
total	33

Table 3.6: XML data sets and query languages used in scientific articles for experimental evaluation of XML query processing engines and techniques.

In most cases, these experiments are based on existing (real life or synthetic) data sets and specifically designed queries (often parametrized). Among the most frequently used data sets are existing XML collections such as DBLP [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-complience 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

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

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

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 Qizx/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

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,[8] 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>
```

[8] http://ilps.science.uva.nl/Resources/MemBeR/other-benchmarks/queries.html

```
sortby (buildDate DESCENDING)
```

We rewrote it to XQuery as follows:

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

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

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:

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

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

As an example of the second kind of error, consider query Q6 of XBench DC/MD:

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

When applying the value comparison `gt`, the left operand is first atomized, then the untyped atomic operand is cast to `xs:string`. Since `xs:string` and `xs:decimal` (the type of the right operand) are incomparable types a type error is raised. To solve this problem, we could explicitly cast the left operand to `xs:decimal` or we could use the general comparison operator `>` that assures the conversion of the untyped operand to the numeric type of the other operand. We take the latter option.

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." [Runapongsa *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:

```
if(empty($e)) then ()
else
<average avgaSixtyFour="{
  avg(for $a in $e return $a/@aSixtyFour)
  }"
  aLevel="{$e[1]/@aLevel}">
 {my:one_level($e/eNest)}
</average>
```

The second error is a mismatch between structure of the resulting elements and the description of the result. When the first error is fixed, then the query yields a deep tree with one more level than there are levels in the input document. This is due to the fact that the recursive function call is nested in the result element construction. This does not conform with the query description, which talks about a shallow tree with a dummy root and as many children as levels in the input documents. This can be corrected in two ways: changing the syntax of the query to fit the description, or changing the description to fit the formal semantics of the query. The Michigan benchmark authors explicitly say that the natural language description is the normative query definition. We thus picked the first option. The corrected query is below.

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

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

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

We changed the queries of X007, XMark, MBench and XBench (TC/SD and DC/SD) to access their input data by invoking the `fn:doc()` function. The document URI is left out to be filled in at query execution. Most benchmarks test data scalability, so they run the same queries on different documents. Thus the input document(s) of a query is a *parameter* which should be filled in by the testing platform.

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

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

We then query this document to obtain the sequence of document nodes in the collection. We added the computation of this sequence as a preamble to each

query. The result is stored in a variable that is further used instead of the `fn:collection()` function call. So the query:

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

becomes:

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

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 [World Wide Web Consortium, 2007]. During static type checking the engine tries to assign a static type to the query and it raises a type error if it fails. In order to run the benchmarks on the engines that implement static type checking, we ensure that the benchmark queries do not raise static type errors.

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

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

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

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

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

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:

	data 1	data 2
XMach-1	19MB	179MB
X007	13MB	130MB
XMark	14MB	113MB
MBench	46MB	–
XBench TC/SD	10MB	105MB
XBench TC/MD	9.5MB	94MB
XBench DC/SD	11MB	104MB
XBench DC/MD	16MB	160MB

The 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`.

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

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

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

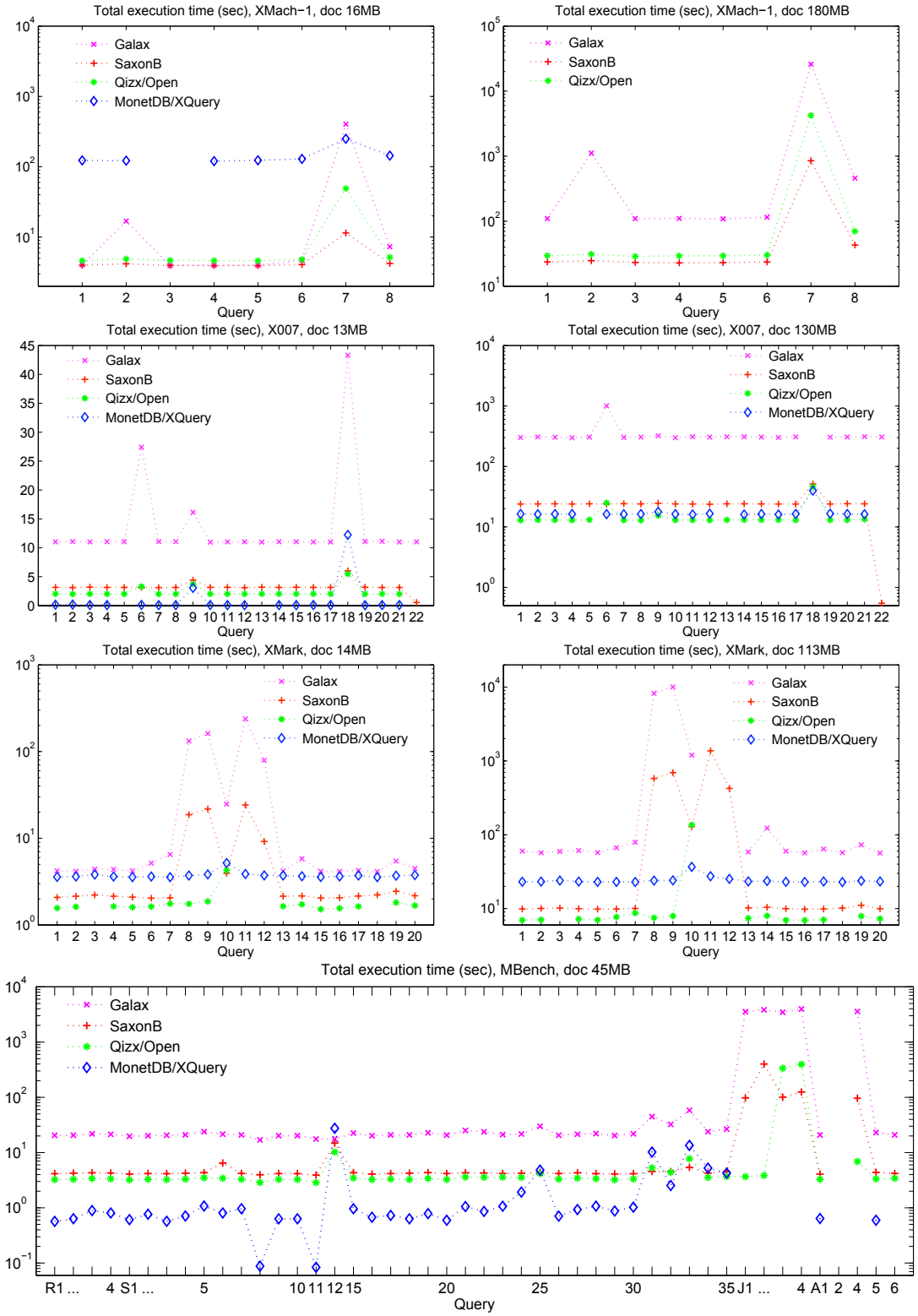


Figure 3.1: XMach-1, X007, XMark, and XBench on Galax, SaxonB, Qizx/Open, and MonetDB/XQuery.

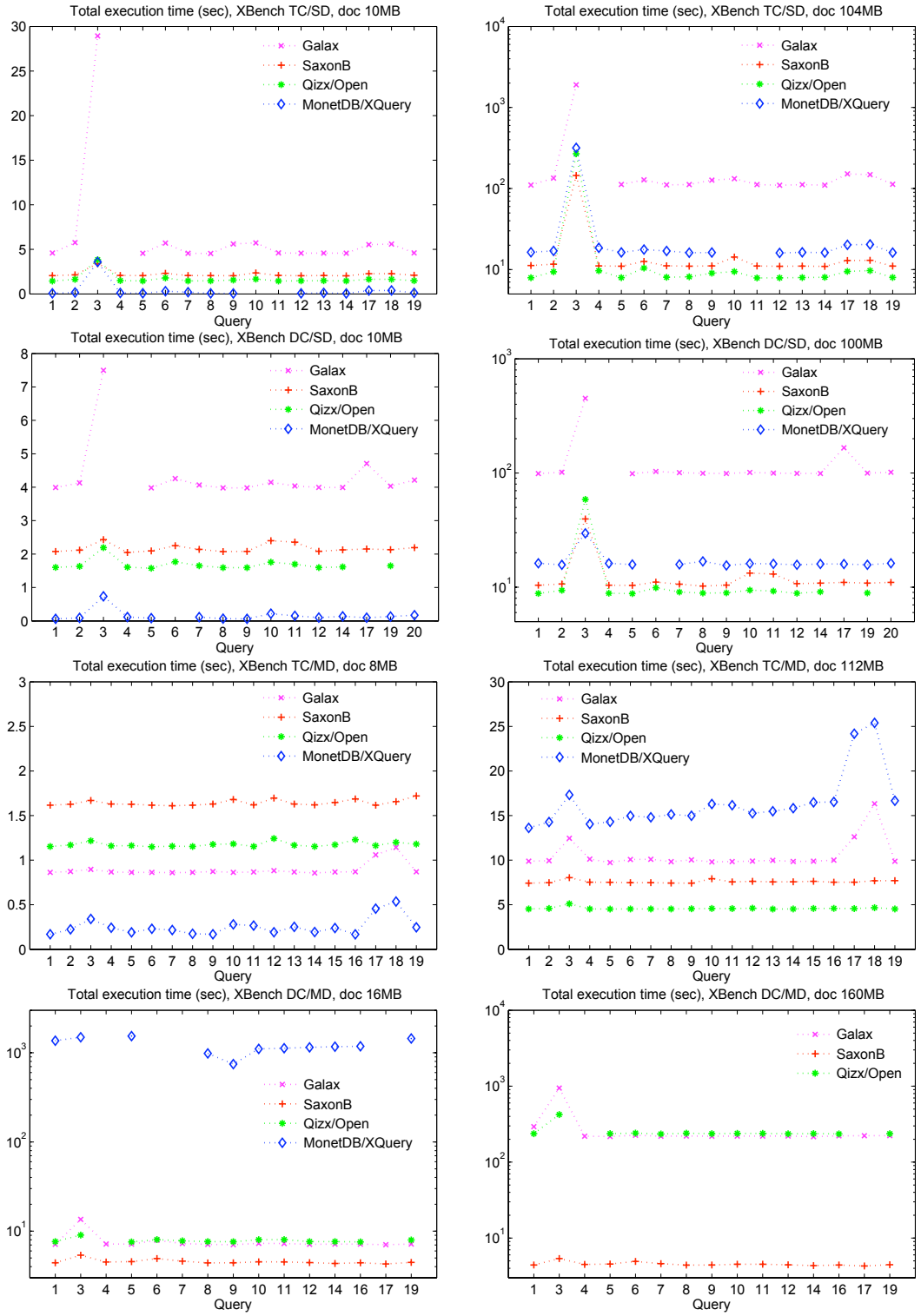


Figure 3.2: XBench TC/SD, DC/SD, TC/MD, and DC/MD on Galax, SaxonB, Qizx/Open, and MonetDB/XQuery.

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 `fn:zero-or-one()` function; the two errors of Galax are caused by the fact that it does not implement the `preceding` axis; the MonetDB/XQuery errors are diverse, for 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.

	Galax	MonetDB/XQuery
XMach-1	–	all queries on data 2
X007	Q18 on data 2	Q5
XMark	Q11, Q12 on data 2	–
MBench	–	QJ1, QJ2, QJ3 QA4
XBench DC/SD	–	Q6
XBench DC/MD	–	all queries on data 2

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 *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, Boncz *et al.* [2006a] 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.

3.6.4 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:

```
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>
```

Varying query parameters	**Expected results**
QJ1 ⇒ QJ2 **query selectivity**: 1.6% ⇒ 6.3%	query processing time grows
QJ3 ⇒ QJ4 **query selectivity**: 0.02% ⇒ 0.4%	query processing time grows
QJ1,QJ2 ⇒ QJ3,QJ4 **average query selectivity**: 3.95% ⇒ 0.21% **join type**: value-based ⇒ id/idref **syntactic form**: `where` form ⇒ `if` form	average query processing time decreases

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

QJ4 is obtained from QJ3 by replacing all the occurrences of the attribute name `aSixtyFour` with `aFour`. The query selectivity of QJ3 and QJ4 is approximately 0.4% and 0.02%.[9]

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

[9]Note that even though the selectivity of the subexpression `//eNest[@aFour=3]` is 16 times larger than the selectivity of the subexpression `//eNest[@aSixtyFour=3]`, the selectivity of QJ4 is 20 times larger than the selectivity of QJ3. The difference is due to the influence of the `eOccasional` elements on the join outcome. For more information about the `eOccasional` elements and their attribute values, see [Runapongsa *et al.*, 2002].

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

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

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 exception 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 compared 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 engines' 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 analysis 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.

Chapter 4

Repeatability of Experimental Studies

One important aspect of experimental studies is their repeatability. The aim of this chapter is to determine and improve the repeatability of the experimental studies in the database field. We report on a repeatability review that was conducted for the research articles submitted to the conference of the Special Interest Group On Management of Data (SIGMOD) of the Association For Computing Machinery (ACM) in 2008 [ACM, 2008].

From the repeatability review, we learn a number of lessons about how to describe experimental studies in order to ensure their repeatability; these lessons are implemented in XCheck, a software tool for executing XML query benchmarks. We present XCheck in Chapter 5.

This chapter is based on work previously published in [Manolescu *et al.*, 2008a].

4.1 Introduction

Repeatability is a property that allows experimental studies to be repeated and their results reproduced. This property is at the basis of the experimental sciences and it has two functions: (*i*) it guarantees the consistency of experimental results and thus, the soundness of the observations and conclusions drawn; and (*ii*) it makes experimental results available for comparison with other research, which facilitates scientific proliferation. A more precise definition of repeatability follows. For experimental studies conducted in computer science repeatability is equally important [Jain, 1991].

Performance of database management systems (DBMSs) is a key research topic in the database community, while experimental studies are the main performance evaluation method. Nevertheless, in the last decade or more, the field has suffered from a lack of standard methodology for conducting and reporting experimental studies [Manolescu and Manegold, 2007, 2008]. In particular, the

repeatability of experimental studies is not ensured. We believe that this affects the quality and progress of database research.

This leads us to the following question:

4.1. Question. *How to ensure the repeatability of experimental studies of database systems?* This question incorporates two sub-questions: (*i*) *what is a proper methodology for designing and reporting on experimental studies that facilitates their repeatability?* (*ii*) *what is a proper mechanism for evaluating the repeatability of experimental studies presented in scientific research?*

Before we present our attempt to answer these questions, we specify what repeatability means exactly. In database research, many experimental studies use measures that are dependent on the experimental environment. For example, performance evaluation measures, such as the execution time of a database system, are hardware dependent. Therefore, when we talk about repeatable experiments we cannot compare directly the performance measurements obtained in similar, but not identical, experimental environments. We define *repeatability* as the property of an experimental study that allows it to be repeated in a similar environment and ensures that the experimental results lead to the same observations and, ultimately, to the same conclusions. This definition implies that, depending on the given experimental study, assessing its repeatability might require human judgement and thus can not always be automated.

In order to achieve repeatability, obviously, the components of the experimental studies, such as the test data and the system under test, should be available or reproducible. There should also be a detailed description of the experimental setup and results. Further, reporting the environmental parameters that influence the experimental results completes the requirements for achieving repeatability. Assembling a standard list of such parameters might help to establish a common practice of reporting their values and impact. With respect to a quality control mechanism, we expect that it can be achieved via a peer-reviewing system, in the same style as the one used to ensure the quality of research papers. As mentioned earlier, requiring human judgment might be unavoidable in an unrestricted domain of experimental studies. The experimental data can be archived in internet-based repositories associated with scientific publication venues.

In this chapter, we report on an attempt at achieving and measuring the repeatability of experimental studies along the lines described above. The key aim of this study is understanding the challenges we face in achieving and promoting the property in database research. Towards this goal, a repeatability review process of the research papers of SIGMOD 2008 [ACM, 2008] was conducted. SIGMOD is an important conference in the database community and many of the submitted articles contain experimental results. The authors were asked to submit, together with the articles, a package containing information about the conducted experimental studies. A committee then assessed the experimental

repeatability. The participation in the repeatability review was optional. Nevertheless, 2/3 of all paper submissions attempted a repeatability review. This was the first attempt to measure the repeatability of experimental studies in the database community. We present the results of this experiment and discuss the lessons it teaches us. We enumerate problems that need to be addressed in order to make this a common practice.

The structure of this chapter is as follows. In Section 4.2, we present the setup of the SIGMOD 2008 repeatability review, including the schema used for describing the experiments and the evaluation protocol. In Section 4.4, we present the results of the reviewing process and the problems the repeatability committee encountered. In Section 4.5, we present an author survey targeted at getting insights about the usefulness of the reviewing process. Finally, in Section 4.6, we discuss the positive and negative sides of this approach and conclude.

4.2 SIGMOD repeatability review setup

For the first time in the history of the conference—SIGMOD has been running since 1975—SIGMOD 2008 introduced an experimental repeatability reviewing process. The goal was to evaluate whether the experiments presented in the submitted articles are repeatable by fellow researchers. A longer term goal was to develop a standard for reporting repeatable experimental studies. The authors of submitted articles were asked to assemble and submit a package containing information about the experimental studies, including the experimental data, a description of the experimental studies and environment. Based on that information and the results presented in the article, a committee formed of volunteer researchers assessed the repeatability.

Participation in the repeatability reviewing process was optional, that is, the submission and the results of the repeatability reviewing process did not have any influence on the paper acceptance to SIGMOD 2008. Nevertheless, if the authors decided not to participate in the repeatability review, they were required to submit a note stating the reasons for not participating. Some papers, especially those submitted to the industrial track of the conference, present research involving Proprietary Data (PD) and/or Intellectual Property (IP), not available for distribution. Therefore, they can not participate in the repeatability review. We consider these reasons as valid reasons for not participating.

The repeatability was evaluated per experiment presented in the paper. If the experiment was successfully executed following a description provided by the authors and if the obtained results led to the same conclusions as presented in the paper, then the experiment was considered repeatable. In Section 4.3 we present the protocol that was used for reporting the experiments and in Section 4.3.1 we present the protocol that was used for assessing the repeatability.

Participation was encouraged by allowing the articles with successfully eval-

uated experiments to include a note that acknowledged the repeatability of their experiments. If only a subset of the experiments could be verified, then the acknowledgement included references to those experiments. If the verified code and data was available for free distribution, the repeatability acknowledgment note contained an URI pointing to where they were stored. More importantly, all the papers that submitted their experiments for repeatability review, got detailed feedback on any problems that arose during the review. The official call for participation in the review can be found at `http://www.sigmod08.org/sigmod_research.shtml`.

The repeatability program committee consisted of 8 volunteers from the database research community, chaired by Ioana Manolescu and including the author of this thesis. In order to assure independence of the regular review from the repeatability review, the committee did not participate in the regular reviewing process of SIGMOD 2008. The reviewing process was double blind, i.e., the committee members did not know the identity of the paper authors and the authors did not know the identity of the reviewers evaluating the repeatability of their experiments.

4.3 Describing experimental studies

In this section, we present the protocol for describing experimental studies used in the SIGMOD 2008 repeatability reviewing process.

The authors were asked to provide a package containing the following information: (*i*) the PDF file containing the anonymized article, (*ii*) the software and data needed to run each experiment subject to the repeatability review, and (*iii*) experiment descriptions that contain all the details necessary for repeating the experiments, e.g., required hardware, software, instructions to install the software, to run experiments.

To achieve the repeatability of experimental studies, the authors need to describe how to set up and execute the experiments. It is also important to determine and list all the environment parameters that influence the results. Though each experiment has different requirements for repeatability, there are generic steps and parameters that the authors need to describe. These include:

- the machines used for the experiments, their hardware and software specifications;
- various third-party software (other than the software being tested) required for running the experiments;
- a detailed description on how to set up the experiments;
- a detailed description on how to execute the experiments; and

```
<!ELEMENT experiments_description (paper, machine*,software*,network*,
  experiment*, nonrepexperiment*, comment?)>

<!ELEMENT paper (#PCDATA)>

<!ELEMENT machine (hardware, os)>
<!ATTLIST machine id ID #REQUIRED>
<!ELEMENT hardware (proc, disk, memory, comment?)>
<!ELEMENT proc (make, model, bit, GHz)>
<!ELEMENT make (#PCDATA)>
<!ELEMENT model (#PCDATA)>
<!ELEMENT bit (#PCDATA)>
<!ELEMENT GHz (#PCDATA)>
<!ELEMENT disk EMPTY>
<!ATTLIST disk gigabytes CDATA #REQUIRED>
<!ELEMENT memory EMPTY>
<!ATTLIST memory megabytes CDATA #REQUIRED>
<!ELEMENT os (windows | linux | mac | otheros)>
<!ATTLIST os softwares IDREFS #IMPLIED>
<!ELEMENT windows (version, comment?)>
<!ELEMENT version (#PCDATA)>
<!ELEMENT linux (distribname, distribno, kernel, comment?)>
<!ELEMENT distribname (#PCDATA)>
<!ELEMENT distribno (#PCDATA)>
<!ELEMENT kernel (#PCDATA)>
<!ELEMENT mac (version, kernel, comment?)>
<!ELEMENT otheros (name, version, comment?)>

<!ELEMENT software (name, function, version?, downloadURL?, comment?)>
<!ATTLIST software id ID #REQUIRED>
<!ELEMENT name (#PCDATA)>
<!ELEMENT function (#PCDATA)>
<!ELEMENT downloadURL (#PCDATA)>

<!ELEMENT network (machineset+, comment?)>
<!ATTLIST network id ID #REQUIRED>
<!ELEMENT machineset EMPTY>
<!ATTLIST machineset mkey IDREF #REQUIRED howmany CDATA #REQUIRED>

<!ELEMENT experiment (dataset*, install, howto)>
<!ATTLIST experiment id1 CDATA #REQUIRED id2 CDATA #IMPLIED hardwkey
  IDREF #REQUIRED>
<!ELEMENT dataset (#PCDATA)>
<!ELEMENT install (#PCDATA)>
<!ELEMENT howto (#PCDATA)>

<!ELEMENT nonrepexperiment (#PCDATA)>
<!ATTLIST nonrepexperiment id1 CDATA #REQUIRED id2 CDATA #IMPLIED>
<!ELEMENT comment (#PCDATA)>
```

Figure 4.1: The DTD containing a schema for describing the experiments presented in a paper for the purpose of repeatability. This schema was used for the repeatability review of SIGMOD 2008.

- any comment regarding the repeatability of the experiments, including a list of all the parameters that might influence the results.

The repeatability committee designed a schema for describing experimental studies. The schema is meant as a guideline for presenting experiments and the submissions are meant to be read by humans and are not for automatic processing or execution of the experiments. It is a first step towards standardizing the description of experimental studies for the purpose of repeatability.

The authors were required to submit their descriptions in XML format conforming to the DTD presented in Figure 4.1. Below, we describe a subset of the elements defined by this DTD. We omit the self-explanatory elements.

- The `paper` element contains an identifier of the paper being tested within the submitted packadge.
- The `machine` element describes an individual computer used in the experiments. It consists of hardware and operating system specifications.
- The `hardware` element describes the machine's hardware. It is important to describe the resources actually used for the experiments. For instance, the experimental machine can have 2GB of RAM but if the software under test used only 1GB, then 1GB should be indicated here.
- The `os` element describes the operating system and references to any auxiliary software (e.g., compilers, libraries, database systems). If the experiment under test uses a virtual machine, it describes the characteristics of that virtual machine.
- The `windows`, `linux`, and `mac` elements describe the most common operating systems. The `otheros` element can be used to describe other operating systems.
- The `software` element describes any software (other than the code under test) needed in order to run or compile the code under test, for example, compilers used to compile the code, libraries, simulators, software for analyzing the data and producing graphs. If the experiment under test was run using a virtualization platform, then one `software` element can describe the virtual image player, and another `software` element can describe the virtual image used.
- The `function` element describes the functionality of the software in the experimental setting. This allows for the replacement of the software by another one with similar functionalities, in case needed.
- The `downloadURL` element specifies an URL where a free or trial version of the software can be found.

- The elements `network` and `machineset` can be used for experiments to be run on several machines. The `machineset` element specifies a set of identical machines: `@mkey` is a foreign key pointing to an existing `machine` element; `@howmany` specifies how many machines of this kind are part of the machine set.

- The `network` specifies a set of machines used in an experiment. The network may include homogeneous subsets of identical machines, each such subset being described by a `machineset` element. If all machines are different, 1-sized machine sets can be used.

- The `experiment` describes an experiment that can be repeated: the `@id1` attribute contains the unique id of the experiment (typically "Figure X" or "Table Y"); the optional attribute `@id2` is a secondary id (typically "left" or "top"), if any; and, `@hardwkey` is a reference to either a machine, or a network, depending on the hardware used for this experiment.

- The `dataset` element specifies the dataset(s) used in the experiment under test. It can contain: a path leading to the data file(s) within the submission package, a download URL, or the information necessary to produce it using a data generator.

- The `install` element contains the installation instructions for the software needed to repeat the experiment, mainly the software under test.

- The `howto` element contains the sequence of steps to be performed in order to run the experiment. It may include: any necessary pre-processing steps; the commands for performing the experiment; the number of runs, if any; the operations needed to gather and analyse the results; the production of graphs corresponding to those of the paper. The ideal description of a given step is the command line needed to run it.

- The `nonrepexperiment` describes an experiment that cannot be repeated due to IP rights or other issues. It contains a primary and possibly secondary key as for repeatable experiments.

All the experiments submitted for the repeatability review at SIGMOD 2008 were described using this schema.

4.3.1 Assessing repeatability

For assessing repeatability the committee members adhered to the following steps:

1. Checking whether the repeatability submission package conforms to the requirements given in the previous section;

2. Matching the hardware and software requirements with available machines;
3. Installing the necessary software, setting up the experiments, estimating the total time needed to execute them;
4. Running the experiments. If the execution takes considerably longer than the estimated time, it is considered failed; and
5. Comparing the obtained results with the results presented in the paper. For this step the following procedure is used:
   - If the performance measure deployed in the experiment is deterministic (e.g., measuring the size of data storage in a DBMS) then the committee member compares the absolute values of the obtained results and the results presented in the article.
   - If the measure deployed in the experiment is non-deterministic (e.g., measuring the query execution times) and the obtained results cannot be compared directly with the results presented in the article, then the committee member judges, case by case, whether the results support the observations and the conclusions made in the article. For example, a statement such as "Engine 1 performs twice as fast as Engine 2" can be confirmed or refuted even if the absolute measurements are different.

An experiment is considered *repeatable* if all these steps were successfully completed. Otherwise, a report is submitted to the authors indicating at what step the assessment failed and how it failed.

Note that the assessment made during the second part of Step 5 might be subjective. To improve the evaluation accuracy, it would have been preferable if two or more reviewers would discuss and agree upon the final assessment. Due to lack of time and (non) availability of suitable experimental environments, it was not possible for each paper to be reviewed by multiple reviewers. Instead, each paper was assessed only by one reviewer, who discussed the assessment with the rest of the committee only when in doubt about its correctness.

4.4 Results

In this section, we present the results of the SIGMOD repeatability review. First, we describe the timeline of the repeatability reviewing process and we give the submission statistics. Then we present the repeatability assessments.

Participation

Nov 16, 2007: SIGMOD's paper submission deadline. There were 436 paper submissions.

	Submitted experimental descriptions	Submitted valid excuses	No submission or invalid excuses
Accepted papers (78)	68% (53)	31% (24)	1% (1)
Rejected papers (358)	66% (236)	23% (82)	11% (40)
All papers (436)	66% (289)	24% (106)	9% (41)

Table 4.1: Participation in the SIGMOD 2008 repeatability reviewing process. The rows contain the statistics for the accepted papers, for the rejected papers, and for all submitted papers. The parentheses contain the absolute numbers.

Dec 16, 2007: The repeatability review submission deadline. For 289 papers an experimental description was submitted; for 106 papers valid reasons (deploying PD or code under IP rights) for not participating were submitted; the authors of the remaining 41 papers submitted reasons that the committee considers invalid (e.g., losing the experimental code and data) or did not submit any reason at all.

Jan 2, 2008: The regular program committee disclosed to the repeatability program committee a list of 82 papers with high chance of acceptance (2–3 positive reviews). Due to time constraints, the repeatability committee did not verify all of the papers that were submitted for repeatability review. The priority was given to papers with high chance of publication.

Feb 22, 2008: The SIGMOD program committee announced the list of 78 accepted papers. Then, the repeatability committee focused on these papers.

Mar 20, 2008: Camera ready submission deadline. By this date, 64 papers covering the accepted papers that participated in the repeatability review, were reviewed and the results were communicated to the authors.

Table 4.1 shows the distribution of participation in the repeatability reviewing process for all submitted papers, for the accepted papers, and for the rejected papers. Note that in the accepted papers category only 1% of the papers did not participate at all in the repeatability reviewing process, while in the rejected papers category 10%. This is easily explainable by the usual percentage of unfinished or invalid papers submitted to a conference. The rest of the percentages are quite similar for the accepted and rejected categories. Considering that this repeatability review experiment was conducted for the first time in the database community and on a voluntary basis, we find 66% participation rate a strong indication of the importance and usefulness of testing repeatability.

The most common invalid reasons the authors presented for not participating in the repeatability reviewing process, are the loss of the experimental data and third party software used in the experiments. Below we include a few quotes from the reasons provided.

	All repeated	Some repeated	None repeated
Accepted papers (53)	55% (29)	26% (14)	19% (10)
Rejected papers (11)	36% (4)	55% (6)	9% (1)
All assessed papers (64)	52% (33)	31% (20)	17% (11)

Table 4.2: Repeatability assessment results. The first two rows contain the statistics for the assessed papers split between the papers that were accepted for the conference and those that were rejected. The last row contains the aggregate statistics for all assessed papers. The parentheses contain the absolute numbers.

> "We cannot distribute code and data because the authors have moved, making the retrieval of code and data infeasible at this point."
>
> "We lost some old code. Due to the short notice, we could not reproduce our lost code for these parts."
>
> "The subsets were chosen randomly from a large dataset, and unfortunately no trace about the identity of the used documents has been kept. The experiments were performed long months ago, and it wasn't expected to send results to SIGMOD, that's why we didn't pay attention about keeping a trace."

These quotes indicate that the experiments were not conducted with repeatability in mind, which we consider bad practice. Some comments hinted at some misunderstandings of the purpose of the repeatability assessment:

> "My experimentation is fully deterministic: if it is wrong, running again my own program would not detect it."

These quotes underscore how useful assessing repeatability could be for the integrity of our field.

Assessment results Due to lack of time, only some articles were assessed by two committee members, namely those that required a second opinion for the final assessment.

Table 4.2 contains the results of the repeatability reviewing process. The first two lines present the statistics for the papers that were accepted for publication and for those that were rejected. The last line presents the total. Out of the 64 repeatability assessed papers, 53 papers were accepted to the conference and 11 papers were rejected. Out of the total of assessed papers, for 33 papers (52%) all experiments were repeated, for 20 papers (31%) some of the experiments were repeated, and for 11 papers (17%) none of the experiments were repeated. The percentage of papers of which all experiments were repeatable is slightly higher for the accepted papers than for the rejected papers.

Among the 11 papers with no repeated experiments, 3 required hardware unavailable to the repeatability committee, 2 required unavailable software, the

installation of the necessary software failed for 1 paper, and 5 papers had various runtime failures that prevented the completion of the experiment.

Assessments costs One committee member, the author of this thesis, recorded how much time it took her to verify the repeatability of 8 articles. The total amount of time needed for the whole assessment procedure, i.e., steps 1–5 presented in Section 4.3.1, varied from 7 hours to 180 hours. This time includes the running time of the experiments (step 4), which is experiment dependent. Without counting the running time of the experiments, the average amount of time that the reviewer spent on the evaluation of an article is 5 hours. Note that these are not continuous work hours, but rather aggregations of different work sessions. Usually, half of this time was spent on setting up the experiments and the other half on understanding and interpreting the results.

On average, each committee member assessed 8 articles. Supposing that every committee member needs the same amount of time for assessing a paper, then the total amount of time spent on the assessment is 40 hours per person and 320 man-hours in total.

Since the repeatability review has been organized for the first time, a large amount of time was spent on the setup. In the future, this time can be reduced by building an automated system for collecting and distributing the articles among the reviewers.

4.5 Authors survey

After the repeatability evaluation was completed, the authors of the papers were asked to participate in a short survey. The purpose was to get insights into whether the SIGMOD repeatability reviewing process was useful, whether it should be continued, and in what format. In this section, we present the results of this survey.

The survey participation request is given in Figure 4.2. The survey contained four questions. We discuss only the first three, the fourth question is outside the scope of this chapter. The first question recorded the authors' participation to the assessment (did not participate; participated and all experiments repeated; participated and some experiments repeated; or, participated and none of the experiments repeated). The second question asked the authors if they found the process useful, and elicited suggestion on how to improve it. The third question asked the authors whether they would participate in a repeatability reviewing process in the future SIGMOD conferences, assuming it would remain optional. The results of the survey were anonymized to encourage the authors to speak their mind.

Most answers were clear Yes/No answers. Less than 20% of the answers were ambiguous, in the style of "Yes and no; on one hand... but on the other hand..."

This is meant to be a sub-5 minute survey about experimental repeatability. In the case of multi-author papers, only one of you needs to answer (though we are happy to receive comments from more than one). We will strip your email headers from your responses programmatically, so please speak your mind.

1. *Did your paper succeed on all/some/none of the repeatability tests? Or did you not submit for intellectual property reason?*
2. *If you submitted, was the repeatability experience helpful? If so, how? If not, how could it be improved?*
3. *Would you attempt repeatability in the future if it remained voluntary (i.e. had no effect on acceptance decision but you would be allowed to mention success in your paper) and you had no intellectual property constraints?*
4. *Do you think it would be useful to have a Wiki page for each paper so the community could comment on it, you could post code etc.?*

Warm Regards, Ioana (repeatability chair) and Dennis (program committee chair)

Figure 4.2: The request for participation in the authors' survey.

For such answers, half a point was counted.

Table 4.3 presents the number of positive answers to the second and third questions about the usefulness of the repeatability review. The results are grouped according to the answer to the first question about the authors' participation to the repeatability assessment and their results. Most of the participants to the review (80%) found the repeatability process useful, though the participants with all experiments repeated were more positive (85%) than the ones with none of the experiments repeated (56%). Even more participants (84%) reported that they would participate in the repeatability reviewing process at future SIGMOD conferences. This indicates that the authors believe that such a repeatability review is useful.

Below we present a list of representative quotes extracted from the authors' answers about the usefulness of the process.

> "Yes, it was helpful to organize the source code properly for future use." (All experiments repeated.)
>
> "It was helpful. It forced me to write documentation which I would otherwise have postponed indefinitely." (Some experiments repeated.)
>
> "It was helpful. It required us to further clean up my code and scripts and prepare documentation." (Some experiments repeated.)
>
> "Helpful? Greatly yes. Some scripts written for this test could be used to append additional experimental results immediately. To package experiments in a script form, at first, seemed bothersome, but we

	Number of participants	Found it useful	Would do it again
Did not participate	16	–	12 (75%)
All experiments repeated	24	20.5 (85%)	21.5 (90%)
Some experiments repeated	12	10 (83%)	11 (91%)
None of the experiments repeated	8	4.5 (56%)	6 (75%)
Total	60	35 (80%)	50.5 (84%)

Table 4.3: Authors survey on the usefulness of the repeatability review at SIGMOD 2008. The survey participants are split in 4 categories depending on their participation and their results in the repeatability review. The last two columns contain the counts of positive answers to the respective questions. A half point (0.5) was given to the answers of the form "Yes, and No."

> found out that it is good for ourselves, and improves our productivity." (All experiments repeated.)
>
> "It is a great thing for the community that this service is available, and I hope that it will have a very positive effect on both the trustworthiness of SIGMOD results and the quality of publicly-available research tools." (All experiments repeated.)
>
> "It's only helpful in the sense that it provides some extra credibility to the paper. It was not helpful to myself in any way." (Some experiments repeated.)
>
> "We are happy to see that our algorithms show consistent results through machines with different hardware/software configuration." (All experiments repeated.)

In the next section, we list the lessons learned and conclude.

4.6 Lessons learned and conclusions

In this chapter, we pursued Question 4.1: "How to ensure the repeatability of experimental studies of database systems?" The SIGMOD 2008 repeatability reviewing process provides a solution and shows that repeatability is possible to achieve and measure.

Referring to Question 4.1 (i), the methodology used for describing and reporting experiments that the repeatability committee developed was enough to cover the 289 papers that were submitted for the repeatability review. Out of 64 papers that were assessed by the repeatability committee, 33 (52%) achieved the repeatability of all presented experiments and 20 (31%) achieved repeatability of some of the experiments. Considering that we strive for all experiments to be

repeatable, 52% is a low fraction. Nevertheless, we consider these results to be a good start towards achieving the repeatability of experimental studies in the database research field.

Referring to Question 4.1 (ii), the high percentage of participation in the optional review, 66% of the total submissions to the conference (289 out of 436), hints at the usefulness of a peer reviewing process. The positive feedback from the authors of the papers recorded by the survey also indicates that such a review is useful for the community: 80% of the surveyed authors found the process useful, while 84% would participate in such a process in the future.

Lessons learned Our experience with the SIGMOD experiment showed that there are a few problems that need to be addressed, if ensuring repeatability of experimental studies is to become a common practice in the database community.

The first problem is the effort the authors need to undertake to ensure the repeatability of their experiments. Currently, due to a lack of common practice, achieving repeatability is tedious and time consuming. A software tool can be very useful for automating (parts of) this task. In Chapter 5, we present a tool for automating benchmark tasks in the context of XQuery, including the automated recording of the testing environment specifications for the purpose of repeatability. More such tools are needed.

The second problem is the tremendous amount of handwork that the reviewing committee had to do in order to set up and verify the repeatability. We estimated that it took 320 (40×8) human hours for the assessment process alone. It is not realistic to pursue such a review in the future without a proper framework that would facilitate the task. We also expect that the assessment will become easier as the practice of presenting experiments in a way that facilitates repeatability will increase.

A third problem is reducing the number of papers that claim non-repeatability of their experiments due to legal reasons. At SIGMOD 2008, 24% of the paper submissions could not attempt the repeatability review because they deploy PD and/or use software under IP rights that does not allow free distribution. This number can be reduced by designing and using openly available benchmarks. In Chapter 3, we show that the standard benchmarks are rarely used for experimental studies, while it is possible and advisable to design synthetic data and workloads that emulate the properties of real data [Jain, 1991].

Future work One aspect of repeatability that we did not discuss in this chapter is what is a proper archiving mechanism for ensuring the accessibility of experimental data and results. Long-term preservation and even experimental results curation is another key factor of scientific proliferation. This question is actively being addressed in other database related fields, such as Information Retrieval [Agosti *et al.*, 2007]. We leave this as future work.

Chapter 5

XCheck: a Tool for Benchmarking XQuery Engines

In Chapters 3 and 4, we identified the need for a software tool that helps executing benchmarks and reporting performance results. In this chapter, we address the question whether it is possible to automate the execution of performance benchmarks on many XML query engines and the comparison of their performance? As a solution, we present XCheck, a tool for running performance benchmarks that measure execution times on sets of XML documents and sets of queries, formulated in an XML query language, such as XPath and XQuery. Given a benchmark and a set of engines, XCheck runs the benchmark on these engines, collects performance times, query results, and the testing environment configuration. Version 0.2.0 of XCheck runs the XPath and XQuery benchmarks that were openly available in 2007 on 9 different XPath and XQuery engines. XCheck's design makes it easy to include new engines and new benchmarks.

Part of this work was published in [Afanasiev *et al.*, 2006].

5.1 Introduction

The task of *running a performance benchmark (T1)* is notoriously tedious and time consuming [Jain, 1991]. This task becomes even more problematic, when benchmarking engines in a relatively new research field, like XML querying, where the engines are still immature and require a lot of tuning and error-handling (see Section 3.6 of Chapter 3). The difficulties grow when evaluating the relative performance of several engines. In Section 3.6, we have seen that such evaluations are very useful, but tedious to implement—one has to keep track of the execution of many engines, handling their output and performance times.

As we argue in Chapter 4, one important aspect of performance evaluation is its repeatability. In order to make an evaluation repeatable, one needs to rigorously document all performance critical parameters of the engines and the testing

environment; evaluations that do not take in consideration this information can lead to misleading conclusions [Jain, 1991]. We call this task *documenting the benchmark experiment (T2)*. This is yet another tedious task that many performance evaluators fail to complete (see Chapter 4).

After running the benchmark and gathering the measurements comes the most difficult task: *analyzing the benchmark results (T3)*. This requires mostly intellectual work, though analyzing large numbers of measurements is impossible without statistical tools and data visualization.

To a large degree, the three tasks described above—running a performance benchmark (T1), documenting the benchmark experiment (T2), and analyzing benchmark results (T3)—can be automated. A software tool is needed to help execute these tasks under a single roof, from running the experiments to aggregating the results and presenting them in an easily-readable format. Such a tool is valuable to

- *developers* and *researchers*, for evaluating the performance of their engine or research prototype, also in comparison with other implementations; and
- *users*, for comparing and choosing a query engine that performs well on their data.

We are aware of only two open source tools, BumbleBee [BumbleBee, 2006] and XSLTMark [XSLTMark, 2006], that target the automatic execution of benchmarks in the area of XML querying. BumbleBee is a test harness aimed at testing the correctness of XQuery engines, rather than performance. XSLTMark is a performance benchmark and testing platform for XSLT engines only. The goals of both tools are too narrow for covering the tasks described above. Moreover, as of September 2008, both testing platforms are no longer available. We discuss these tools in more detail in Section 5.4. Further, there are no generic testing platforms that aim at automating tasks T1–T3 and at accommodating any XML query engine and any benchmark.

In this chapter we address the following question:

5.1. Question. *Is it possible to build a generic tool for automating the tasks T1–T3 and what are the design choices that need to be made?*

We answer these questions by developing XCheck. XCheck is a testing platform for running performance benchmarks measuring *performance times*. It runs benchmarks whose atomic measure is the performance time of processing *one query* possibly against a *document/collection* in *a single-user scenario*. The platform can test any XML query engine, stand-alone or DBMS, that has a *command line interface*. It allows one to test several query engines in one experiment and helps analyzing their relative performance; new benchmarks and engines can easily be added.

XCheck is delivered under the GNU General Public License and it is freely available at `http://ilps.science.uva.nl/Resources/XCheck/`.

This chapter is organized as follows. In Section 5.2, we describe XCheck's functionalities and architecture. In Section 5.3, we give an example of XCheck's usage and describe XCheck's coverage. In Section 5.4, we describe the two related tools mentioned above, BumbleBee and XSLTMark. We conclude in Section 5.5.

5.2 XCheck

In this section, we describe XCheck's goals, functionalities, and architecture. We give a general overview of XCheck; for a complete description of the platform and its options, see XCheck's webpage: `http://ilps.science.uva.nl/Resources/XCheck/`.

XCheck's goal is to automate tasks T1–T3 described in the introduction while satisfying the following requirements: (*i*) it should have a good coverage of existing XML query engines and performance benchmarks; (*ii*) it should be easy to integrate new engines and benchmarks; (*iii*) it should have a flexible interaction with the engines, e.g., it can selectively collect the data that the engine outputs; (*iv*) it should allow for the comparison of performance of several engines; (*v*) it should have an output that is easily readable for humans, but also allows for further automatic processing; and (*vi*) it should be easy to run.

As a result, XCheck is a testing platform that takes as input a performance benchmark workload consisting of a set of XML documents and a set of queries, and runs it on a given set of XML query engines. The platform targets benchmarks whose atomic measure is the performance times of processing one query, possibly against a document/collection. It can test stand-alone or DBMS XML query engines that have a command line interface. It communicates with the engines via an engine adapter that allows the users to specify all the relevant information about the engine, including running instructions and output format. It has an easy-to-run command line interface, and a workflow oriented towards minimizing the total time the user spends on execution and analysis of the benchmark. It stores the experimental data in XML for further automatic processing. It also aggregates the results, builds plots and presents the information in HTML format for a quick overview and human interpretation.

XCheck has the following functionalities:

1. Running performance benchmarks, collecting performance times, errors, and optionally, query results. These functionalities address tasks T1 and T2 presented in the introduction;

2. Documenting engine configuration and the configuration of the testing environment. These functionalities address task T2;

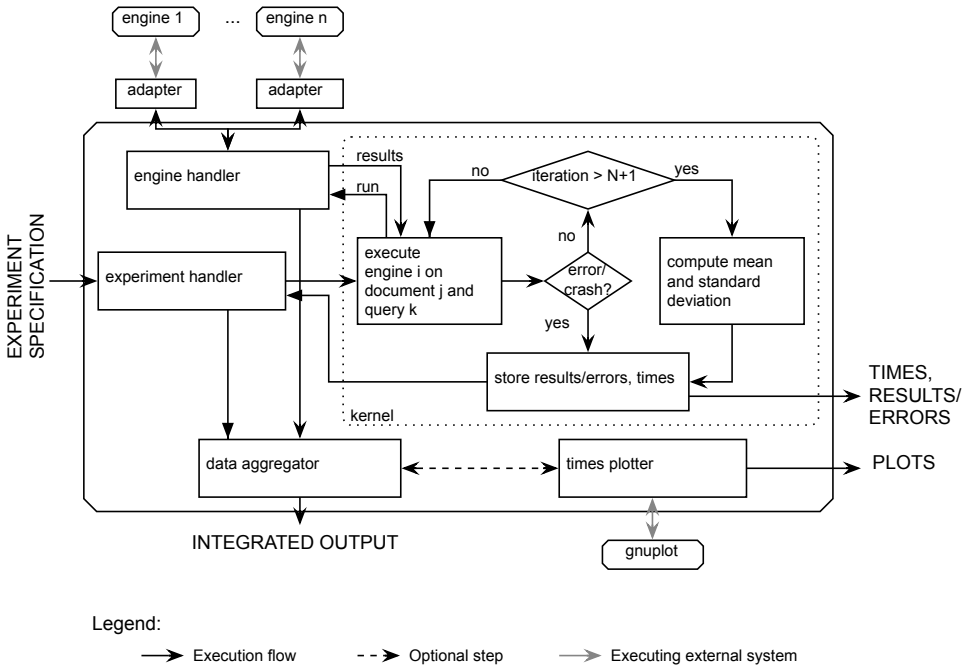


Figure 5.1: XCheck architecture and workflow.

3. Facilitating the analysis of performance measurements, by computing statistics and generating plots of the results. These functionalities address task T3.

In the following subsection, we present XCheck's architecture and workflow in more detail.

5.2.1 Architecture and workflow

XCheck works in two steps. During the first step, XCheck runs a given experiment specification describing which XML query engines to run on which documents and on which queries and it gathers the engines' outputs and processing times. During the second step, XCheck aggregates the data obtained during the running step, and documents the configuration of the engines and of the testing environment. The output of the running step is raw performance time measurements and results presented in a machine-readable format, while the output of the augmenting step is a collection of raw and aggregate data presented in an integrated human-readable format.

The general workflow of XCheck is shown in Figure 5.1. The EXPERIMENT SPECIFICATION label denotes XCheck's input. The TIMES, RESULTS, and ERRORS labels denote raw performance time measurements, query processing results, and errors, respectively. The PLOTS label denotes different plots created

for visualization of the raw performance times. The INTEGRATED OUTPUT label denotes an easy-to-read output that integrates all the raw measurements and aggregate data collected by the platform. A detailed description of XCheck components and workflow follows below.

Input XCheck takes as input an *experiment specification*. An experiment specification consists of: (*i*) a non-empty list of XML query engines; (*ii*) a possibly empty list of documents/collections, or the commands to generate the documents whenever a document generator is provided; (*iii*) a non-empty list of queries, or the commands to generate the queries whenever a query generator is provided. The list of documents can be empty since, for example, XQuery queries do not necessarily need an input document, a query can construct a document. This design is sufficient to represent all XQuery benchmarks presented in Chapter 3. The input is represented in XML. Since XML is both human and machine readable, it is a natural choice for this purpose.

Engine adapters XCheck communicates with the XML query engines via *engine adapters*. The platform requires that the XML query engine has a command line interface for the execution of query processing operations: document processing, query compilation and processing. Then XCheck executes the engine via a system call on a particular input and stores the engine's output. The engine adapter contains: (*i*) the engine's command line execution instructions, (*ii*) a formal description of the engine's output format, and (*iii*) natural language documentation of the engine.

The execution instructions are specified as a template with the running parameters, like specific query and/or document, to be filled in by XCheck at the running step. The output is described with the help of regular expressions that allow XCheck to extract pieces of relevant information from the engine's output. The documentation of the engine contains the description of the performance critical engine parameters, like the software version, software compilation options, engine execution options, etc. The adapters are represented in XML, again chosen because it is both human and machine readable. The *engine handler* shown in Figure 5.1 is responsible for the validation and interpretation of the engine adapters.

Kernel The role of the *kernel* of XCheck consists of executing one atomic performance measurement: running one engine on one given query and possibly on a document/collection and measuring the query processing times. If the engine crashes or outputs an error at the first execution, the kernel stores this information. Examples of errors that an engine outputs are: document parsing errors, static and dynamic query processing errors [World Wide Web Consortium, 2007b]. Otherwise, the kernel re-executes the same measurement another N times ($N+1$

times in total) and stores the mean and the standard deviation of different processing times obtained during these last N executions. The times obtained during the first execution are ignored. This is done to increase the accuracy of the time measurement (see Section 5.2.2). Optionally, XCheck can store the query results output by the engine. In Figure 5.1, the kernel is indicated with a dotted line.

The general running strategy of XCheck is to iterate through each engine, each document/collection, and each query in the order given by the experiment specification, and to execute the kernel for each triplet (engine, document, query) or pair (engine, query), in case the document list is empty. This is the job of the *experiment handler* from Figure 5.1.

Data aggregator The *data aggregator* does the following: (*i*) it aggregates the raw performance times and presents them in a easily-readable format like tables and plots; and (*ii*) it documents the configuration of the engines and testing environment. It also performs a "pseudo-correctness" test of the query results. This test is meant to signal possibly wrong query results that might invalidate the performance studies.

When the experiment has finished executing, the data aggregator collects the raw performance times and computes simple descriptive statistics over them: the *sum* and the *average processing times* per engine, per engine and document, and per engine and query. Also, it keeps track of the total running time. The performance times and statistics are presented in tables. Optionally, the data aggregator calls the *times plotter* (Figure 5.1) to generate several types of plots. XCheck plots the times for: (*i*) each engine, (*ii*) each document, (*iii*) each query, (*iv*) each engine and document, and (*v*) each engine and query. The plots are generated with Gnuplot [Williams *et al.*, 2008]. To make it easy for the user to edit the plots, XCheck stores the Gnuplot code used to generate them. Examples of plots produced by XCheck are given in Section 5.3.

Further, the data aggregator collects and documents the configuration of the engines and of the testing environment. The environment information such as computer details: CPU frequency, cache memory size, available hard disk size, and the operating system, are obtained automatically. The engine configuration—performance critical parameters such as version, compilation options, and execution parameters—are retrieved from the engine adapters.

XCheck's main goal is to automate the execution of *performance* benchmarks as opposed to *correctness tests*. A considerable effort is done by the W3C XML Query Testing Task Force to develop and maintain the XML Query Test Suit (XQTS) [World Wide Web Consortium, 2006a] that provides query results and guidelines for assessing the correctness of an XQuery implementation. Executing such correctness tests is out of the scope of XCheck—when evaluating the performance of an engine, the correctness of its results is assumed. However, in practice, implementations are often incomplete or erroneous, therefore the data

aggregator also performs a test meant to signal the user when an engine outputs possibly wrong results. This test is done by comparing the size of the output of the different engines. A warning is produced if the size of a result output by an engine significantly differs from the average size of the results output by the other engines participating in the experiment. This test is not accurate and it can be applied only when there is more than one engine in the experiment, nevertheless, it proved to be a useful tool for the benchmark analysis presented in Chapter 3.

Output The default output consists of an XML document containing the query processing times and the error messages of the failed queries, grouped by engines, documents and queries. It also contains the total experiment running time, engine and testing environment configuration. Optionally, XCheck saves the answers to the queries. Another optional output is a large set of plots displaying the performance times. The Gnuplot code for generating these plots is also provided, so that the user can easily edit and modify them. We chose the XML format to store this information in order to facilitate future automatic processing. The user can compute any aggregate measure or statistics that are not implemented by XCheck by using an XML query engine.

A more readable HTML format containing all the information collected or computed by XCheck is also provided and browsable from a single HTML webpage. The HTML webpage lists the following information: a natural language description (provided by the user) of the experiment; the total time it took to execute the experiment; a link to the XML file containing the input experiment specification; the list of engines, their description and configuration; the list of documents, their description, and links to the actual files; the list of queries, their description, and links to the files containing the queries; the configuration of the testing environment; a link to the XML file containing the output; a list of tables, one per engine, containing the performance times or errors per document (the rows) per query (the columns); a list of plots containing the average processing times for each engine, the average times for each document and engine, the average times for each query and engine; 5 links to HTML galleries of plots, one per each type of plot that XCheck outputs; and finally, a table containing the results of the pseudo-correctness tests—one row per document, one column per query, each cell contains the result sizes obtained with each engine. The HTML presentation greatly improves the readability of the performance results produced by the experiment. An example of XCheck's output is given in Figures 5.2–5.10 in Section 5.3.

5.2.2 Collecting performance times

XCheck measures the total time it takes to run an engine on one query and zero or one document/collection, i.e., one run of the kernel. The platform measures the CPU time with the Unix command `time`. The unit of measure is seconds.

XCheck also keeps track of the wall-clock time elapsed from the beginning of the experiment until its end. This time is reported to the user as an indication of the total amount of time it takes to run the experiment. This time is not used for comparisons.

To avoid unreliable results, XCheck runs the same experiment $N + 1$ times and takes the average and the standard deviation of the last N evaluation times. The first measurement is ignored because it usually contains the warm-up time of the operating system, i.e., the time spent on loading the application in the system's cache. For Java applications, the Java virtual machine imposes another level of software between the application and the operating system that may alter the runtimes of the applications under evaluation. Taking the average over the last N evaluation times improves the accuracy of the measurement. In our experiments of executing the standard XQuery benchmarks on 4 different engines (see Chapter 3), we took $N = 3$ and obtained standard deviations within 2% of the mean time. Based on this observation we set the default value of N to 3. The user can change this with a command line option of XCheck.

It is often desirable to measure the times taken by individual processing steps, such as document processing time, query compilation time, query execution time, and result serialization time [Manolescu *et al.*, 2008b]. Measuring these times is difficult or impossible, unless the engine provides this information. In the latter case, XCheck captures these times. XCheck defines and collects, if available, the following types of query processing times:

- *document processing time* is the time that an engine takes to parse the input XML documents and create the internal document representation (in main memory or disk).

- *query compile time* is the time an engine takes to parse the query and translate it in an internal formalism of the engine. This time includes query normalization and rewriting, if implemented by the engine.

- *query execution time* is the time the engine takes to execute the query. This time includes only the time it takes to locate and/or construct the query results without outputting them. Usually, this is the most interesting elaboration time for comparing different query processing techniques and implementations.

- *serialization/output time* is the time it takes an engine to serialize and output the query result.

- *total time* is the total time an engine takes to process a query, starting with the engine invocation until the engine outputs results.

Out of all these times, the total time is measured by XCheck, while the rest can be provided by the engines. It is not always possible to separate the query

processing steps, and it is not always the case that an engine outputs detailed times, but whenever it does, XCheck can record them. Note that XCheck is not responsible for the accuracy of these times, nor can it determine what unit of measure they use, CPU or wall clock time. It is important to remember this fact when comparing the detailed performance times of different engines, since the measurements might not always be comparable.

5.3 XCheck in action

In this section we give an example of XCheck's usage by running the most popular XQuery benchmark, XMark [Schmidt *et al.*, 2002], on 4 XQuery engines: Saxon [Kay, 2009], Galax [Fernández *et al.*, 2006], Qizx/Open [Axyana Software, 2006], and MonetDB/XQuery [Boncz *et al.*, 2006b]. Further, we show XCheck's coverage in terms of the engines and benchmarks it already accommodates.

5.3.1 Running XMark

As an example of XCheck usage and output, we run the XMark benchmark on the following XQuery engines: SaxonB 8.7, Galax 0.5.0, Qizx/Open 1.0 and MonetDB/XQuery 0.10.3. The input query set consists of the 20 XMark queries and the document set consists of 7 documents corresponding to the scaling factors[1] 0.016 to 1.024 of size 1.80 MB to 113.99 MB, respectively. The times reported are the mean of the last three ($N = 3$) executions. The experiment was run on a machine with the following specifications: Intel(R) Xeon(TM) CPU 3.40GHz, with 2 GB of RAM, running Debian Gnu/Linux version 2.6.16.

Figures 5.2–5.10 give a first impression of the HTML webpage output by XCheck. The full output of this example is accessible at `http://ilps.science.uva.nl/Resources/XCheck/results/Lisa/xmark/output/outcome.html`.

Figures 5.11, 5.12, and 5.13 give a closer look at 3 plots output by XCheck. Figure 5.11 shows the relative performance of the four engines, by showing the total execution times for each query on one specific document of size 57MB (document scaling parameter f=0.512). This time is measured by XCheck, and it is the CPU time of one kernel run measured in seconds. Note that there are a few missing values in the plot: Galax crashes on queries Q11 and Q12, which contain nested for-loops and data value joins; Qizx/open outputs static type checking errors on Q3, Q11, Q12, and Q18, which contain the `op:numeric-multiply(A,B)` operation on arguments with static type `xs:anySimpleType`. The plot gives a quick overview of the relative performance when the engines are treated as off-the-shelf (no special tuning) and on-the-fly (no document pre-processing) XQuery

[1] As detailed in Chapter 3, XMark provides a document generator that produces documents whose sizes are proportional to a unique parameter called the scaling factor. A scaling factor of 1 produces a document of about 100 MB.

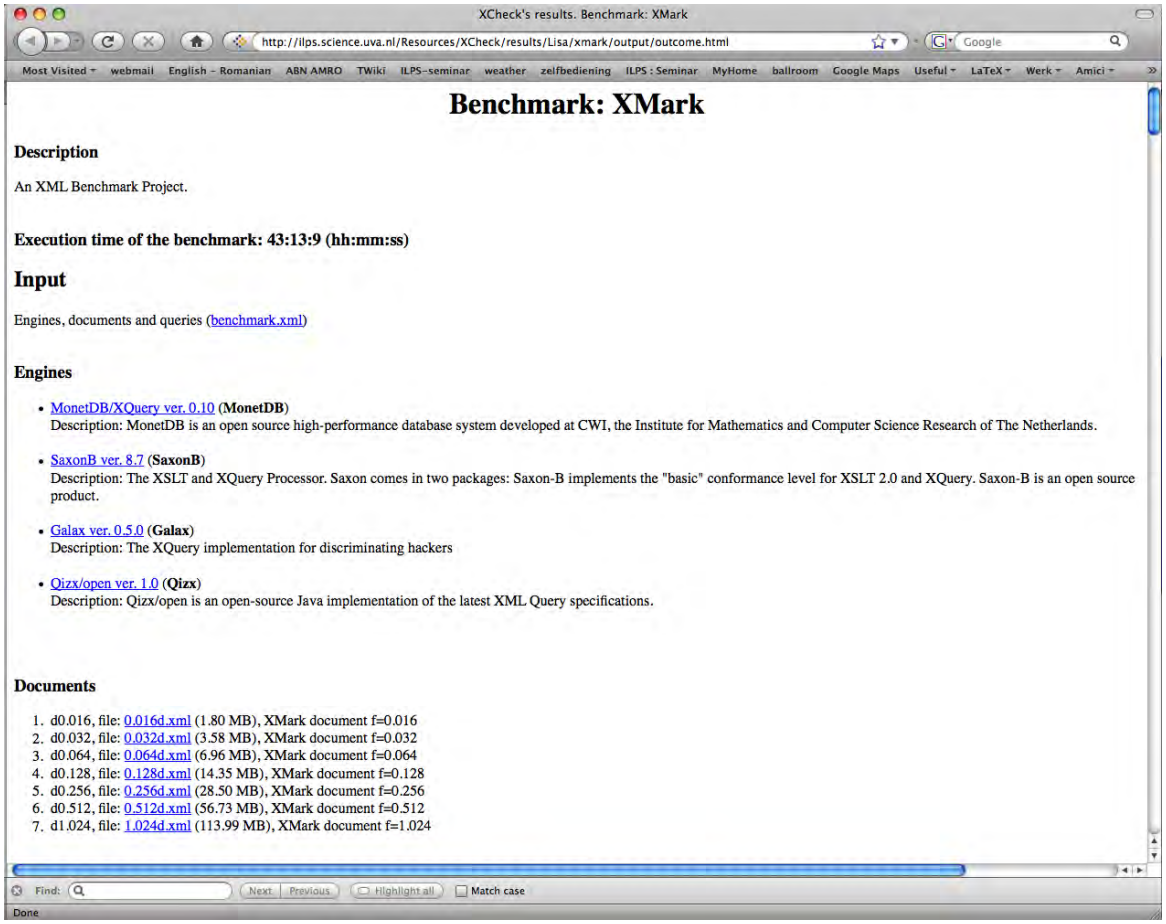
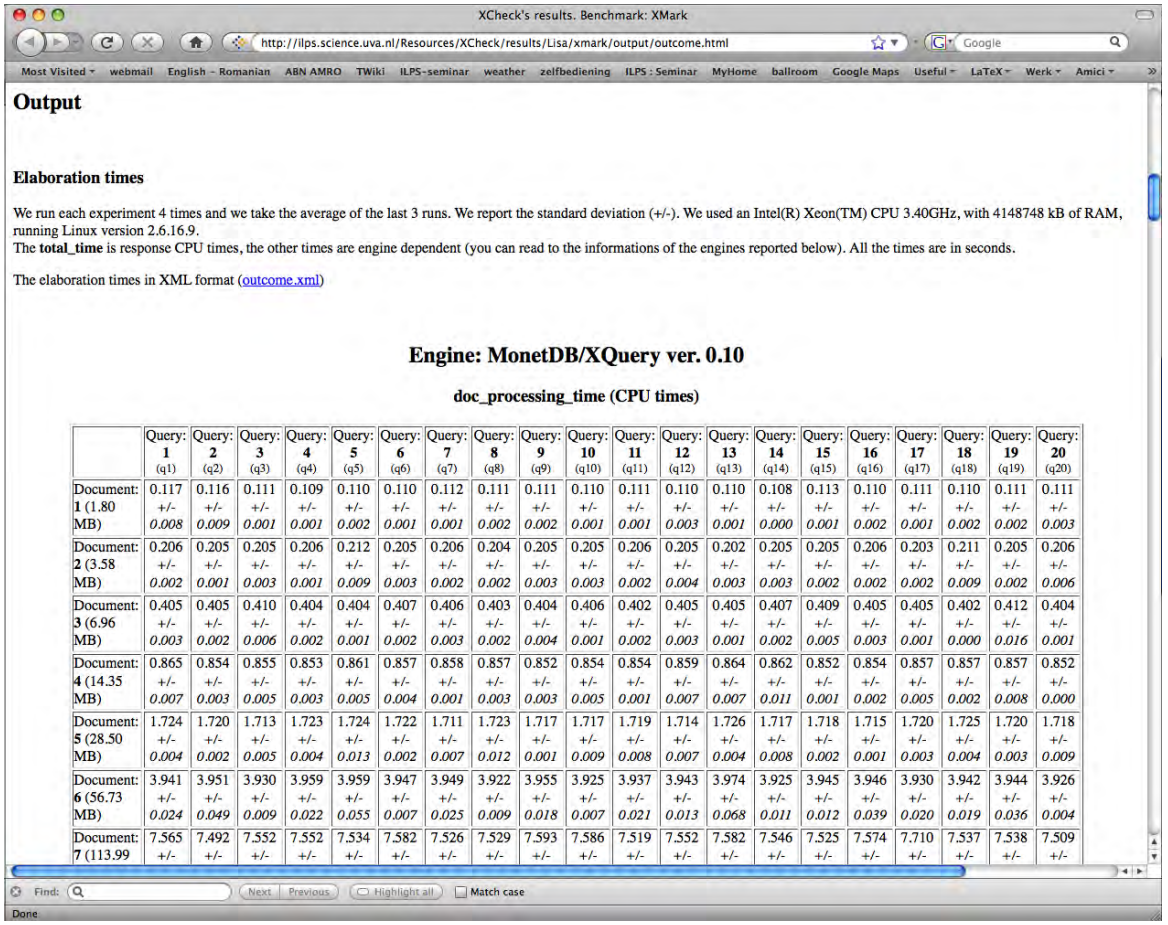


Figure 5.2: XCheck's HTML output listing: the name of the experiment; a short description of the experiment; the total time it took to run the experiment; a list of engines that were tested, their description and specifications; a list of documents, their names, sizes, and description.



	Query: 1 (q1)	Query: 2 (q2)	Query: 3 (q3)	Query: 4 (q4)	Query: 5 (q5)	Query: 6 (q6)	Query: 7 (q7)	Query: 8 (q8)	Query: 9 (q9)	Query: 10 (q10)	Query: 11 (q11)	Query: 12 (q12)	Query: 13 (q13)	Query: 14 (q14)	Query: 15 (q15)	Query: 16 (q16)	Query: 17 (q17)	Query: 18 (q18)	Query: 19 (q19)	Query: 20 (q20)
Document: 1 (1.80 MB)	0.117 +/- *0.008*	0.116 +/- *0.009*	0.111 +/- *0.001*	0.109 +/- *0.001*	0.110 +/- *0.002*	0.110 +/- *0.001*	0.112 +/- *0.001*	0.111 +/- *0.002*	0.111 +/- *0.002*	0.110 +/- *0.001*	0.111 +/- *0.001*	0.110 +/- *0.003*	0.110 +/- *0.001*	0.108 +/- *0.000*	0.113 +/- *0.001*	0.110 +/- *0.002*	0.111 +/- *0.001*	0.110 +/- *0.002*	0.111 +/- *0.002*	0.111 +/- *0.003*
Document: 2 (3.58 MB)	0.206 +/- *0.002*	0.205 +/- *0.001*	0.205 +/- *0.003*	0.206 +/- *0.001*	0.212 +/- *0.009*	0.205 +/- *0.003*	0.206 +/- *0.002*	0.204 +/- *0.002*	0.205 +/- *0.003*	0.205 +/- *0.003*	0.206 +/- *0.002*	0.205 +/- *0.004*	0.202 +/- *0.003*	0.205 +/- *0.003*	0.205 +/- *0.002*	0.206 +/- *0.002*	0.203 +/- *0.002*	0.211 +/- *0.009*	0.205 +/- *0.002*	0.206 +/- *0.006*
Document: 3 (6.96 MB)	0.405 +/- *0.003*	0.405 +/- *0.002*	0.410 +/- *0.006*	0.404 +/- *0.002*	0.404 +/- *0.001*	0.407 +/- *0.002*	0.406 +/- *0.003*	0.403 +/- *0.002*	0.404 +/- *0.004*	0.406 +/- *0.001*	0.402 +/- *0.002*	0.405 +/- *0.003*	0.405 +/- *0.001*	0.407 +/- *0.002*	0.409 +/- *0.005*	0.405 +/- *0.003*	0.405 +/- *0.001*	0.402 +/- *0.000*	0.412 +/- *0.016*	0.404 +/- *0.001*
Document: 4 (14.35 MB)	0.865 +/- *0.007*	0.854 +/- *0.003*	0.855 +/- *0.005*	0.853 +/- *0.003*	0.861 +/- *0.005*	0.857 +/- *0.004*	0.858 +/- *0.001*	0.857 +/- *0.003*	0.852 +/- *0.003*	0.854 +/- *0.005*	0.854 +/- *0.001*	0.859 +/- *0.007*	0.864 +/- *0.007*	0.862 +/- *0.011*	0.852 +/- *0.001*	0.854 +/- *0.002*	0.857 +/- *0.005*	0.857 +/- *0.002*	0.857 +/- *0.008*	0.852 +/- *0.000*
Document: 5 (28.50 MB)	1.724 +/- *0.004*	1.720 +/- *0.002*	1.713 +/- *0.005*	1.723 +/- *0.004*	1.724 +/- *0.013*	1.722 +/- *0.002*	1.711 +/- *0.007*	1.723 +/- *0.012*	1.717 +/- *0.001*	1.717 +/- *0.009*	1.719 +/- *0.008*	1.714 +/- *0.007*	1.726 +/- *0.004*	1.717 +/- *0.008*	1.718 +/- *0.002*	1.715 +/- *0.001*	1.720 +/- *0.003*	1.725 +/- *0.004*	1.720 +/- *0.003*	1.718 +/- *0.009*
Document: 6 (56.73 MB)	3.941 +/- *0.024*	3.951 +/- *0.049*	3.930 +/- *0.009*	3.959 +/- *0.022*	3.959 +/- *0.055*	3.947 +/- *0.007*	3.949 +/- *0.025*	3.922 +/- *0.009*	3.955 +/- *0.018*	3.925 +/- *0.007*	3.937 +/- *0.021*	3.943 +/- *0.013*	3.974 +/- *0.068*	3.925 +/- *0.011*	3.945 +/- *0.012*	3.946 +/- *0.039*	3.930 +/- *0.020*	3.942 +/- *0.019*	3.944 +/- *0.036*	3.926 +/- *0.004*
Document: 7 (113.99	7.565 +/-	7.492 +/-	7.552 +/-	7.552 +/-	7.534 +/-	7.582 +/-	7.526 +/-	7.529 +/-	7.593 +/-	7.586 +/-	7.519 +/-	7.552 +/-	7.582 +/-	7.546 +/-	7.525 +/-	7.574 +/-	7.710 +/-	7.537 +/-	7.538 +/-	7.509 +/-

Figure 5.3: XCheck's HTML output listing: how the time measurements were computed; the hardware and software specifications of the machine on which the experiment was run; detailed document processing times for MonetDB/XQuery.

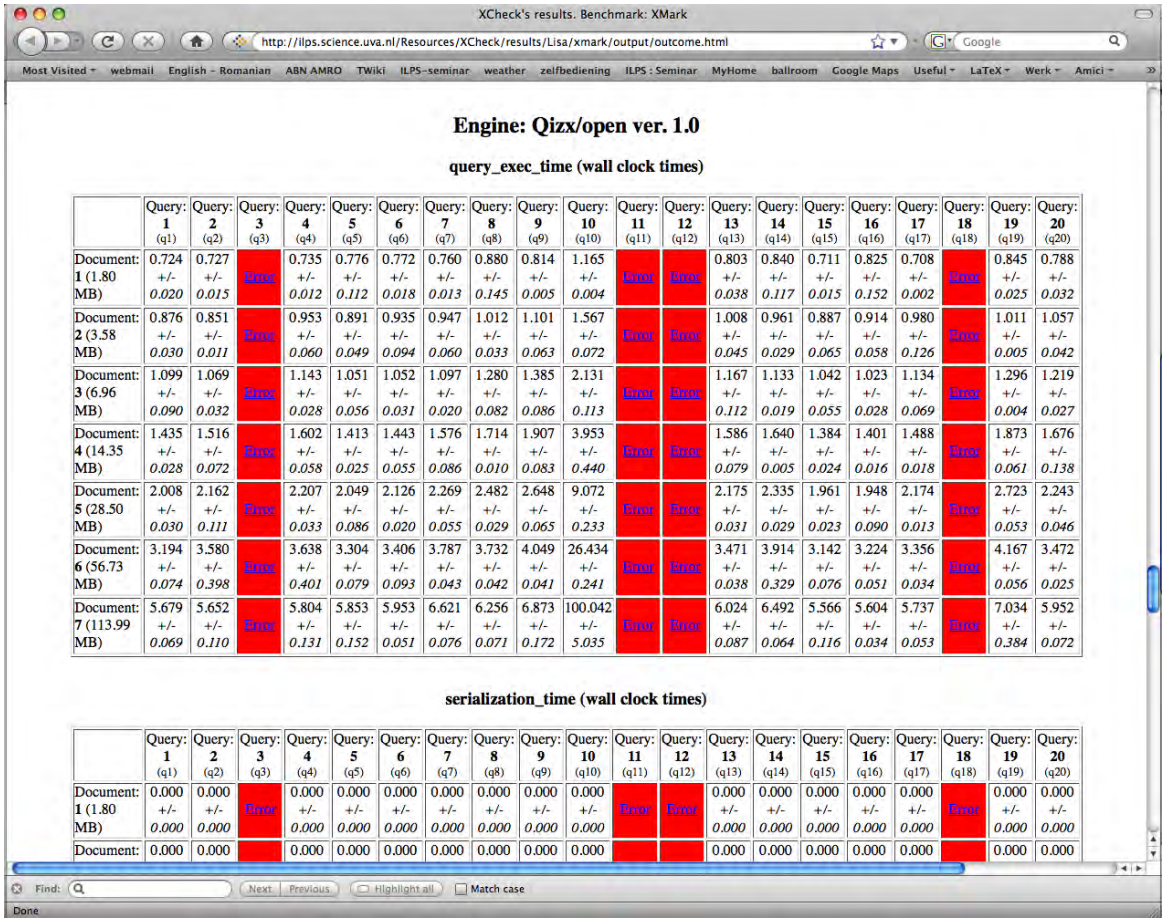


Figure 5.4: XCheck's HTML output listing detailed query execution times for Qizx/open. The table cells with a red background indicate an error obtained during the execution of a query (given by the column name) on a document (given by the row name). The links given in these cells lead to the error messages output by XCheck or by the engine.

Figure 5.5: XCheck's HTML output containing: three bar-plots showing the average query execution times per engine, per document and engine, per query and engine; links to plots, one for each types of plots.

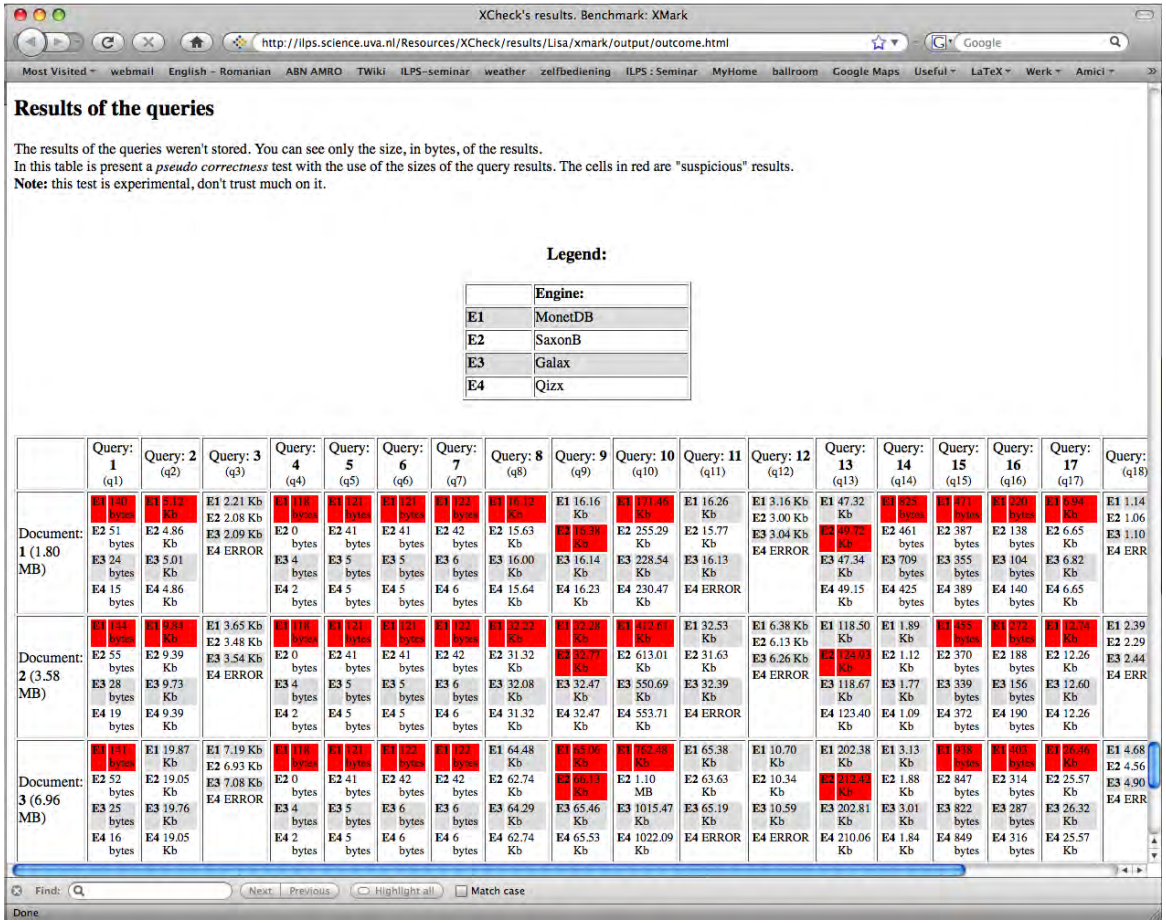


Figure 5.6: XCheck's HTML output containing the sizes (in KB) of the query results. Each cells lists the sizes of the results obtained by the four engines, on the respective query and document. The table cells with red background indicate large deviations of the result size output by an engine from the average query result sizes output by the other engines. In this way, an incorrect query result is signaled.

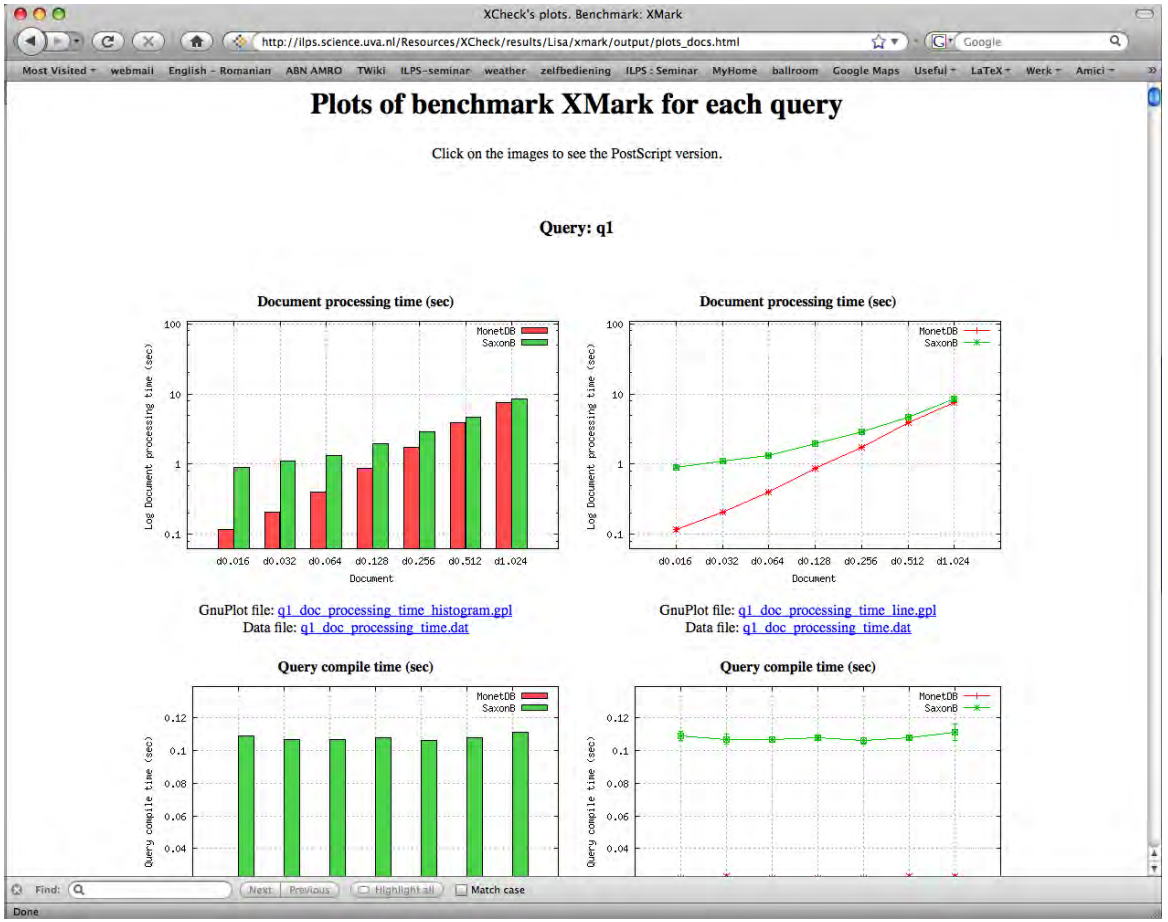


Figure 5.7: XCheck's HTML output containing a gallery of bar and line plots showing the performance times for each query in the experiment per document and engine. For each query and for each time measure used there are two plots presented, a bar and a line plot. Under each plot there are links to the raw data and to the GnuPlot source code that generated the plot. Note that the plots visible in this screenshot show the document processing times only for MonetDB/XQuery and SaxonB. The other two engines, Galax and Qizx/open, do not report these times.

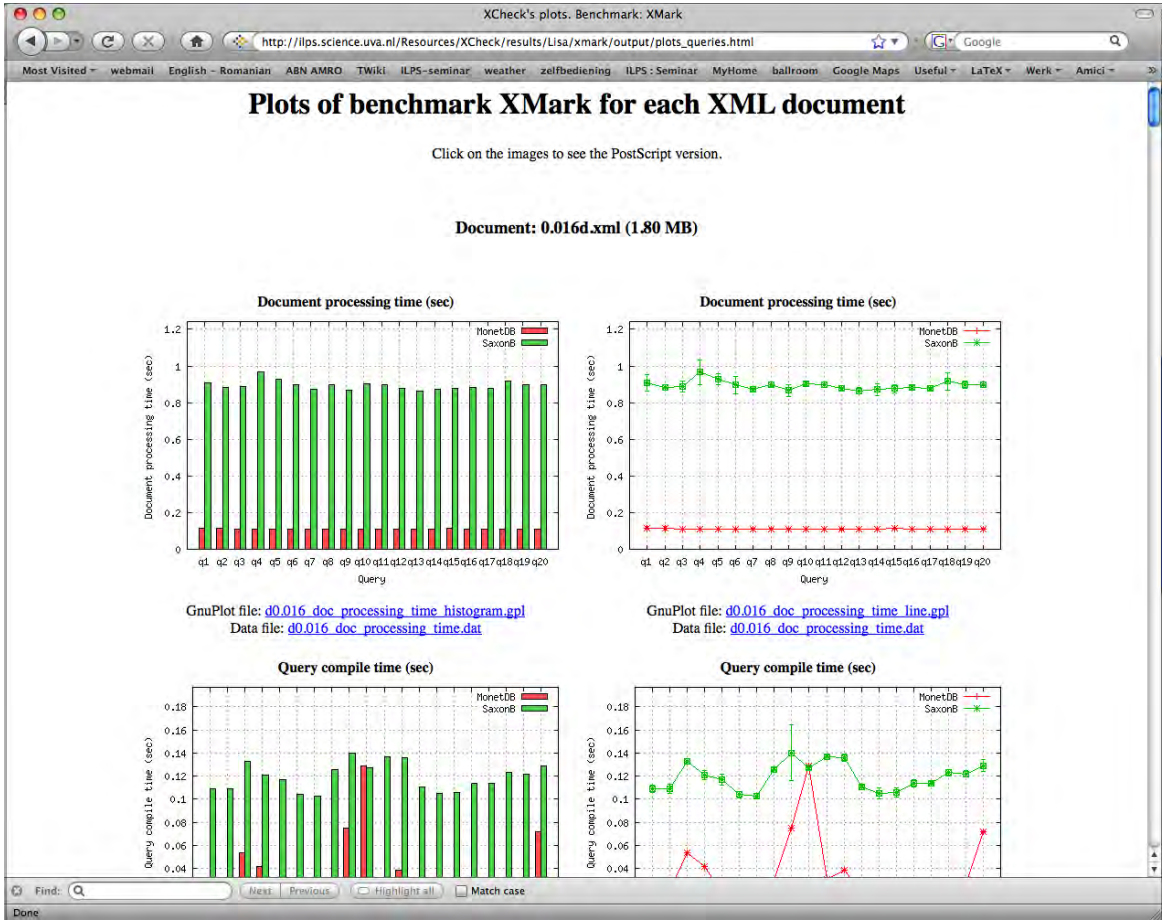


Figure 5.8: XCheck's HTML output containing a gallery of bar and line plots showing the performance times for each document in the experiment per query and engine. For each document and for each time measure used there are two plots presented. Under each plot there are links to the raw data and to the GnuPlot source code that generated the plot. Note that the plots visible in this screenshot show the document processing times only for MonetDB/XQuery and SaxonB. The other two engines, Galax and Qizx/open, do not report on these times.

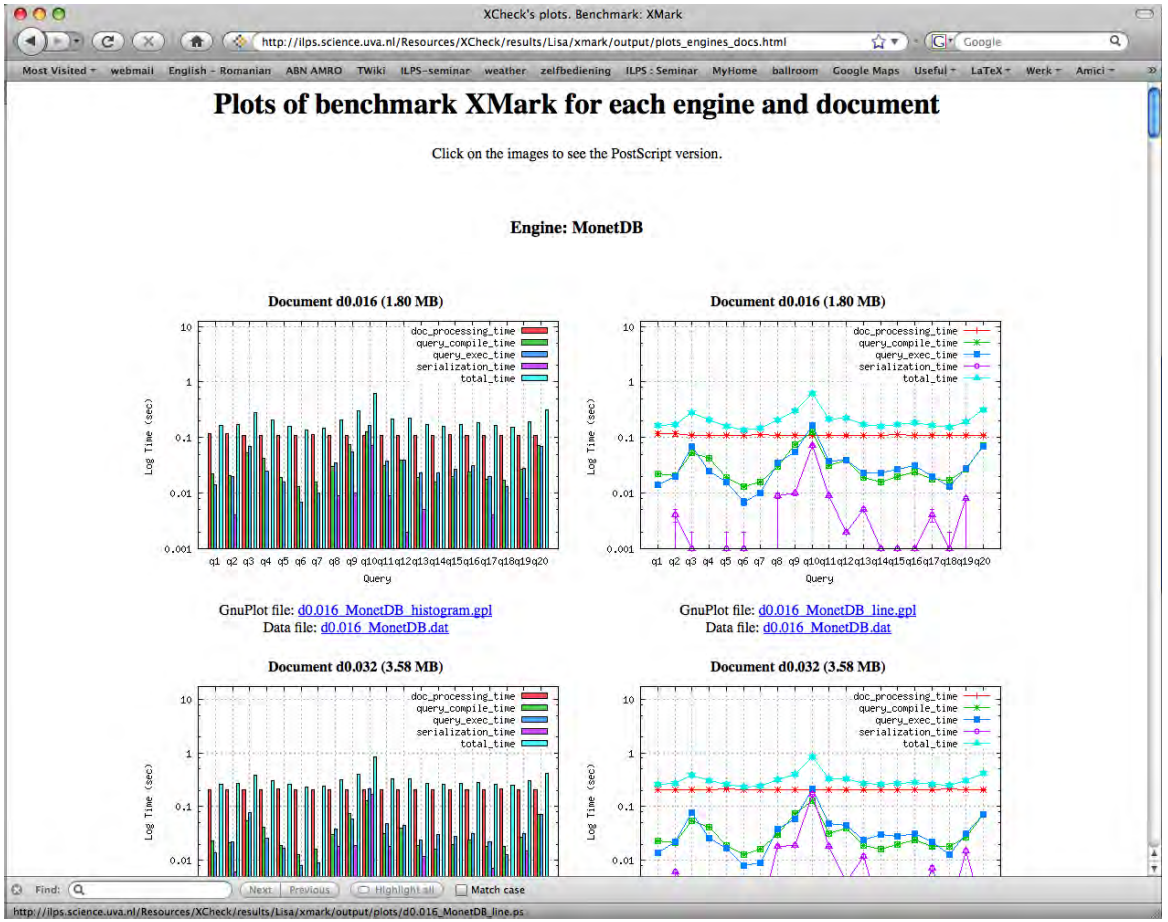


Figure 5.9: XCheck's HTML output containing a gallery of bar and line plots showing the performance times for each engine in the experiment per query and time measure used. For each engine and document in the experiment there are two plots presented. Under each plot there are links to the raw data and to the GnuPlot source code that generated the plot.

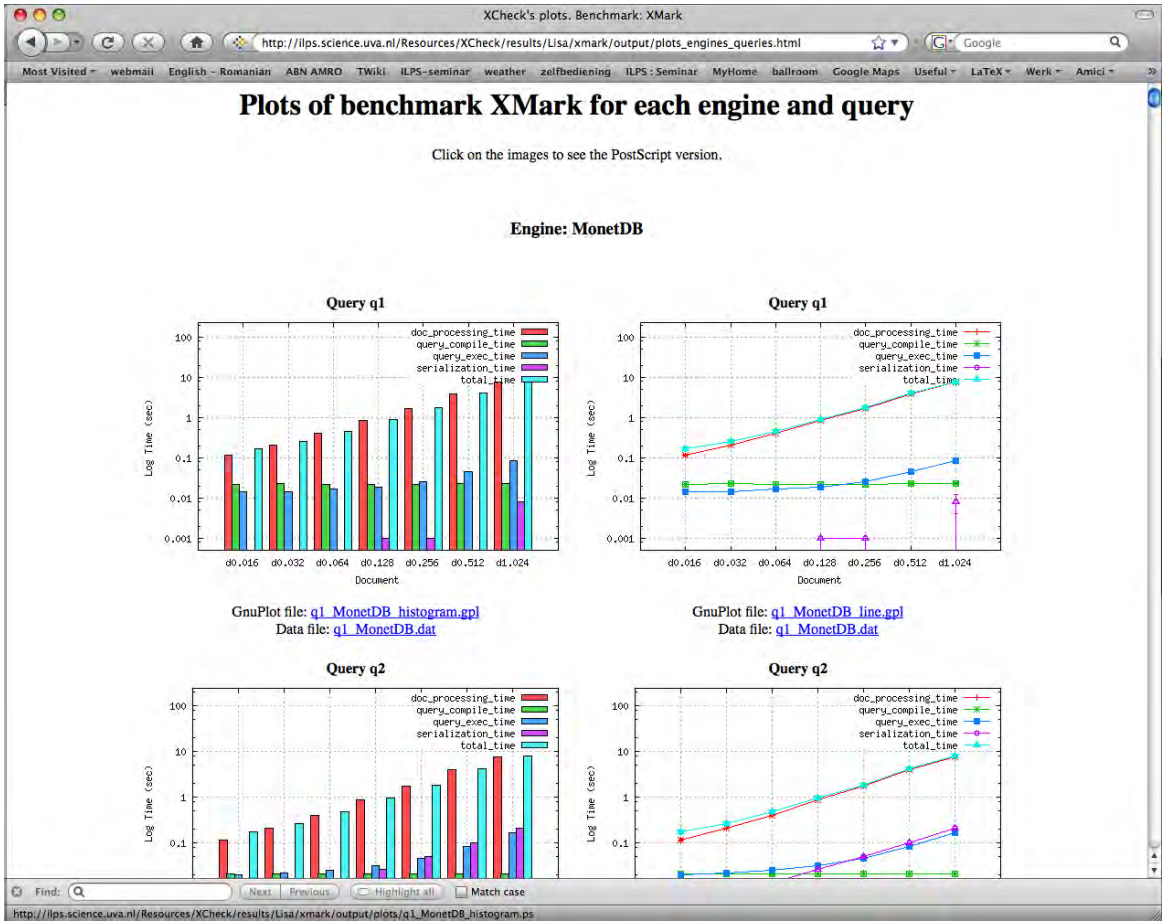


Figure 5.10: XCheck's HTML output containing a gallery of bar and line plots showing the performance times for each engine in the experiment per document and time measure used. For each engine and query in the experiment there are two plots presented. Under each plot there are links to the raw data and to the GnuPlot source code that generated the plot. Note how the query execution time for "q1" gradually grows as the document size increases, and how the total execution time is mostly influenced by the document processing time.

engines. It also shows that Q8-Q12 are challenging queries.

Figure 5.12 shows how the query execution times scale with respect to document size on query Q8. The times are those reported by the engines: MonetDB/XQuery reports CPU time; Saxon and Qizx/open report wall-clock time. The times for Galax are not plotted, since version 0.5.0 of the engine reports incorrect query execution times and we configured the engine adapter for Galax not to store these times. Note that these times cannot be compared directly; in this plot only the slopes of the lines can be compared.

Q8, given below, contains a nested for-loop and a data value join.

```
let $auction := doc("name") return
  for $p in $auction/site/people/person
  let $a :=
    for $t in $auction/site/closed_auctions/closed_auction
    where $t/buyer/@person = $p/@id
    return $t
  return <item person="{$p/name/text()}">{count($a)}</item>
```

The number of `person` and `closed_auction` elements grows linearly with the size of the document. Thus, we can expect the performance of this query to be bound from above by a quadratic function in the size of the document. The lines in Figure 5.12 show a super-linear growth, with SaxonB having the steepest line.

Figure 5.13 shows the detailed performance times (document processing time, query compilation time, query execution time, and total execution time) output by Saxon on all XMark queries and on the document of size 114MB (f=1.024). Note that the document processing time and the query compilation time are constant over all queries; the document processing time dominates the query execution time for all queries but the queries Q8-Q12, thus the total execution time is mostly determined by the document processing time. For the difficult queries, most of the the total time is spent on the query execution, which is 1–2 orders of magnitude larger than the document processing time. This plot also shows that the total time can be an inadequate measure for evaluating query execution techniques, when the total time is dominated by the performance of another engine component.

5.3.2 XCheck's coverage

The current version of XCheck, version 0.2.0, includes adapters for 9 XML query engines listed in Table 5.1: the first column contains the engine's name and reference, the second column contains the engine's version, the third column contains the query language that the engine implements, and the last column contains the detailed execution times that the engine outputs.

XCheck powers the execution of all the performance studies presented in this thesis, including the execution and analysis of 5 XQuery benchmarks on

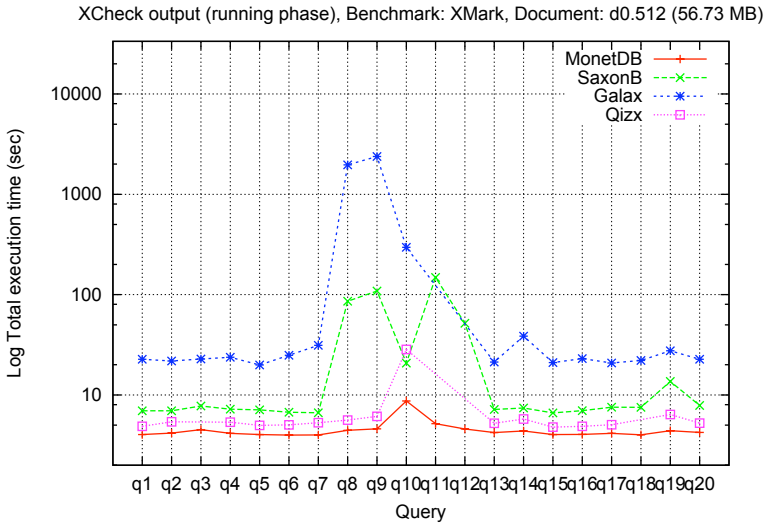


Figure 5.11: Total execution time for XMark queries on a document of size 57MB (f=0.512). The plot presents the CPU time measured in seconds. Missing values: Galax crashes on Q11 and Q12; Qizx/open outputs static type checking errors on Q3, Q11, Q12, and Q18.

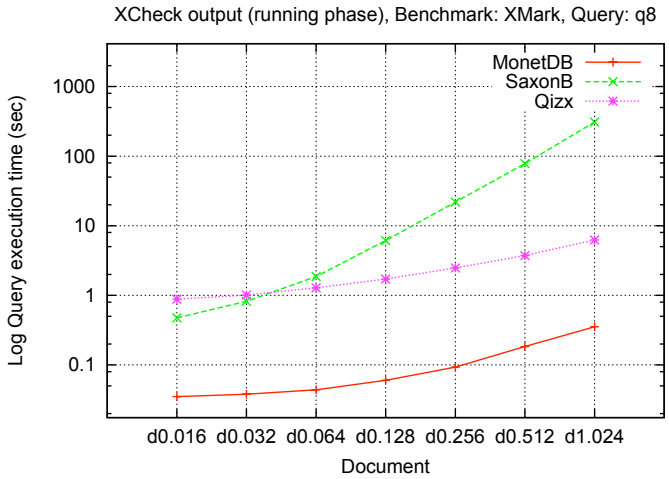


Figure 5.12: Query execution time of XMark query Q8 on documents scaling from 1MB (f=0.016) to 114MB (f=1.024) (note that the document size doubles from one document to the next). The plot shows the times measured by the engines: MonetDB/XQuery reports CPU time; Saxon and Qizx/open report wall-clock time.

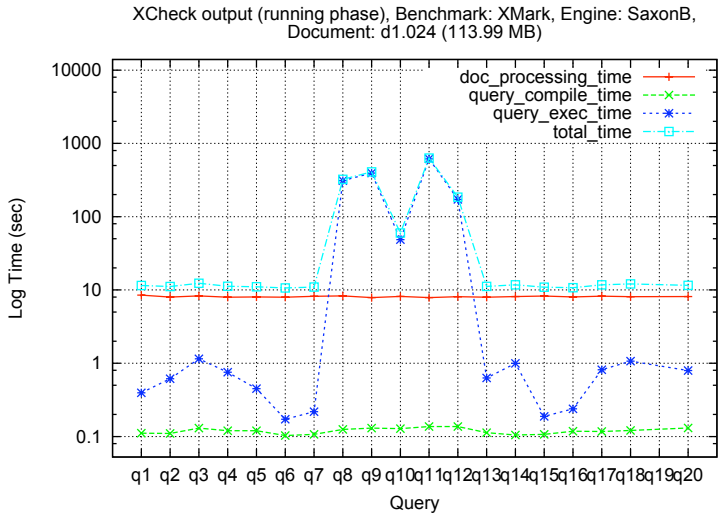


Figure 5.13: Different performance times for SaxonB on a XMark document of size 114MB (f=1.024). The document processing time, query compilation time, and query execution time are output by the engine and measured in wall-clock time, while the total execution time is measured by XCheck in CPU time.

4 XQuery engines discussed in Chapter 3. Another detailed performance analysis of 6 XQuery engines on these 5 benchmarks using XCheck is presented in [Manegold, 2008]. This experiment consisting of running 6 engines on 5 benchmarks, with a total of 3720 measurements (running one engine on one query and one document), takes approximatively 2 weeks to run on a commodity PC. Such large scale experiments are only possible with the help of an automated tool like XCheck.[2]

5.4 Related systems

At the time of development of XCheck, there were two other open source automated testing platforms for evaluating XML query engines, BumbleBee [BumbleBee, 2006] and XSLTMark [XSLTMark, 2006].[3] BumbleBee is a test harness for evaluating XQuery engines and for validating queries expressed in the XQuery language. Although it measures the total execution times its main goal is to test an engine's compliance with the XQuery language specification. The application can execute user defined tests containing reference answers for the correctness check. XSLTMark is a similar application for XSLT processor performance and

[2] As of September 2009, a Google Scholar search shows 9 citations for [Afanasiev *et al.*, 2006], the paper presenting XCheck.

[3] As of September 2008, both testing platforms are no longer available.

Engines	Version	Query language	Times
SaxonB [Kay, 2009]	8.7	XQuery	D, QC, QE, T
Galax [Fernández *et al.*, 2006]	0.5.0	XQuery	T
MonetDB/XQuery [Boncz *et al.*, 2006b]	0.10.3	XQuery	D, QC, QE, S, T
Qizx/open [Axyana Software, 2006]	1.0	XQuery	QE, S, T
eXist [Meier, 2006]	1.0	XQuery	T
Qexo [Qexo, 2006]	1.8.1 alpha	XQuery	T
Blixem [University of Antwerp, 2006]	16 Jun 2005	LiXQuery	T
XmlTaskForce [Koch, 2004]	30 Sep 2004	XPath 1.0	T
Arb [Koch, 2006]	unknown	CoreXPath	D, QE, T

where D=*document processing time*, QC=*query compile time*, QE=*query execution time*, S=*serialization/output time* and T=*total time*.

Table 5.1: The list of engines for which XCheck provides adapters.

compliance benchmarking. It comes with a collection of default test cases that are performance oriented. XSLTMark outputs more information about the execution, such as (document) preprocessing time, total execution time, the size of engine input/output, the engine's throughput in size per second, and the correctness check. Neither Bumblebee nor XSLTMark processes an engine's output for extracting detailed times or error messages.

In comparison with BumbleBee, XCheck targets the execution of user defined performance tests rather than correctness tests. Although XCheck does not perform a proper correctness test, it implements a pseudo test by comparing the size of the query results of several engines relative to each other. In comparison with XSLTMark, XCheck is a generic testing platform that targets XML query engines, rather than XSLT engines. On top of this, XCheck is based on a more flexible input/output adapter design than either of the two platforms. This adapter design allows users to customize what information output by the engines the platform is documented. Moreover, XCheck performs a statistical analysis of the data and outputs graphs, facilitating interpretation of the results.

5.5 Summary and conclusion

The question we pursued in this chapter (Question 5.1) is whether a generic (engine and benchmark independent) tool for running performance benchmarks is feasible and what design choices need to be made in order to build it. The realization and success of XCheck is a clear answer to this question.

During the development of XCheck we had to address several issues. First, we had to decide how XCheck should communicate with the tested engines. The command line adapter design that XCheck implements is elegant and easily implementable—many of the XML query engines have a command line interface. Second, we had to decide what atomic measure XCheck should implement. Cur-

rent XQuery benchmarks measure performance times of a set of queries on a set of documents/collections, where the atomic measure is the performance time of processing one query on a document/collection. If the engines provide more detailed performance times, e.g., document processing, query compilation, etc., XCheck also collects these times. Third, we had to decide on how to obtain accurate performance measures. XCheck computes the atomic measure $N + 1$ times and takes the mean and the standard deviation of the last N runs. Finally, we had to decide how to store and present the performance results. XCheck uses XML to store the raw measurement data, and it uses HTML and plots to present it to the user in an easily readable format.

We use XCheck to execute all experimental studies presented in this thesis. In Chapter 3, we use XCheck to execute existing XQuery benchmarks; in Chapters 6 and 7, we use XCheck to execute two micro-benchmarks; and in Chapter 8, we use XCheck to evaluate the performance of a proposed optimization technique for recursion in XQuery. All experiments are conducted on multiple XQuery engines.

Chapter 6

A Repository of Micro-Benchmarks for XQuery

In Chapter 3, we identified a need for precise and comprehensive tools for experimental evaluation, as well as a need for general methodology for experimental evaluation. In this chapter, we propose a micro-benchmarking methodology, as well as a repository of micro-benchmarks for XQuery, called MemBeR. First, we refresh our findings from Chapter 3 that motivate our work (Section 6.1). Then, we describe the micro-benchmarking methodology associated with MemBeR (Section 6.2). Further, we describe the MemBeR repository of micro-benchmarks (Section 6.3). To illustrate the MemBeR methodology, we also give an example micro-benchmark (Section 6.3.1). Finally, we discuss the benefits and weaknesses of our approach and conclude (Section 6.4).

This chapter is based on work previously published in [Afanasiev *et al.*, 2005a].

6.1 Introduction

The existence of suitable performance evaluation tools is imperative for the development of the XML processing engines. Performance benchmarks have proven to be a successful catalyst for the development of relational databases [Jain, 1991]. Since the introduction of XML and its query languages, many XML benchmarks have been proposed. In Chapter 3, we surveyed and analyzed five XQuery benchmarks publicly available in 2006: the Michigan benchmark (MBench) is a *micro-benchmark suite*, while XMach-1, XMark, X007, and XBench are *application benchmarks*. Among other questions, we investigated how the benchmarks are used in the database research community and whether they can be used for in-depth analysis of XML query processing techniques.

As a result of surveying scientific articles on XML query processing, we observed that the benchmarks are rarely used for performance evaluations of presented research (in less than 1/3 of the surveyed articles). Instead, ad-hoc exper-

imental evaluations are used. While the empirical evaluations used in scientific articles are focused on a particular language feature or query processing technique, the application benchmarks aim at the evaluation of a whole system in a particular application scenario. Our analysis showed that application benchmarks are not suitable for detailed and systematic evaluations of query processing techniques. The micro-benchmark suite MBench targets evaluation of XQuery language features in isolation and it is a good starting point for detailed analysis; however, it provides a query workload that is insufficient for a systematic and conclusive evaluation with respect to the tested language feature. Based on this, we concluded that the development of XML query engines is being held back by a lack of systematic tools and methodology for evaluating performance of query processing and optimization techniques.

The first problem we are facing is the lack of performance assessment tools allowing system developers and researchers to obtain *precise* and *comprehensive* evaluations of XML query processing techniques and systems. An evaluation is *precise* if it explains the performance of a processing technique or a system component on one language feature *in isolation*. In other words, it allows us to obtain an understanding of which parameters impact the performance of the target component on the target feature without "noise" in the experimental results due the performance of other components or other features. An evaluation is *comprehensive* if it considers *all* parameters that may impact the performance of the evaluation target and it explains the impact of every important parameter-value pair in a systematic way.

Second, a *standard methodology* is needed, explaining how to choose or develop appropriate performance evaluation tools for a given target, how to choose the parameters that are likely to be important, how to choose the value combinations for these parameters, and how to analyze the results. A standard methodology brings many benefits. It eases the task of performance evaluation. It also reduces the effort spent on experimental design, on dissemination of experimental results and comparison.

The research question that we address in this chapter is:

6.1. Question. *What is a suitable methodology for precise and comprehensive performance evaluation of XML query processing techniques and systems?*

As an answer to this question, we are proposing MemBeR, a structured repository of micro-benchmarks and related methodology. The micro-benchmarks target XML query processors on XML query language features in isolation (the main focus is on XQuery and its fragments). We find that micro-benchmarks are the most fitting tools for a precise and comprehensive performance evaluations. We also endow the repository with a micro-benchmarking methodology for facilitating the correct usage and creation of suitable micro-benchmarks. MemBeR is intended mainly for system developers and researchers, to help them analyze and optimize their techniques and systems.

Given the wide range of interesting XQuery features, ongoing development of XML processing engines, and ongoing developments of language extensions, such as full text search [World Wide Web Consortium, 2009a] and XML updates [World Wide Web Consortium, 2009b]), a fixed set of micro-benchmarks devised today is unlikely to be sufficient and/or relevant in the future. Thus, we develop the repository as an *open-ended community effort*:

- repository users can contribute by *creating* new micro-benchmarks or by *enhancing* existing ones,
- quality control is guaranteed by a *peer-review process*, verifying that the proposed addition or change adheres to the micro-benchmarking methodology and it is not yet covered by the repository content.

MemBeR allows for continuous addition and improvement of micro-benchmarks, also targeting new performance challenges coming from applications and architectures perhaps not yet available today. In this manner, we hope that MemBeR will grow to provide a complete coverage of XQuery language features.

With MemBeR we aim to consolidate the experience of individual researchers that spend time and effort in designing micro-benchmarks for performance evaluation of their query optimization and processing techiniques. We hope MemBeR will provide the necessary performance evaluation tools and methodology and will be widely used in the XML data management community.

MemBeR has a web-based interface and it is freely accessible at `http://ilps.science.uva.nl/Resources/MemBeR/`. Currently, MemBeR consists of 8 registered users, 5 contributors, and it contains 34 micro-benchmarks targeting different XQuery language features.

In the next section, we describe the MemBeR micro-benchmarking methodology.

6.2 The MemBeR micro-benchmarking methodology

Our goal is to build a repository of micro-benchmarks for studying the performance of XQuery processing techniques and engines. We identify four aspects of performance:

Efficiency: how well does a system perform, e.g., in terms of completion time or query throughput? The primary advantage of a data management system, when compared with an ad-hoc solution, should be its efficiency.

Resource consumption: a system's efficiency should be naturally evaluated against its resource needs, such as the size of a disk-resident XML store, with

or without associated indexes; the maximum memory size required by a streaming system, etc.

Correctness: does the output of the system comply with the query language specifications? For a complex query language such as XQuery, and even its fragments, correctness is also a valid target for benchmarking.

Completeness: are all relevant language features supported by the system? Some aspects of XQuery, such as its type system, or its functional character, have been perceived as complex. Correspondingly, many sub-dialects have been carved out [Hidders *et al.*, 2004, Miklau and Suciu, 2002, Paparizos *et al.*, 2004]. Implementations aiming at completeness could use a yardstick to compare against.

In this chapter, our focus is mainly on benchmarks for testing efficiency and resource consumption. Nevertheless, we stress the importance of the other measures for a correct interpretation of performance. For devising correctness and completeness benchmarks, one can build on top of the XML Query Test Suite (XQTS) [World Wide Web Consortium, 2006b]. Alhough XQTS is not officially meant to test for an engine's compliance to the XQuery standard, it is the best compliance test available today.

6.2.1 Micro-benchmark design principles

A well designed micro-benchmark is one for which the analysis of results is straightforward and the impact of each benchmark parameter is clear. In the following, we list the design principles for micro-benchmark creation that we adopt for MemBeR.

In some sense, these principles can be seen as refining our view on what micro-benchmarks are. Recall that [Runapongsa *et al.*, 2002] were the first to propose micro-benchmarking in the context of XQuery. Two of the principles below, P1 and P3, were already implicit in [Runapongsa *et al.*, 2002], and played a role in the design of MBench. The fourth principle, P4, was inspired by our analysis of MBench in Chapter 3, where we showed that the queries of MBench that are intended to test join processing are not sufficient for a detailed analysis since they vary several parameters simultaneously. All other principles are introduced here and we have not been able to trace them back to earlier literature on XML benchmarking. Note that P6 is not a design principle for individual micro-benchmarks, but rather concerns the structure of the entire micro-benchmark repository.

P1: A micro-benchmark should reduce to a minimum the influence of all but the tested system functionality and language feature. This can be achieved by designing a focused workload. For example, the presence of an XML Schema for the input document enables a large number of optimizations, at the level of an XML store and indices, at the level of XQuery rewriting and optimization, etc.

For any micro-benchmark whose target is not on schema-driven optimizations, one should use documents without a schema. Otherwise, schema-driven optimizations might effect the system's performance in a non-transparent manner and make results uninterpretable. If the purpose is to test path expressions navigating downward, the queries should not use sibling navigation, and vice versa.

P2: A micro-benchmark should (also) measure individual query processing steps. To get an accurate evaluation of a system component or functionality, it is often required to measure individual processing steps, such as query normalization, query rewriting, query optimization, data access, output construction, etc. For instance, XPath micro-benchmarks may measure the time to *locate* the elements that must be returned (this often means finding their IDs). Measuring such processing steps might require hooks into the tested engine. Even if the workload is carefully designed to trigger one system component and reduce the influence of the others, the total query execution times may still reflect the impact of too many factors.

P3: A micro-benchmark should strive to explicitly list and provide value ranges for all document, query, and other system or environment parameters that may impact the performance results. This is important in order for the benchmark results to be interpretable, reproducible, and comprehensive. In this way, a micro-benchmark provides well-documented measures.

At the time of creating a micro-benchmark, the creator most likely has a particular testing scenario in mind. Typically, the effort needed to invest in covering other testing scenarios as well is considerable, which makes P3 difficult to adhere to in practice. Nevertheless, we stress the importance of the principle.

P4: Whenever possible, micro-benchmark measures should vary one benchmark parameter at a time. This allows for analyzing the impact of each parameter on the performance measure. A micro-benchmark should aim at explaining the target's performance in terms of the impact of the benchmark parameters. In this way, a micro-benchmark provides systematic measures.

The above implies that for any micro-benchmark measure and any data parameter likely to impact the measure's result, *at least one data set can be constructed by controlling the value of that parameter in its interesting range.* This has an impact on the choice of data sets (see Section 6.2.5).

P5: A micro-benchmark should be extensible. A micro-benchmark should aim to remain useful even when systems undergo substantial development and achieve higher performance. The benchmark parameters should therefore allow for a wide enough range of values. The micro-benchmarks should also be regularly updated to reflect new performance standards.

P6: The number of micro-benchmarks for any given language feature in the repository should be kept to a minimum. This principle is meant to keep the repository focused. Instead of having two micro-benchmarks targeting two different aspects of the same language feature, the difference could be captured by a parameter in a single unified micro-benchmark. Still, there should be a balance

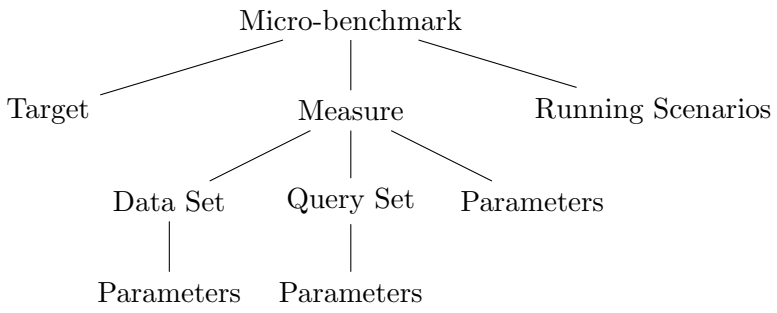


Figure 6.1: MemBeR micro-benchmark high-level hierarchical structure.

between P6 on the one hand and simplicity and ease of interpretation on the other hand.

6.2.2 Micro-benchmark structure

Following the design principles stated above, we propose a more detailed specification of MemBeR micro-benchmarks.

A *MemBeR micro-benchmark* is an experimental tool for the performance evaluation (efficiency, consumption, correctness, or completeness) of a given component or functionality of an XML query processing system on a query language feature *in isolation*. The feature that the micro-benchmark studies (e.g., the system functionality, such as the query optimizer, on a language feature, such as structural joins) it is called the *target* of the micro-benchmark. The target also specifies the system type(s) and scenario(s) for which the micro-benchmark is proposed, for example a persistent database scenario or streaming scenario, etc.

The micro-benchmark includes a *measure*, which is a function of a parametrized workload and of other input parameters (e.g, parameters of the targeted system, or of the experimental set-up).

The *workload* consists of parametrized an *XML data set* and *queries*. The *XML data set* is parametrized by XML data characteristics that might impact performance results, such as document size, tree-depth, element fan-out, etc. (For a comprehensive list of XML data characteristics see Section 2.1.2.) The value ranges for these parameters (and other relevant information, such as value distribution, value units, etc.) are provided. The data set can be empty, since XML fragments are legal XQuery expressions and thus an XML query may carry "its own data." It might have an associated schema (e.g., DTD, XML Schema) or not.

The *query set* is characterized by its query language (e.g., XPath 1.0, XQuery 1.0). They can also be parametrized by query characteristics that might impact performance results, such as number of steps in a path expression query, numbers of query nesting levels, selectivity of a value selection predicate, etc. The

value ranges for these parameters (and other relevant information, such as value distribution, value units, etc.) are provided. The query set can also be empty, e.g., a benchmark that targets document pre-processing in the persistent database scenario does not contain queries.

Finally, a micro-benchmark is endowed with *running scenarios* that are guidelines on how to execute the benchmark and interpret the results. A running scenario specifies how to vary the values of one benchmark parameter in order to determine its impact on performance. Intuitively, it yields a family of curves where the varying parameter values are on the x axis and the measurements are on the y axis. A curve corresponds to a parameter-value configuration of the background parameters. The number of running scenarios in a micro-benchmark depends on the number of benchmark parameters.

Note that this micro-benchmark structure differs slightly from the structure presented in [Afanasiev *et al.*, 2005a]. The differences concern only the presentation and are not conceptual. Specifically, we use a hierarchical model while in [Afanasiev *et al.*, 2005a], an Entity Relationship model was used.

6.2.3 Micro-benchmarking methodology

In order to facilitate micro-benchmark execution and interpretation of results, we provide a list of general guidelines that comprises MemBeR benchmarking methodology.

When executing a micro-benchmark and analyzing results, *the benchmark parameters should vary one at a time, while keeping the other parameters constant.* This typically yields a family of curves where the varying parameter values are on the x axis, and the measured results on the y axis.

A micro-benchmark can have many parameters and the space of all parameter-value pairs, and thus measurements, can be huge (e.g., a measure involving 7 parameters with at least 3 values each that generates minimum $3^7 = 2187$ measurements). Performing and analyzing all the measurements might be not feasible. In such cases, at least *the measurements for all end-of-range parameter values should be provided.* Trying the measure with these values may give the system developer early feedback, by exposing possible system shortcomings. More measurements can be added to the analysis in a pay-as-you-go fashion further clarifying the initial results.

Micro-benchmark executions that vary less parameters than specified by the micro-benchmark require proper motivation. Neglecting parameters without further explanation compromises the correct interpretation of benchmark results and the comprehensiveness of the evaluation. *Changes or restrictions of the parameter value-ranges should also be motivated*, while testing extra values for a parameter is encouraged. In the case that the measurements obtained on the extended value-range reveal a significant difference in performance results in comparison with the measurements obtained on the original value-range, a revision of the

micro-benchmark should be considered.

Determine and declare all hidden parameters that may impact the benchmark results. Besides the parameters that the micro-benchmark explicitly varies, there might be hidden parameters that impact the benchmark results. For example, many efficient query processing techniques are conditioned by some underlying language simplifications, such as: unordered semantics, simplified atomic types set, unsupported navigation axes, unsupported typing mechanism, etc. Such simplifications can be considered as hidden parameters and their values should be made clear when interpreting benchmark results. *When reporting results, declare the language and/or dialect supported by the system, even for features not used by the micro-benchmark.*

6.2.4 Preliminary classification of micro-benchmarks

The micro-benchmarks can be organized in the repository conform their target and measure. In this section, we outline a general classification of micro-benchmarks. This classification guides a user looking for a specific micro-benchmark and serves as a road map for our ongoing micro-benchmark design work.

A first micro-benchmark classification criterion distinguishes between *efficiency*, *consumption*, *correctness*, or *completeness*. Further, we can classify micro-benchmarks according to the following criteria:

- The metric used by the benchmark measure, for example *execution time*, *query normalization or optimization time*, *query throughput*, *memory occupancy*, *disk occupancy*, etc. It may also be a simple boolean value, in the case of correctness measures.

- Benchmarks may test *data scalability* (fixed query on increasingly larger documents) and/or *query scalability* (increasing-size queries on fixed documents).

- Whether or not a micro-benchmark uses an *XMLSchema*; the particular schema used.

- The *type of the targeted engine*, such as: persistent database (store the document once, query it many times), streaming (process the query in a single pass over the document), and main-memory (the document is parsed and queried entirely in main-memory).

- The *targeted query language* and perhaps dialect which must be supported in order to run the micro-benchmark.

- The *targeted language feature* in a micro-benchmark is a precise classification criteria. We strive to provide exactly one micro-benchmark for each interesting feature.

There might be other classification criteria as the repository grows. Similarly to micro-benchmark principle P5, the repository should be easily extensible and its organization should also be regularly updated to provide the best classification of micro-benchmarks.

6.2.5 Data sets for micro-benchmarking

In order to satisfy micro-benchmark principles (P1, P3, P4, and P5), micro-benchmarks must provide parametrized data sets that allow for systematic varying of its parameters. This can be achieved only with synthetic data sets. Synthetic data generators are suitable for obtaining easily customizing test data. Nevertheless, coming up with one unified data set, even a synthetic one, on which all important characteristics can be varied at will, is hardly feasible. Notice that some parameters, such as size, depth, and fan-out are inter-related and thus cannot be independently controlled.

In this section, we consider two broad classes of synthetic documents for the MemBeR repository. Documents in the first class are schema-less and allow full control over the basic XML document characteristics. We propose a syntactic document generator for this class. Documents in the second class are schema-driven. We propose using existing declarative document generators for obtaining data sets in this class. These classes of documents and their generators are easy-to-use solutions for obtaining data sets for micro-benchmarking and we believe that they cover the basic needs for data sets for MemBeR. Nevertheless, other data sets can always be added to the repository.

Below, we briefly describe the two classes of documents.

Schema-less parametric data generator

For the class of schema-less documents, we propose a synthetic data generator that allows controlling: (*i*) the maximum node fanout, (*ii*) maximum depth, (*iii*) total tree size (number of elements), (*iv*) document size (disk occupancy), (*v*) the number of distinct element names in the document, and (*vi*) the distribution of tags inside the document. Out of these, the following parameters are required: (*i*) either tree size or document size; and (*ii*) either depth or fan-out.

The number of distinct element names is 1 by default; elements are named `a1`, `a2` etc. The distribution of element tags within a document can be controlled in two ways. *Global* control allows tuning the overall frequency of element named `a1`, `a2`, ..., `a`*n*. Labels may nest arbitrarily. Uniform and normal distributions are available. *Per-tag* control allows specifying, for every element name `a`*i*, the minimum and maximum level at which `a`*i* can appear may be set; furthermore, the relative frequency of `a`*i* elements at that level can be specified as a number between 0.0 and 1.0.[1] Global distributions allow generating trees where any `a`*i*

[1] The generator checks the frequencies of several element tags at a given level for consistency.

may appear at any level. Close to this situation, for instance, is the Treebank data set,[2] containing annotated natural language; tags represent parts of speech and can nest quite freely. Per-tag distributions produce more strictly structured documents, whereas, e.g., some names only appear at level 3, such as `article` and `inproceedings` in the DBLP data set,[3] other elements appear only below level 7, such as `keywords` in XMark etc.

Fan-out, depth, and tag distribution have impact on different aspects of XML query processing. For example, they have an impact on the disk occupancy of many XML storage and structural indexing schemes, on the complexity and precision of XML statistical synopses, on the size of in-memory structures needed by an XML stream processor, and on the performance of path expression evaluation for many evaluation strategies. Thus, we will rely on this data set for varying these parameters and for assessing such aspects.

The number and size of text values follow uniform or normal distributions. Values can be either filled with random characters, or taken from the Wikipedia text corpus (72 MB of natural language text, in several languages). The latter is essential in order to run full-text queries.

Schema-derived data sets

For the class of schema-derived data sets, we propose using ToXGene [Barbosa *et al.*, 2002], a declarative data generator that produces synthetic data according to schema specifications of the desired test data. ToXGene relies on XML Schema specifications. It allows for the specification of skewed probability distributions for elements, attributes, and textual nodes. Being based on XML Schema, the tool supports different data types, as well as id/idref constraints. It also offers a simple declarative query language that allows one to model relatively complex dependencies among elements and attributes involving arithmetic and string manipulation operations. For instance, it allows one to model that the total price of an invoice should be the sum of the individual prices of the items in that invoice multiplied by the appropriate tax rates. Finally, ToXgene offers support for generating recursive XML documents.

6.3 The MemBeR repository

In this section, we discuss the current version of the MemBeR repository and give an example of a MemBeR micro-benchmark. We then discuss how well the example meets the micro-benchmarking principles P1–P6.

A web-based interface to the MemBeR repository is located at `http://ilps.science.uva.nl/Resources/MemBeR/`. The current version of MemBeR has the

[2] Available at `http://www.cs.washington.edu/research/xml/datasets`.
[3] Available at `http://dblp.uni-trier.de/xml`.

following components:

- *Micro-benchmarks* organized in categories based on the preliminary micro-benchmark classification presented in Section 6.2.4;
- *Micro-benchmark results* corresponding to the respective micro-benchmark and contributed by the repository users;
- The *synthetic XML data generator* presented in Section 6.2.5;
- *Repository users* that contribute micro-benchmarks and micro-benchmark results; and
- A list of *XQuery benchmarking resources*, such as links to the five XQuery benchmarks discussed in Chapter 3; the corrected and standardized queries of the five benchmarks; a list of open-source XQuery engines; link to XCheck, the tool for automatizing the process of running a benchmark and analyzing the results presented in Chapter 5, etc.

Currently, MemBeR contains 34 micro-benchmarks, contributed by 5 out of 8 registered users. It also contains micro-benchmark results corresponding to 14 micro-benchmarks.

6.3.1 An example of MemBeR micro-benchmark

In this section, we give an example of a MemBeR micro-benchmark created and published by Manolescu, Miachon, and Michiels. The benchmark can be found at: `http://ilps.science.uva.nl/Resources/MemBeR/CHILD-ATTRIB.html`. This micro-benchmark is part of a family of seven MemBeR micro-benchmarks targeting the performance XPath child navigation.[4] We choose this example for its simplicity and in order to illustrate the micro-benchmarking design principles at work. In the next section, we discuss the extent to which the benchmark meets the MemBeR design principles.

In the next chapter, Chapter 7, we present a more comprehensive micro-benchmark (with respect to the number of tested parameters) for evaluating value-based joins processing techniques. For even more examples of MemBeR micro-benchmarks we refer to the MemBeR website and related publications [Afanasiev *et al.*, 2005a, Manolescu *et al.*, 2008b].

Target

The performance of child axis navigation of any type of XML query engine.

[4] In compliance with the design principle P6, these micro-benchmarks should be combined in one if possible.

Measure

The benchmark measure is the query processing time as a function of the benchmark parameters. The benchmark varies two parameters: the *number of child steps* in a path expression and the *child step selectivity*. The number of child steps varies from 1 to 19 and it is controlled by a query set of 19 queries, one for each path depth. The child step selectivity varies from 1 to 2^{18} items in an exponential manner and it is controlled by setting the fan-out of one of the benchmarks XML documents to 2. Note that the child step selectivity is also responsible for the size of the context sequence of any intermediate child step.

The measure can be seen as *query scalability* with respect to the number of child steps in a path expression and with respect to query intermediate and end result size. Controlling both the path depth and the query selectivity at each depth is important, since these parameters have important independent impacts on the performance of child-navigation queries.

The unit of measurement is CPU seconds.

Data set

The data set consists of two documents, "layered.xml" and "exponential2.xml", constructed with the MemBeR synthetic data generator. The first document is of size 12.33 MB. The root of the document is labeled "t1", and it has 32,768 children labeled "t2". At every level i comprised between 3 and 19, there are 32,768 nodes labeled "ti". Each element labeled "ti", with $3 \leq i \leq 18$, has exactly one child labeled "t$(i+1)$". Elements labeled "t19" are leaves.

The second document is a binary tree of size 11.39 MB. At level i (where the root is considered level 1), the document has $2^{(i-1)}$ elements labeled "ti". Elements labeled "t19" are leaves. Every element of both documents have a unique attribute "@id".

The two documents have the same *tree depth*, *size*, *element-name distribution over the tree layers*, etc. They differ only in the *average tree fan-out*—the first document has an average tree fan-out 1.06, while the second document has tree fan-out 2.

Note that the tree shapes of "layered.xml" and "exponential2.xml" are extremely regular; real-life documents are likely to be somewhere in between them in terms of average tree fan-out.

Query set

The query set consists of nineteen queries of the form

`doc(`*name*`)/t1/t2/.../t`*n*`/data(@id)` ,

where n varies from 1 to 19 and *name* is "layered.xml" or "exponential2.xml". The queries retrieve the IDs of nodes at increasing depths of the document. Note

that all the queries have non-empty results. Query selectivity depends on the document against which the query is evaluated and on the number of child steps. When evaluated on "layered.xml" the query selectivity is constant for all the queries; when evaluated on "exponential2.xml" the query selectivity is exponential in the number of child steps.

The attribute step at the end of the child path is made in order to reduce the result serialization time. In this manner, the query processing time gets closer to the actual time it takes the query engine to locate the nodes.

This query is designed to test the ability of the query processor to deal with increasing lengths of child-path expressions and with increasing intermediate and end result size. For instance, materialization of intermediate path results has an increasing performance impact for longer queries.

Running scenario

The benchmark has one running scenario *varying the number of child steps* while fixing the child step selectivity by fixing the document on which the queries run. The results are represented by two curves, one for each document in the data set.

For assessing the query selectivity with respect to the number of child steps, the benchmark measures the query processing time of the query set on "layered.xml". Since the number of returned nodes is constant (2^{16}) for all queries on this document, the performance time might be influenced only by the number of child steps. The number of nodes visited by a naive XPath evaluation strategy is in $O(n)$, thus the processing times of a query engine should behave, in the worst case, as a linear function of the number of child steps.

For assessing the query selectivity with respect to the intermediate and end result size, the benchmark measures the query processing time on "exponential2.xml". The number of returned items is exponential in the number of child steps. The number of nodes visited by a naive XPath evaluation strategy is also in $O(2^n)$, thus the processing times of a query engine should be behave, in the worst case, as a exponential function of the number of child steps.

Benchmark results

To illustrate the benchmark in action, we execute it on four open-source XQuery engines: SaxonB v9.1 [Kay, 2009], Qizx/Open v3.0 [Axyana Software, 2009], Galax v0.5.0 [Fernández *et al.*, 2006], and MonetDB/XQuery v0.30.0 [Boncz *et al.*, 2006b].

The experiments are conducted on a Fedora 8 machine, with a 64 bit compilation, with 8 CPUs, Quad-Core AMD Opteron(tm) of 2.3GHz, and 20GB RAM. When running the Java implementations, SaxonB and Qizx/Open, we set the Java Virtual Machine maximum heap size value to 10GB. The experiments are run with XCheck, the testing platform presented in Chapter 5. The time mea-

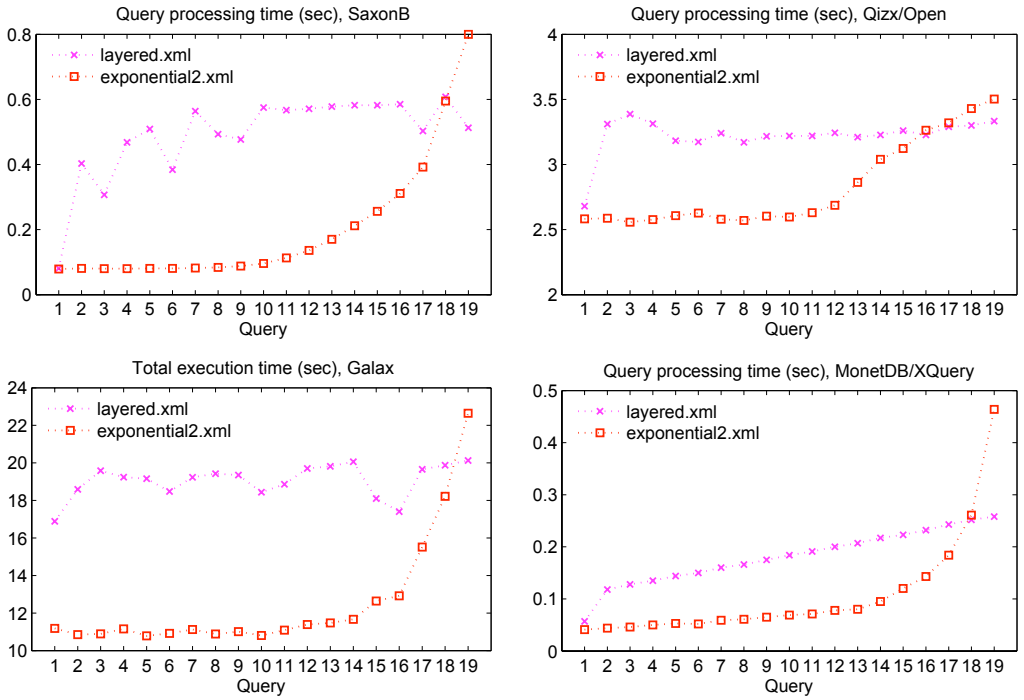


Figure 6.2: Micro-benchmark results on four XQuery engines.

surements are computed by running each query 4 times and taking the average performance time(s) of the last 3 runs. SaxonB, Qizx/Open, and MonetDB report on detailed processing times, including the *query processing time*. Thus, conform the benchmark specification, we record the query processing time reported by the engines. Galax does not report on detailed times, thus we measure the total query execution time.

Figure 6.2 contains the micro-benchmark results for the four engines. On "layered.xml", MonetDB/XQuery shows a linear increase in performance times with respect to the number of child steps. The other engines, though their corresponding curves are more irregular, do not show a steady linear increase of performance times. On "exponential2.xml", SaxonB, Galax, and MonetDB/XQuery show a super-linear increase in the performance time with respect to the number of child steps of the query and a linear increase with respect to the child step selectivity. All three curves grow slowly or are almost constant on the first dozen queries and grow steeply on the rest. Qizx/Open shows a linear behavior on this document, with a drastic increase of the curve slope at the query of path depth 12.

Further analysis is needed to check whether the curves for testing the impact of child step selectivity are indeed linear or sub-linear. For example, we know that MonetDB/XQuery implements staircase join with pruning for evaluating XPath axes, thus it does one scan of both the context sequence and the document (with skipping) for every child step of the path expression. [Grust *et al.*, 2003]

6.3.2 Meeting the design principles

In this section, we discuss how well the example micro-benchmark conforms to the design principles stated in Section 6.2.1.

Conform P1, the micro-benchmark isolates the impact on the performance of the tested language feature and of the two parameters (the number of child steps, and the child-step selectivity). The data set and query set are simple and they do not vary other features than the tested ones. By measuring the query processing time rather than the total query execution time, the benchmark conforms to design principle P2. As a result, the benchmark measures the performance of the dynamic evaluation phase of the XQuery processing model in isolation from the other phases.

The benchmark running scenario varies the number of child steps, while fixing the document on which the queries run. On "layered.xml", only the number of child steps varies. On "exponential2.xml", the child step selectivity varies together with the number of child steps. Conform P4 (and conform our micro-benchmarking methodology), a second running scenario is needed that varies only the child step selectivity while keeping the number of child steps constant. This can be achieved by executing the same query on different documents and comparing the results. Though the impact of the second parameter can be already seen on the results of the first running scenario, one needs to be aware that these results are dependent also on the number of child steps, thus the impact of the second parameter is not measured in isolation.

The benchmark tests for two parameters only. Arguably, there are many other parameters and parameter values that might impact the performance of child navigation. But whether they all need to be included in this micro-benchmark or separate benchmarks need to be created might be a subjective matter. The authors of this micro-benchmark developed six other micro-benchmarks testing the performance of child navigation (*i*) in the presence of position predicates, (*ii*) where the navigation is included in a predicate expression or not, and (*iii*) where the result nodes are retrieved or only located [Manolescu *et al.*, 2008b]. Whether to assemble these micro-benchmarks together or not is subject to argumentation and might be a matter of opinion. Thus design principles P3 and P6 allow for different interpretations and debates.

Since these principles are difficult to fully adhere to when creating a micro-benchmark, we propose a pay-as-you-go strategy: allowing MemBeR users to submit and use micro-benchmarks that do not fully conform to P3 and P6 yet, and allow for updates to the benchmarks with new parameters or merge micro-benchmarks as the need arizes.

Note that due to its simplicity, the above example benchmark can easily be extended with new parameters or, conform P5, with new values for the two parameters.

6.4 Conclusions

In this chapter, we tackled the problem of performance evaluation of XML query processors and the lack of suitable tools for *precise* and *comprehensive* performance evaluations. As a solution to this problem, we proposed MemBeR, an open-ended, community driven, repository of micro-benchmarks. We endowed the repository with micro-benchmarking design principles and methodology, with a fixed micro-benchmark structure, with suggestions for potentially interesting parameters, and tools for generating parametrized data sets.

MemBeR is freely accessible on the web and serves as a proof of concept of the MemBeR vision, its use, and potential.

In the next chapter, Chapter 7, we present a micro-benchmark for evaluating value-based joins processing techniques that follows the MemBeR methodology.

Chapter 7

A Micro-Benchmark for Value-Based Equi-Joins

In Section 3.7, we investigated the part of the Michican benchmark that was designed to test value-based joins expressed in XQuery and we found that its query set is not sufficient for a satisfactory performance analysis of this query operation. In this chapter, we provide an accurate and more comprehensive micro-benchmark inspired by the join queries of MBench.

We present a micro-benchmark for evaluating the performance of query processing techniques for value-based joins expressed in XQuery (Section 7.2). The benchmark allows for a detailed analysis of engine performance with respect to seven query and data parameters. Using this micro-benchmark, we conduct an extensive analysis of performance of four open-source XQuery engines (Section 7.3). Our analysis indicates that the join-processing techniques deployed by the engines have room for improvement and that the micro-benchmark we propose provides an accurate and comprehensive tool for testing value-based joins (Section 7.4).

The micro-benchmark we present here is an extended version of the micro-benchmark previously published in [Afanasiev and Marx, 2008]. We extended the benchmark with more parameters and with a more systematic way of varying their values and analyzing their impact on the performance results.

7.1 Introduction

In relational databases, the join operation is one of the fundamental query operations. It combines information from two different relations based on their Cartesian product. It is inherently one of the most difficult operations to implement efficiently, as no predefined links between relations are required to exist. Since it is executed frequently and it is expensive, for more than 30 years now, a lot of research effort has been invested in the optimization of join processing [Mishra and Eich, 1992].

In the settings of XML databases and XQuery, we distinguish two types of joins: *value-based joins* and *structural joins*. Just as in the relational case, the value-based join combines information from two sequences of items based on their Cartesian product and the join condition is expressed on the atomic values of the items (e.g., attribute values, the string value of the item, etc.). The structural join, on the other hand, expresses conditions on the structural relationships between the pair of nodes in the XML tree. The `where` clause of the FLWOR expression in XQuery is especially designed to express joins. Nevertheless, XQuery allows for other equivalent ways of expressing joins (we count four different syntactic patterns), which adds to the complexity of the join-processing task.

Early on, much of the research effort on XML databases, focused on optimizing structural joins as it is a new and essential challenge to querying them [Gou and Chirkova, 2007]. The consolidation of the language standard and the need for improving performance of XQuery engines, draws more attention to improving the handling of data values, including optimizing the value-based join. Recall from Chapter 3, where we evaluated the performance of four XQuery engines on five XQuery benchmarks, that the most challenging benchmark queries involved value-based joins. In this chapter, we focus on value-based joins.

As performance evaluation is essential for the development of query processing techniques, a benchmark is needed for testing the performance of value-based joins. The MBench micro-benchmark [Runapongsa *et al.*, 2002] makes a first attempt to address the performance of value-based joins by dedicating four queries to testing them. However, as we argue in Section 3.7, the four queries are not sufficient to get a satisfactory view of the performance of an engine on value-based joins. The previously proposed *application* benchmarks for XQuery are of no help either: although they contain value-based joins in their queries, as we discuss extensively in Chapter 3, their queries and measures do not focus on a particular language operation and thus, they are not suitable for a detailed analysis of value-based joins in particular. A benchmark that targets an accurate and comprehensive evaluation of this operation is needed.

Our goal in this chapter is to design a micro-benchmark targeting the performance of processing techniques for value-based joins. The micro-benchmark should provide a comprehensive view on the performance of an engine on these joins, taking into account performance-critical query and data parameters. The micro-benchmark should allow developers to accurately evaluate their join-processing techniques and it should allow users to analyze the performance of an engine with respect to external parameters they can control.

The research question we address in this chapter is:

7.1. Question. *How to measure the performance of value-based joins expressed in XQuery? What is a suitable measure and which parameters are important to consider?*

Our approach to developing the micro-benchmark is to follow the general MemBeR micro-benchmarking methodology as a design guideline. We draw inspiration from the literature on optimizing joins and construct a list of parameters that are important for the performance of joins in relational databases. Further, we consider a new, XQuery-specific parameter, namely the syntactic pattern used for expressing the join. We observed in Section 3.7, that this parameter has a significant impact on the performance of at least one XQuery engine. Finally, we measure the impact of each parameter on the query processing time(s).

We evaluate our micro-benchmark by analyzing the performance of four open-source XQuery engines: SaxonB, Qizx/Open, Galax, and MonetDB/XQuery. As a result, we obtain the most comprehensive analysis of these engines with respect to value-based joins to date (September 2009). For this analysis, we assume the role of a user that treats the engine as a black box and we explain the performance of the engines entirely in terms of the impact of the micro-benchmark parameters.

7.2 A micro-benchmark for value-based equi-joins

In this section, we describe our micro-benchmark in accordance with the MemBeR methodology.

7.2.1 Target

We present a micro-benchmark that targets the performance of the join-processing mechanism (the component under study) of an XQuery engine (the system under test). The targeted language feature is *value-based equi-joins*, i.e., joins that express equality on data values. We consider equi-joins on numeric data values stored in XML attributes.

7.2.2 Measure

Following the MemBeR methodology (see Chapter 6), the general measure of the micro-benchmark is the performance time as a function of its parameters. We consider six query parameters and one data parameter: *syntactic pattern*, *number of join conditions*, *Boolean connectives*, *join-type*, *join-value data type*, *join selectivity*, and *document size*. We define these parameters in the next section.

For two of the parameters, the measure can be described in a different way. Measuring the impact of the syntactic pattern used to express the join on performance, can be seen as measuring the *robustness* of the engine against syntactic variations. The idea behind this measure is to compare the performance of queries expressed in different syntactic variants. The difference between the performance times of each variant indicates the robustness of the engine.

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

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

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

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

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

Figure 7.1: Four logically equivalent ways of expressing an equi-join.

The impact of the document size on performance measures the *scalability* of join-processing techniques.

The benchmark targets mainly the *total query processing time*, which is the time the engine spends to process the query, from when it was fired to the moment of returning the results. However, it is also interesting to measure the *query compilation time* and the *query execution time* apart. Query compilation time is the time the engine spends in the static analysis phase of the XQuery processing model [World Wide Web Consortium, 2007]. This time includes static optimizations. Query execution time is the time the engine spends on the dynamic evaluation phase of the XQuery processing model [World Wide Web Consortium, 2007]. This time includes dynamic optimizations. The performance metric is CPU time.

7.2.3 Parameters

Following the MemBeR methodology, we design the micro-benchmark to analyze the performance of value-based joins by measuring the impact of different parameters on performance. We vary parameters that are known to have an impact on join processing techniques in relational databases [Mishra and Eich, 1992, Lei and Ross, 1998]:

p1. *syntactic pattern* – the syntactic construction used to express the join;

p2. *number of join conditions* – the number of join conditions used in one join;

p3. *Boolean connectives* – the Boolean connectives used to combine multiple join conditions within a join;

p4. *join type* – whether the path-expressions `A` and `B` (Figure 7.1) are the same or not;

p5. *join-value data type* – the data type of the attributes on which the join is expressed;

p6. *join selectivity* – the number of pairs of items returned by the join; and

p7. *join input size* – the sizes of the two sequences participating in the join, i.e., the sequences selected by the path-expressions `A` and `B` in Figure 7.1. We control this parameter by varying the *document size*—the size of the document on which the join is expressed.

Below we explain each parameter in detail.

p1. Syntactic patterns We consider four equivalent syntactic variants for expressing value-based joins: *where*, *if*, *pred*, and *filter* shown in Figure 7.1, where `A` and `B` are path expressions and `att1` and `att2` are attribute names.

A common way of expressing joins in XQuery is by using the *where* clause of a FLWOR expression. For example, all five XQuery benchmarks discussed in Chapter 3 contain joins expressed in this way. In accordance with the XQuery semantics [World Wide Web Consortium, 2007b], the *where* clause is normalized to an *if* expression in XQuery Core, the complete fragment of XQuery that is used to specify the formal semantics. Thus, the *where* and the *if* join patterns have the same normal form in XQuery Core. Engines that use this normalization are guaranteed to treat these two syntactic patterns equivalently.

In the *predicate* pattern, the join is expressed in a predicate. Two out of the five benchmarks we studied in Chapter 3 (XMach-1, XBench) contain joins expressed with this pattern. In the *filter* pattern, the same predicate condition appears in the return clause as a filter to the sequence construction.

p2. Number of join conditions and **p3. Boolean connectives** We consider joins with *one*, *two*, or *three* different join conditions combined with *conjunction* or *disjunction* between them. For example, the following join pattern contains two join conditions combined with conjunction:

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

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

p4. Join type We consider two different types of joins, self-joins and general joins. If the path-expressions `A` and `B` of a join (Figure 7.1) are the same, then the join is called a *self join*, otherwise it is a *general join*. Thus the self-join is a special case of the general join, where the input sequence is joined with itself.

p5. Join-value data type We consider joins on attributes of data value types *integer* and *id/idref*. In the presence of a DTD, the integer attributes are declared as `CDATA` and the id/idref attributes are of type `ID` and `IDREF`. In the presence of XML Schema, the integer attributes are declared as `xs:integer` and the id/idref attributes are of type `xs:ID` and `xs:IDREF`. The micro-benchmark data set contains both a DTD and an XML Schema to describe the documents.

p6. Join selectivity The number of pairs of items returned by the join, or in other words the *join result size*, we measure as a percentage of the number of pairs of the underlying Cartesian product of the join. We vary the selectivity in four discrete steps: *tiny* (XS, 0.002%), *small* (S, 0.2%), *medium* (M, 14%), and *large* (L, 62%).

p7. Join input size and **Document size** By fixing the selectivity of path-expressions `A` and `B` to a percentage of the number of nodes in the queried document, we tie the join input size directly to the document size. Then we consider documents ranging from 1MB to 46MB (approximately $1,500$ to $67,000$ nodes). The result size of `A` and `B` is $1/64 \times N$ (1.6%), where N is the number of nodes in the document.

7.2.4 Documents

We use the documents and schema of MBench [Runapongsa *et al.*, 2002] for our micro-benchmark. In Section 3.7 of Chapter 3, we have seen that these documents have data value distributions that allow us to easily control the selectivity of the benchmark queries and are the key to the micro-benchmark's ability to vary parameters in isolation. Below, we briefly recall their structure and properties.

MBench documents have two types of elements, `eNest` and `eOccasional`. Most (99%) of the elements of the MBench data are of type `eNest`. The `eNest` element has six numeric attributes with precise value distributions:

- `aUnique1`: A unique integer indicating the element's position in the data tree in breadth-first order; it serves as the element identifier (type `ID`);
- `aUnique2`: A unique integer generated randomly;
- `aLevel`: An integer whose value equals the length of the path from the node to the root;

- `aFour`: An integer set to `aUnique2` *mod* 4;
- `aSixteen`: An integer set to `aUnique1` + `aUnique2` *mod* 16; and
- `aSixtyFour`: An integer set to `aUnique2` *mod* 64.

The remainder (1%) of the elements in the data set are of type `eOccasional`. An `eOccasional` element is added as a child to an `eNest` element if its `aSixtyFour` attribute is 0. The `eOccasional` element contains only one attribute `aRef` of type `IDREF`. The value of `aRef` is set to the `aUnique1` attribute of the parent minus 11 (`aUnique1` − 11), i.e., `aRef` refers to an `eNest` element that precedes the parent of `eOccasional` with 11 positions in the breadth-first order (if it exists, otherwise it refers to the root).

MBench contains three documents of varying sizes. In Section 3.6, we have seen that the smallest document of 46MB is already large enough to seriously challenge the tested engines on join queries. Therefore, for our micro-benchmark we scale the document size by cutting off the 46MB MBench document at different depths starting with depth 9. The original document is of depth 16. As a result we obtain the following data set:

	Depth	Size	# of `eNest` elements ($\times 10^3$)
d1	9	1.1 MB	1.6
d2	10	1.4 MB	2.1
d3	11	2 MB	3.2
d4	12	3.3 MB	5.2
d5	13	5.9 MB	9.3
d6	14	12 MB	17.5
d7	15	22 MB	33.9
d8	16	46 MB	66.7

The document set can be extended in the same manner and granularity by taking as bases the MBench documents of medium (496MB) and large (4.8GB) sizes.

The MBench documents are accompanied by a DTD and an XML Schema.

7.2.5 Queries

It is difficult to design a set of queries that covers all valid value combinations of the parameters we consider. Even if possible, the set might be to too big to be manageable. With p1 having 4 values, p2 and p3 together having 6 value combinations, p4 having 2 values, p5 having 2 values, and p6 having 4 values, there are 384 possible queries. We choose a set of value combinations in which the values of p1, the values *one* and *two* of p2, and the values of p3 are combined in all possible ways, while the value combinations for p2 (the value *three*), p4, p5, and p6 respect the following rule: for every two values of each of these parameters

	$\widetilde{\texttt{AaS}}$			$\widetilde{\texttt{AbM}}$					$\widetilde{\texttt{BaS}}$		
	AaS	Aa&S	AavS	AbM	Ab&M	AbvM	Ab&&M	Ab&vM	BaS	Ba&S	BavS
p2	1	2	2	1	2	2	3	3	1	2	2
p3	–	and	or	–	and	or	and,and	and,or	–	and	or
p4	self	self	self	self	self	self	self	self	gen.	gen.	gen.
p5	int	int	int	int	int	int	int	int	int	int	int
p6	S	S	S	M	M	M	M	M	S	S	S

	$\widetilde{\texttt{BaL}}$			$\widetilde{\texttt{BbXS}}$			$\widetilde{\texttt{BbRefXS}}$		
	BaL	Ba&L	BavL	BbXS	Bb&XS	BbvXS	BbRefXS	Bb&RefXS	BbRefXS
p2	1	2	2	1	2	2	1	2	2
p3	–	and	or	–	and	or	–	and	or
p4	gen.	gen.	gen.	gen.	gen.	gen.	gen.	gen.	gen.
p5	int	int	int	int	int	int	id/idref	id/idref	id/idref
p6	L	L	L	XS	XS	XS	XS	XS	XS

Table 7.1: Micro-benchmark queries and their corresponding parameter values.

there are two queries in the set that differ only by these values. In this way, the difference in performance times of these queries can be safely attributed to the influence of the parameter that is varied and to particular values being chosen. This leads to 80 different queries.

The query set is divided into six classes. Within each class p1 is fully varied creating a set of *logically equivalent queries*, and p2 (values *one* and *two*) and p3 are varied together to obtain valid value combinations in such a way that it creates a set of *document equivalent queries*, i.e., the queries return the same result on the MBench documents. This means that the queries in each equivalence class return the same result. Thus, within one class, p1, p2, and p3 are varied, while p4, p5, and p6 are fixed. From one class to another only one of the parameters p4, p5, or p6 is varied. Figure 7.2 shows which parameter varies between classes. The class $\widetilde{\texttt{AbM}}$ contains two more document equivalent queries with the parameter p2 having value *three*.

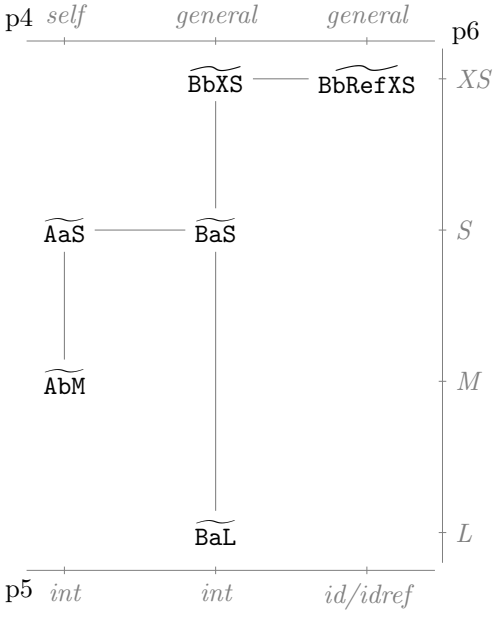


Figure 7.2: The six classes of document-equivalent joins and their parameter dependencies.

All in all, the query set consists of

20 joins (80 syntactically different queries). Table 7.1 lists the joins and their corresponding parameter values. The name of a query is an encoding of its properties. If the query is a self-join, i.e., the path expression `A` is the same as the path expression `B`, then the query name contains the capital letter "A", otherwise it contains the capital letter "B". If any join condition in a query (any query can have one, two or three join conditions) is expressed between two attributes with the same name, then the query contains the small letter "a", otherwise "b". Further, the join name contains "&" and/or "v" for every conjunct and/or disjunct it contains. Queries whose name contains "Ref" are id/idref chasing joins. Finally, the join selectivity of a query is indicated by the suffix "XS", "S", "M", and "L".

For example, query `Ab&vM` is a self-join containing three join conditions on different attributes connected by a conjunction and a disjunction, and it has a medium selectivity. The query has the following general pattern:

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

Each query belongs to one of the six classes of document-equivalent queries. The classes are named after their member with only one join condition. For example, `Ab&vM` is document equivalent to `AbM` and both queries fall into the $\widetilde{\texttt{AbM}}$ class.

The *where* variant of the actual queries can be found in Figure 7.3 and 7.4. The whole set of queries can be found online at `http://ilps.science.uva.nl/Resources/MemBeR/mb-joins/output/outcome.html`.

Note that we fix the selectivity of path `A` and `B` in all the queries by filtering the `eNest` elements with a particular property that always yields approximately 1/64th (1.6%) of all `eNest` elements. For example, the path-expression `//eNest[@aSixtyFour=0]` returns all `eNest` elements whose unique random number stored in `aUnique2` is divisible by 64 (`@aUnique2` *mod* 64 = 0). The path-expression `//eNest[eOccasional]` returns the same elements as the expression `//eNest[@aSixtyFour=0]`, since `eOccasional` occurs whenever an `eNest` has the attribute `aSixtyFour` equal to 0.

We further exploit the correlation between the attribute data of the MBench documents to create document equivalent queries and thus maintain a fixed join selectivity within each class. This equivalence does not hold for all documents conforming to the MBench schema (given by DTD or XML Schema), but for those that respect value dependencies used in the creation of the MBench documents.

AaS:
```
for $e1 in doc()//eNest[@aSixtyFour=0],
    $e2 in doc()//eNest[@aSixtyFour=0]
where $e2/@aUnique2=$e1/@aUnique2
return
(data($e1/@aUnique1),data($e2/@aUnique1))
```

AavS:
```
for $e1 in doc()//eNest[@aSixtyFour=0],
    $e2 in doc()//eNest[@aSixtyFour=0]
where $e2/@aUnique2=$e1/@aUnique2 or
$e2/@aUnique1=$e1/@aUnique1
return
(data($e1/@aUnique1),data($e2/@aUnique1))
```

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

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

BaS:
```
for $e1 in doc()//eNest[eOccasional],
    $e2 in doc()//eNest[@aSixtyFour=0]
where $e1/@aUnique2=$e2/@aUnique2
return
(data($e1/@aUnique1),data($e2/@aUnique1))
```

BavS:
```
for $e1 in doc()//eNest[eOccasional],
    $e2 in doc()//eNest[@aSixtyFour=0]
where $e1/@aUnique2=$e2/@aUnique2 or
      $e2/@aUnique1=$e1/@aUnique1
return
(data($e1/@aUnique1),data($e2/@aUnique1))
```

Aa&S:
```
for $e1 in doc()//eNest[@aSixtyFour=0],
    $e2 in doc()//eNest[@aSixtyFour=0]
where $e2/@aUnique2=$e1/@aUnique2 and
$e2/@aSixtyFour=$e1/@aSixtyFour
return
(data($e1/@aUnique1),data($e2/@aUnique1))
```

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

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

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

Ba&S:
```
for $e1 in doc()//eNest[eOccasional],
    $e2 in doc()//eNest[@aSixtyFour=0]
where $e1/@aUnique2=$e2/@aUnique2 and
      $e1/@aSixtyFour=$e2/@aSixtyFour
return
(data($e1/@aUnique1),data($e2/@aUnique1))
```

Figure 7.3: The *where* variant of the micro-benchmark query set, classes $\widetilde{\texttt{AaS}}$, $\widetilde{\texttt{AbM}}$, and $\widetilde{\texttt{BaS}}$.

7.2.6 Running scenarios

For measuring the *impact of the query parameter*, we fix the document size to the largest value (d8, 46MB) and execute all queries on this document.

For measuring *data scalability*, all queries can be run on a subset (including a small, medium, and a large size document) of the proposed documents or on all document sizes. The latter case generates 640 measurements. Another approach is to first analyze the impact of the query parameters and then choose a few query

BaL:
```
for $e1 in doc()//eNest[@aSixtyFour=4],
    $e2 in doc()//eNest[@aSixtyFour=0]
where $e1/@aLevel=$e2/@aLevel
return
(data($e1/@aUnique1),data($e2/@aUnique1))
```

Ba&L:
```
for $e1 in doc()//eNest[@aSixtyFour=4],
    $e2 in doc()//eNest[@aSixtyFour=0]
where $e1/@aLevel=$e2/@aLevel and
      $e1/@aFour=$e2/@aFour
return
(data($e1/@aUnique1),data($e2/@aUnique1))
```

BavL:
```
for $e1 in doc()//eNest[@aSixtyFour=4],
    $e2 in doc()//eNest[@aSixtyFour=0]
where $e1/@aLevel=$e2/@aLevel or
      $e1/@aSixtyFour=$e2/@aSixtyFour
return
(data($e1/@aUnique1),data($e2/@aUnique1))
```

BbXS:
```
for $e1 in doc()//eNest[eOccasional],
    $e2 in doc()//eNest[@aSixtyFour=4]
where
    $e2/@aUnique2 = $e1/eOccasional/@aRef
return
(data($e1/@aUnique1),data($e2/@aUnique1))
```

Bb&XS:
```
for $e1 in doc()//eNest[eOccasional],
    $e2 in doc()//eNest[@aSixtyFour=4]
where
    $e2/@aUnique2 = $e1/eOccasional/@aRef
    and $e2/@aFour = $e1/@aFour
return
(data($e1/@aUnique1),data($e2/@aUnique1))
```

BbvXS:
```
for $e1 in doc()//eNest[eOccasional],
    $e2 in doc()//eNest[@aSixtyFour=4]
where
    $e2/@aUnique2 = $e1/eOccasional/@aRef
    or $e2/@aSixtyFour = $e1/@aSixtyFour
return
(data($e1/@aUnique1),data($e2/@aUnique1))
```

BbRefXS:
```
for $e1 in doc()//eNest[eOccasional],
    $e2 in doc()//eNest[@aSixtyFour=4]
where
    $e2/@aUnique1 = $e1/eOccasional/@aRef
return
(data($e1/@aUnique1),data($e2/@aUnique1))
```

Bb&RefXS:
```
for $e1 in doc()//eNest[eOccasional],
    $e2 in doc()//eNest[@aSixtyFour=4]
where
    $e2/@aUnique1 = $e1/eOccasional/@aRef
     and $e2/@aFour=$e1/@aFour
return
(data($e1/@aUnique1),data($e2/@aUnique1))
```

BbvRefXS:
```
for $e1 in doc()//eNest[eOccasional],
    $e2 in doc()//eNest[@aSixtyFour=4]
where
    $e2/@aUnique1 = $e1/eOccasional/@aRef
    or $e2/@aSixtyFour=$e1/@aSixtyFour
return
(data($e1/@aUnique1),data($e2/@aUnique1))
```

Figure 7.4: The *where* variant of the micro-benchmark query set, classes $\widetilde{\texttt{BaL}}$, $\widetilde{\texttt{BbXS}}$, and $\widetilde{\texttt{BbRefXS}}$.

parameter value combinations—worst performing, medium performing, and best performing—for the scalability analysis. We implement this approach for the analysis of four XQuery engines presented in Section 7.3.

7.2.7 Analyzing benchmark results

In this section, we explain how the benchmark results should be analyzed.

Impact of p1 The robustness of the join recognition mechanism is measured by comparing the average performance times computed on queries expressed using

one syntactic pattern. Note that all 20 joins of the micro-benchmark are expressed using four different patterns, thus by taking the average for each syntactic variant we cancel the influence of the other parameters on the performance times.

If the average performance times for each value of p1 are similar (not significantly different), then we can conclude that the engine is robust at recognizing the equivalence of the syntactic join patterns.

Impact of p2 The impact of p2, number of join conditions, is measured by comparing the average performance times computed for each value of p2, while the other query parameters are varied.

Note that the set of queries that have the value of p2 fixed on *one* and the set of queries that have the value of p2 fixed on *two* vary the other parameters in all possible ways (note that in the case where p2 has value *one*, p3 has only one possible value). The set of queries that correspond to p2 equals *three* is composed of eight queries from the class $\widetilde{\texttt{AbM}}$ varying only p1 and p3, while p4–p6 are fixed. Thus, the difference between the average performance times for the first two values of p2 show the impact of p2, indifferent of the values of the other parameters. The average performance time computed for the third value of p2 must be interpreted in the settings of the fixed values of p4–p6.

Impact of p3 The impact of p3, the Boolean connectives, is measured by comparing the average performance times computed for each value of p3, while the other query parameters are varied.

The set of queries that have no Boolean connectives, the set of queries that have one conjunct, and the set of queries with one disjunct as the Boolean connective of two join conditions vary all the other parameters completely. The difference between the average times computed on these query sets indicate the impact of the Boolean connective used to combine join conditions, indifferent of the values of the other tested parameters.

The average times computed on the set of queries that have two Boolean connectives to combine three join conditions (queries `Ab&&M` and `Ab&vM`) must be interpreted relative to the fixed values of p4–p6 corresponding to the class $\widetilde{\texttt{AbM}}$.

Impact of p4 The impact of p4, the self or general join type, is measured by comparing the average performance times computed for the $\widetilde{\texttt{AaS}}$ and $\widetilde{\texttt{BaS}}$ classes. The results must be interpreted relative to the fixed values of p5 and p6, all the other parameters are varied exhaustively.

Impact of p5 The impact of p5, the data-value type, is measured by comparing the average performance times computed for the $\widetilde{\texttt{BbXS}}$ and $\widetilde{\texttt{BbRefXS}}$ classes. The results must be interpreted relative to the fixed values of p4 and p6, all the other parameters are varied completely.

Impact of p6 The impact of p6, join selectivity, can be measured by comparing the average performance times computed on $\widetilde{\texttt{BbXS}}$ and $\widetilde{\texttt{BbRefXS}}$, on $\widetilde{\texttt{AaS}}$ and $\widetilde{\texttt{BaS}}$, on $\widetilde{\texttt{AbM}}$, and on $\widetilde{\texttt{BaL}}$, i.e., for each fixed value of p6. Note that p4 and p5 vary non-systematicaly among these classes—by taking the first average we cancel the impact of p5, the second average cancels the impact of p4, while the third and fourth averages have different fixed value combinations for p4 and p5—thus we cannot draw any definite conclusion with respect to this measure. We use this measure only as an indication of the impact of p6 relative to the variance of other parameters.

An alternative analysis of the impact of p6 can be done relative to fixed values for p4 and p5. For example, we can compare the average performance times computed on $\widetilde{\texttt{BbXS}}$, on $\widetilde{\texttt{BbS}}$, and on $\widetilde{\texttt{BaL}}$ to determine the impact of p6 relative to p4 fixed to *general* and p5 fixed to *integer*. Or we can compare the average performance times computed on $\widetilde{\texttt{AaS}}$, and on $\widetilde{\texttt{AbM}}$ to determine the impact of p6 relative to p4 fixed to *self* and p5 fixed to *integer*.

Data scalability (impact of p7) We compare the data scalability of the (chosen) different queries by comparing the angle of the scalability curves.

7.3 The micro-benchmark in action

In this section, we execute the join micro-benchmark on four XQuery engines and analyze their performance. Our primary goal is to evaluate the design of the micro-benchmark.

Experimental setup Our choice fell on the following open-source engines, mainly because of their availability and ease of use: SaxonB v9.1 [Kay, 2009], Qizx/Open v3.0 [Axyana Software, 2009], Galax v0.5.0 [Fernández *et al.*, 2006], and MonetDB/XQuery v0.30.0 [Boncz *et al.*, 2006b]. SaxonB, Qizx/Open, and Galax, are main-memory XQuery engines, while MonetDB/XQuery is a DBMS handling XML databases and XQuery. SaxonB and Qizx/Open are open-source counterparts of commercial engines SaxonA and Qizx, while Galax and MonetDB/XQuery are research prototypes. All engines are of similar maturity and language coverage [World Wide Web Consortium, 2006b]. The engines are run with their default settings, without special indices or tuning for this particular task.

The experiments are conducted on a Fedora 8 machine, with a 64 bit compilation, with 8 CPUs, Quad-Core AMD Opteron(tm) of 2.3GHz, and 20GB RAM. When running the Java implementations, SaxonB and Qizx/Open, we set the Java Virtual Machine maximum heap size to 10GB.

The experiments are run with XCheck, the testing platform presented in Chapter 5. The time measurements are computed by running each query 4 times and

taking the average performance time(s) of the last 3 runs. We interrupt the query executions that take longer than 500 seconds. This is approximately two orders of magnitude larger than the best performance time of each engine on our benchmark.

For the present analysis we consider only the *total query processing time*. The complete experimental data and results, containing more detailed time measurements, can be found at `http://ilps.science.uva.nl/Resources/MemBeR/mb-joins/output/outcome.html`.

Analysis of the results For determining the impact of every parameter on the engines' performance we follow the instructions presented in Section 7.2.7. When the results are not conclusive due to large variances of the time measurements within a study group, we perform more detailed analysis by fixing parameter p1, *syntactic patterns*, to a subset of its values and analyzing the impact of the remaining parameters.

To determine whether the impact of a query parameter on performance times is significant, we use *analysis of variance (ANOVA)* with the significance level set to 0.05 ($\alpha = 0.05$). ANOVA is designed to test whether the means of several groups are equal. It does so by computing the fraction of the variance between the tested groups and the variance within the groups. This fraction is referred to as F. The probability of obtaining the value of F assuming that the groups have the same mean is referred to as p. In our case, a group corresponds to the set of join queries that have the tested parameter set to a particular value. For example, when testing the impact of p1, we consider four groups of queries and analyze whether the performance times obtained for those groups have the same mean. If F is a large number and p is close to zero, then there are at least two groups whose means are significantly different, thus there are at least two values of the parameter on which the engines perform significantly different. When we find that a parameter has a significant impact, we determine the impact of every parameter value using a pairwise comparison analysis called *Least Significant Difference (LSD)*. LSD is a significance test similar to the *t-test*, commonly used for analyzing whether the means of two groups are the same.

ANOVA and LSD make three assumptions. First, they assume that the value distribution within the groups is normal. We do not have good reasons to believe that the performance times within the tested groups conform to a normal distribution—most likely they do not. Nevertheless, analysis of variance has been argued to be robust against violations of this assumption [Keppel, 1973]. Moreover, we use the statistical tests in a conservative manner: whenever we find a significant impact, we are fairly confident that the impact indeed exists, while when we do not state a significant impact, there might be another, more suitable significance test to detect it. The second assumption is that the groups have similar variances. Third, the measurements within each group are assumed to

be independent. The design of our queries guarantees that the last two assumptions hold. For more information on these statistical methods and their use in computer systems performance analysis, we refer to [Jain, 1991, Cohen, 1995].

After analyzing the impact of the query parameters, we divide the query set into subsets that show significant performance differences among each other. The queries within a subset perform similarly. Then we pick a query from each subset and analyze its scalability. We ignore the subsets that owe their impact on performance to queries that exceeded the timeout.

In the following sub-sections, we present the micro-benchmark results in detail for each engine separately. We use three types of plots to display the performance times. First, we plot the performance of different syntactic patterns of all benchmark queries. Next, we use boxplots to display the performance times grouped per parameter—for each query parameter, the corresponding boxplot groups the performance time per parameter value. The boxplots show the *median*, the lower and upper *quartiles*, the *minimum*, and *maximum* of the performance time within each group of queries. The boxplots also show the group outliers (indicated by empty circles). When we state a *visible* difference in performance time we refer to the plots and when we state a *significant* difference we refer to the statistical analysis mentioned above. Finally, we plot the performance time of a selection of queries against the benchmark documents, d1–d8.

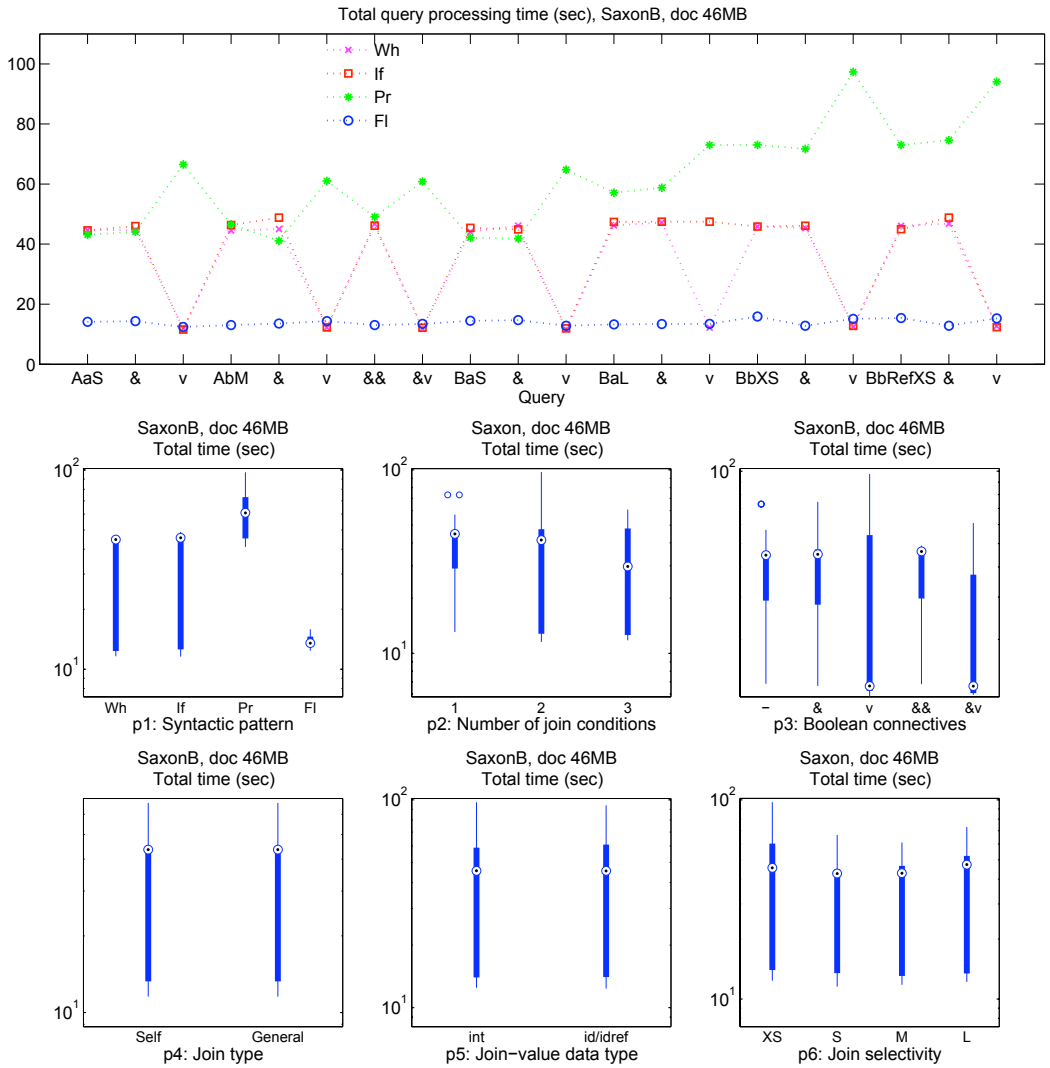
In Section 7.3.5, we further discuss engines' performance and the micro-benchmark design.

7.3.1 SaxonB

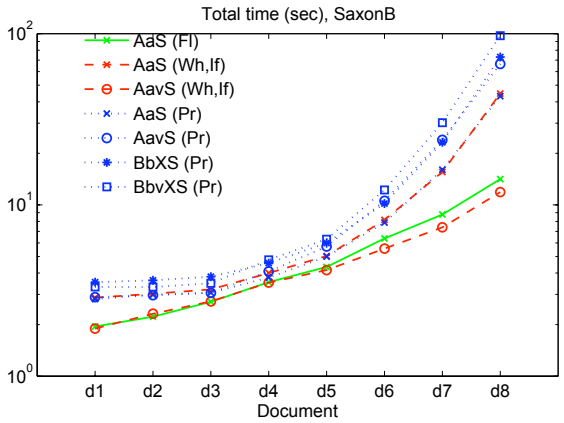
The performance times obtained for SaxonB on our micro-benchmark are shown in Figure 7.5.

For the impact of parameters p1–p6 see Figure 7.5(a). First, we observe that SaxonB performs significantly different on queries expressed via different syntactic patterns ($F = 38$, $p < 0.001$). The *filter* pattern performs best. The times for this pattern are similar on all queries revealing a robust and efficient join processing technique. The performance times on queries expressed via *where* and *if* patterns are very similar and are ranking second best. The queries in the *predicate* pattern perform worse. Pairwise comparison of the four groups of queries shows significant performance differences between *where*, *predicate*, and *filter*, while *where* and *if* are similar.

The performance of different syntactic patterns is very different. For example, *where* and *if* queries that contain a disjunction perform better than average, while for *predicate* the same queries perform worse than average. When analyzing the measurements for the whole query set, due to the large variances in the measurements (see Figure 7.5(a)), none of the remaining parameters shows a significant impact. Thus, we consider that the engine implements three different approaches and analyze them separately with respect to the remaining parameters.



(a) The impact of query parameters p1–p6



(b) Document scalability (parameter p7)

Figure 7.5: SaxonB results on the join micro-benchmark.

On the performance of the *where* and *if* patterns only p3 has a significant impact ($F = 66.7$, $p < 0.001$). The queries containing two join conditions connected by a *disjunction* perform significantly better than the rest.

On the performance of the *predicate* pattern parameters p3 and p6 have a significant impact ($F = 3.7$, $p = 0.049$ and $F = 8.9$, $p = 0.0015$, respectively). The queries containing two join conditions connected by a *disjunction* perform significantly worse than the rest. The queries that have the smallest join selectivity, *XS*, perform significantly worse than the rest.

None of the parameters p2–p6 have a significant impact on the performance of the *filter* pattern.

For the scalability analysis we choose the following queries: `AaS` (Fl), `AaS` (Wh), `AavS` (Wh), `AaS` (Pr), `AavS` (Pr), `BbXS` (Pr), and `BbvXS` (Pr), one representative from each subset of queries obtained by slicing the whole query set conform the impact of p1, p3, and p6. Figure 7.5(b) shows the scalability curves.

Note that the curves fall into two distinctive groups: the curves for `AaS` (Fl) and `AavS` (Wh) have a similar slope growing slower than the curves of the rest of the queries that also share the slope angle. This indicates that the processing approaches used for `AaS` (Fl) and `AavS` (Wh) are essentially different (better performing) than the approaches used for the other queries.

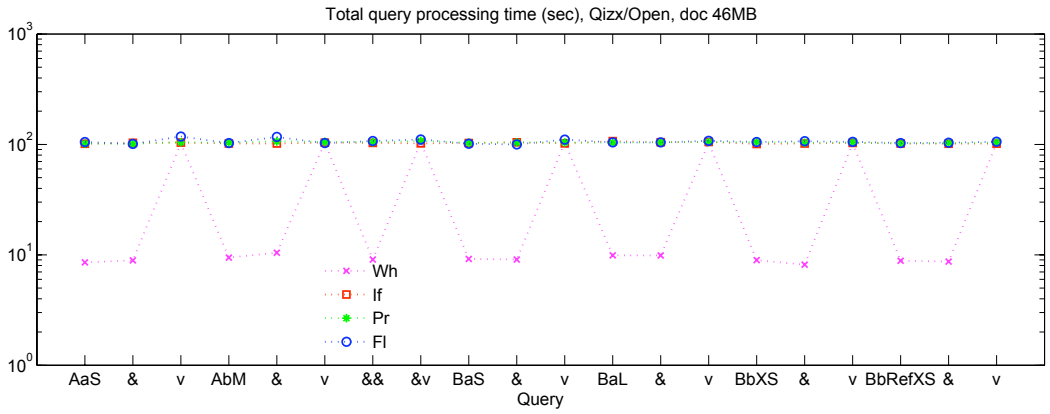
Interpreting the results We were surprised to observe the *filter* variant to be the winner in the case of SaxonB, since this seems a less common way of expressing joins in XQuery. We shared the benchmark results with SaxonB's developer, Michael Kay. The author acknowledged that *filter* is the only variant of the join queries in which the sub-expression `B` (see the query patterns in Figure 7.1) is being pulled out of the nested for-loop, thus it is evaluated once and the results held in memory. In the other cases this opportunity is not being recognized or exploited.

The fact that the *where* and *if* patterns perform the same is, as expected, due to the fact that they share the same normal form in XQuery Core. SaxonB first rewrites a query into its normal form and then executes it. As the developer explains, the disjunctive joins expressed in the *where* and *if* forms are evaluated much faster than the other queries from the same class due to the fact that for these queries the same processing strategy as for the *filter* queries is applied. This explains our results.

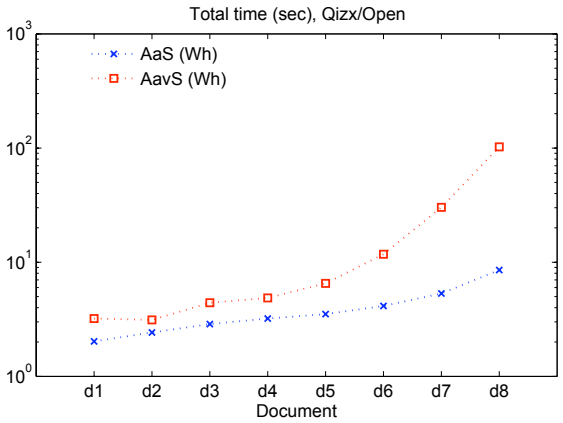
7.3.2 Qizx/Open

The performance results obtained for Qizx/Open on our micro-benchmark are shown in Figure 7.6.

Figure 7.6(a) shows the impact of the query parameters. The queries expressed via the *where* syntactic pattern perform significantly better than the rest ($F = 36.4$, $p < 0.001$). The performance of the other three patterns is very similar.



(a) The impact of query parameters p1–p6



(b) Document scalability (parameter p7)

Figure 7.6: Qizx/Open results on the join micro-benchmark.

Parameter p3 has a significant impact of the performance of the *where* pattern ($F = 2e^{+4}$, $p \approx 0$), with all the queries containing a *disjunction* performing worse than the rest. None of the other parameters p2 and p4–p6 have a significant impact on the performance of Qizx/Open.

For the scalability analysis we choose the following queries: `AaS` (Wh) and `AavS` (Wh), corresponding to the two subsets performing significantly different, obtained by slicing the query set along p1 and p3. Figure 7.6(b) shows the results. `AaS` (Wh) not only performs better than `AavS` (Wh), but it also has a better data scalability, thus the processing technique applied for these queries are essentially different.

Interpreting the results The results for Qizx/Open indicate that the engine deploys a join recognition technique based on a syntactic pattern using the *where* clause. The three other forms are evaluated in the same way, we believe, by

materializing the Cartesian product as intermediate result. Moreover, it seems that the syntactic pattern used for recognizing joins using the *where* clauses fails to capture disjunctive join conditions, since these queries perform as bad as the queries written in the other syntactic forms.

7.3.3 Galax

The micro-benchmark results for Galax are shown in Figure 7.7. Note that Galax exceeded the timeout time on all queries expressed via the syntactic pattern *predicate*.

All four syntactic patterns perform significantly different from each other ($F = 6.2e^{+3}$, $p \approx 0$). *Where* is the best performing, followed by *if*, then by *filter*, and the worst performing is *predicate*.

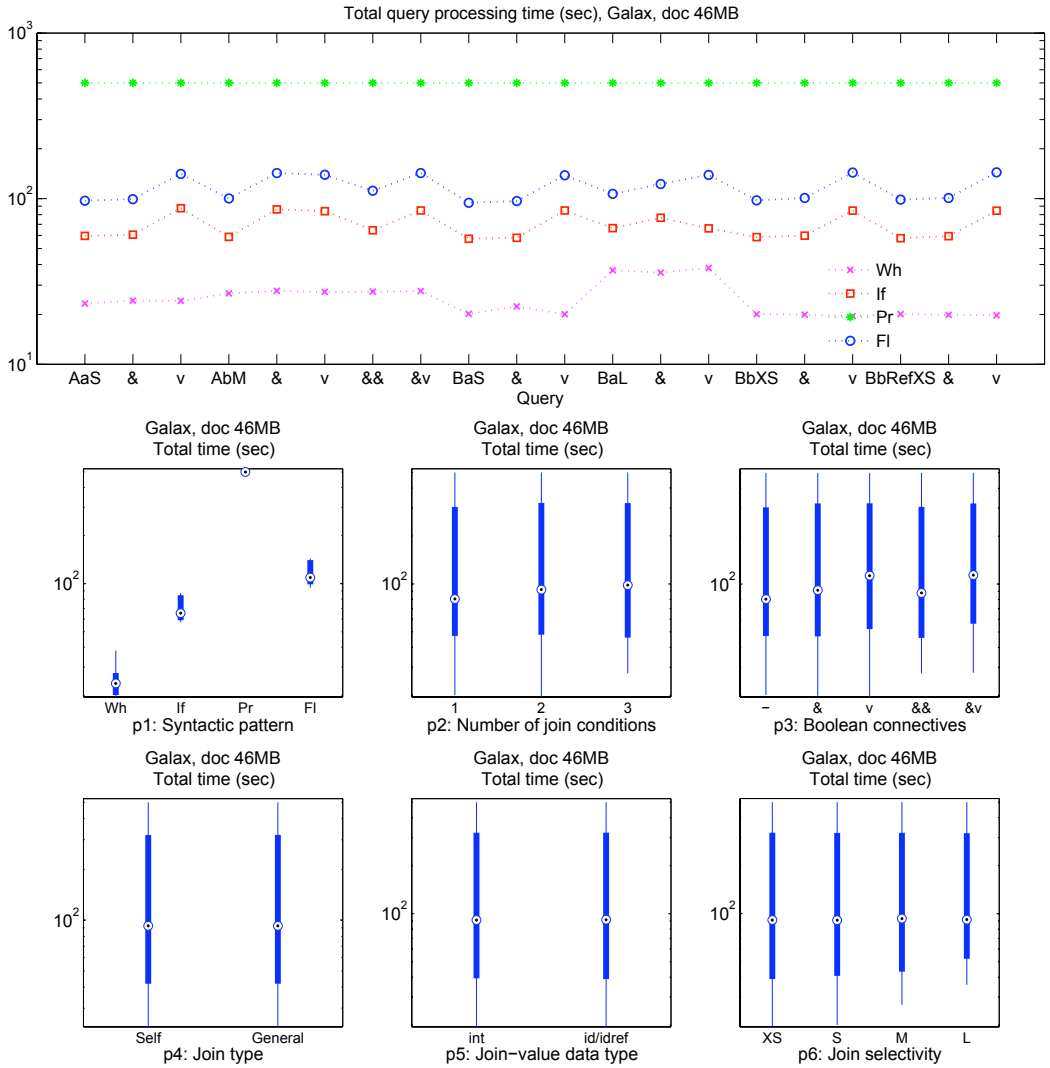
Note that the shape of the curves for the *if* and *filter* patterns in the first plot of Figure 7.7 are similar. Still they are significantly different, thus we analyze each pattern separately. Only parameter p3 has a significant impact on the performance of these patterns ($F = 6.4$, $p = 0.003$ and $F = 8.1$, $p = 0.002$, respectively). The queries that contain two join conditions connected by a *disjunction* perform significantly worse than the ones with only one join condition. The rest of the value pairs perform similarly.

The performance of the *where* pattern is significantly influenced only by the p6 parameter ($F = 164$, $p < 0.001$). All four query groups corresponding to the join selectivity values show significantly different performances—the larger the join selectivity, the longer the processing times.

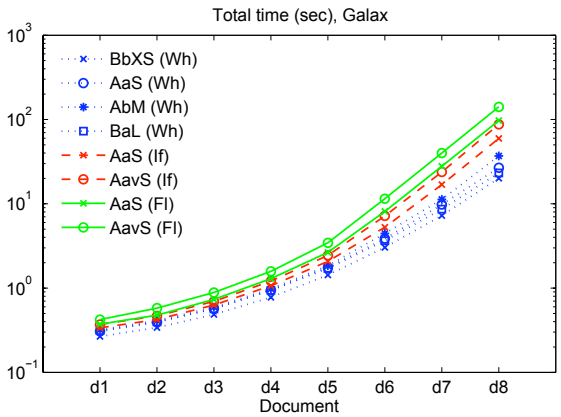
For the scalability analysis we choose the following queries: `BbXS` (Wh), `AaS` (Wh), `AbM` (Wh), `BaL` (Wh), `AaS` (If), `AavS` (If), `AaS` (Fl), and `AavS` (Fl), one representative from each subset of queries obtained by slicing the whole query set conform the impact of p1, p3, and p6. Figure 7.7(b) shows the scalability results.

The queries expressed via the *where* pattern show slightly better scalability than the rest. Further analysis is required to determine how significant the differences are. This can be done by testing the scalability on documents of larger sizes. We indicate in Section 7.2.4 how to obtain larger size documents.

Interpreting the results Even though the Galax implementation pipeline [Fernández *et al.*, 2006] indicates that all the queries are normalized to XQuery Core before processing, the differences between the *where* and *if* patterns indicate that this is not always the case. The performance on the *predicate* pattern suggests that the engine computes the Cartesian product and only then applies the join conditions.

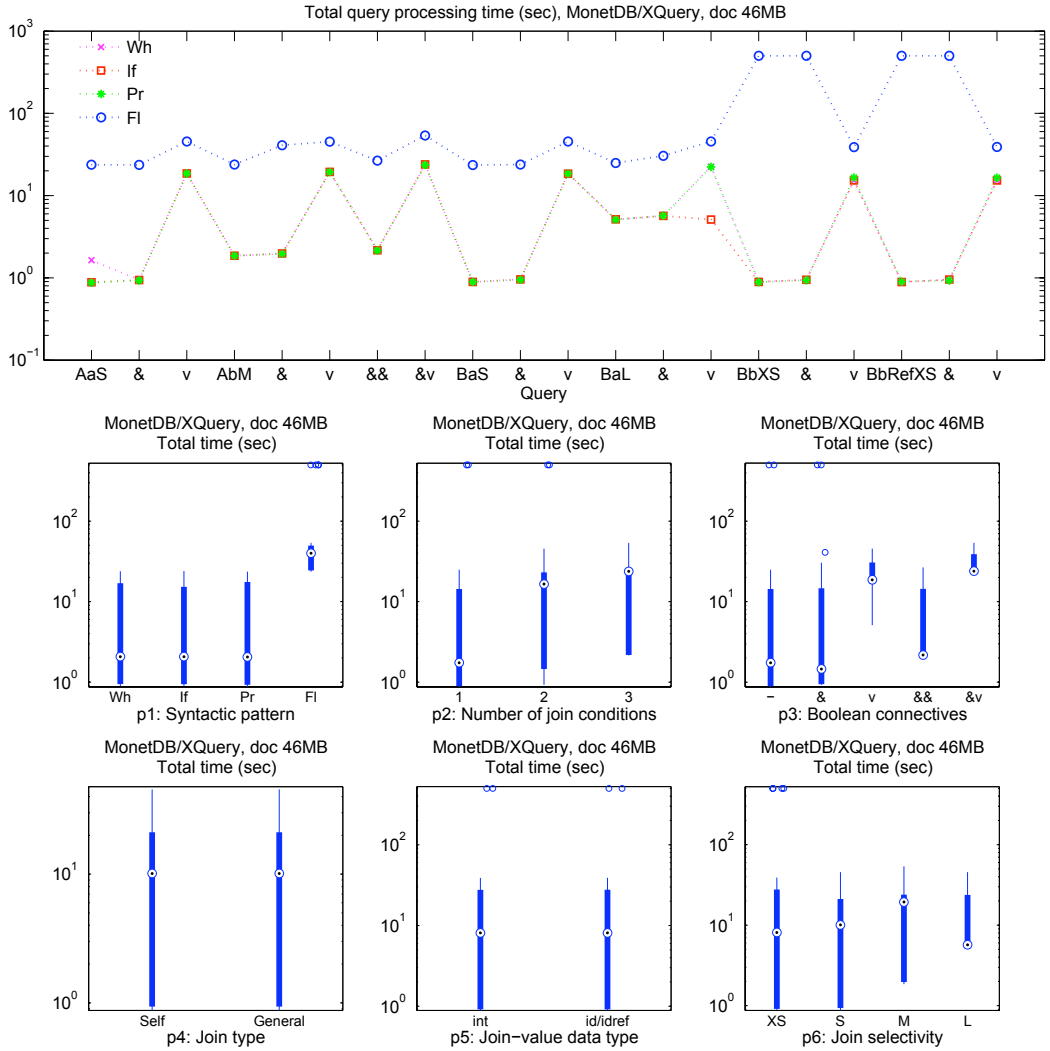


(a) The impact of query parameters p1–p6

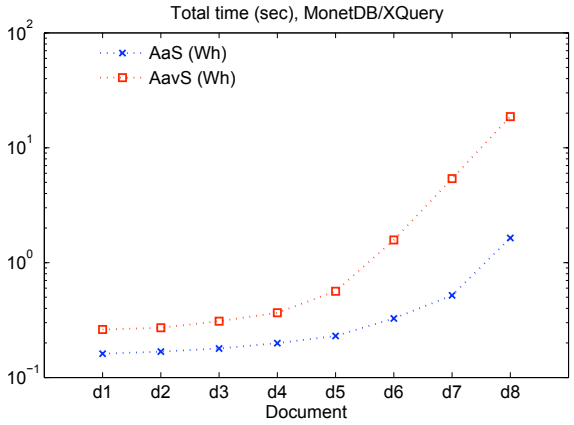


(b) Document scalability (parameter p7)

Figure 7.7: Galax results on the join micro-benchmark.



(a) The impact of query parameters p1–p6



(b) Document scalability (parameter p7)

Figure 7.8: MonetDB/XQuery results on the join micro-benchmark.

7.3.4 MonetDB/XQuery

The performance times obtained for MonetDB/XQuery on our micro-benchmark are shown in Figure 7.8. Note that MonetDB/XQuery exceeded the timeout time on the following queries expressed via the syntactic pattern *filter*: `BbXS`, `Bb&XS`, `BbRefXS`, and `Bb&RefXS`.

Our first observation is that MonetDB/XQuery performs significantly worse on queries expressed via *filter*, than those expressed via *where*, *if*, and *predicate* ($F = 7.9$, $p < 0.001$). The latter three perform almost identically.

Although there is a visible difference in the performance times for every value of p2—the more conditions the join has, the longer the engine takes to evaluate it—the differences are not statistically significant due to large variances in the measurements within each group.

The queries that contain a *disjunction* perform visibly worse than the rest. Again, due to large variances in the performance times within each group, the difference is not statistically significant. If we analyze only the queries expressed via the syntactic patterns *where*, *if*, and *predicate*, thus excluding the impact of the *filter pattern*, the difference becomes significant ($F = 155$, $p \approx 0$). The pairwise value comparison reveals that the queries that contain only a *disjunction* and the queries that contain a *conjunction* and a *disjunction* perform significantly worse than the rest and each other. The other groups do not show a significant difference in performance.

There is no visible difference between the performance times of different values of parameter p4 and p5.

The times obtained for the query group corresponding to query selectivity set to *XS* are significantly worse than the rest ($F = 3$, $p = 0.03$). The difference is due to the four queries that take more than 500 seconds to run. Although there is a visible increase in performance times with the increase of query selectivity, there is no significant difference found among the other query groups. This is due to large variances in the measurements within each group.

In conclusion, varying p1, p2, p3, and p6 shows impact on MonetDB/XQuery performance. For p1, p3, and p6 the impact is statistically significant. A further detailed analysis for subsets of the benchmark queries corresponding to particular parameter configurations might be interesting to consider.

For the scalability analysis we consider the following queries: `AaS` (Wh) and `AavS` (Wh), corresponding to the two subsets performing significantly different, obtained by slicing the query set along p1 and p3. Figure 7.8(b) shows the results. The curve for `AavS` (Wh) seems to grow faster than the curve for `AaS` (Wh), again showing essential differences in the processing approaches used for these queries. Considering larger document sizes might help to determine whether the slope angles are indeed significantly different.

Interpreting the results The results for MonetDB/XQuery indicate that the engine's join recognition mechanism detects the equivalence of the *where*, *if*, and *predicate* patterns. Nevertheless, the performance times for two queries, `AaS` (Wh) and `BavL` (Pr), deviate from the performance times of the queries expressed in the other two patterns. We do not have an explanation for this.

The join processing strategy used for the *filter* pattern performs worse than the strategy used for the joins expressed in the other patterns. On queries `BbXS`, `Bb&XS`, `BbRefXS`, and `Bb&RefXS`, the engine stumbles and does not recognize the join operation—it seems that the engine is computing the Cartesian product as an intermediate result.

7.3.5 Lining up the micro-benchmark results

	SaxonB	**Qizx/Open**	**Galax**	**MonetDB/XQ**
p1	Fl,(Wh,If),Pr	Wh,(If,Pr,Fl)	Wh,If,Fl,Pr	(Wh,If,Pr),Fl
p2	–	–	–	–
p3	(Wh,If): (-,&,&&),(v,&v) Pr: (v,&v),(-,&,&&)	Wh: (-,&,&&),(v,&v)	If, Fl: (-,&,&&),(v,&v)	(Wh,If,Pr): (-,&,&&),v,&v
p4	–	–	–	–
p5	–	–	–	–
p6	Pr: (S,M,L),XS	–	Wh: XS,S,M,L	(S,M,L),XS

Table 7.2: Summary of which query parameters have significant impact on the performance of the four engines. The parameter values are given in the order of significantly decreasing performance. The values that perform similarly are grouped together.

Table 7.2 contains a summary of the results. For every engine we indicate which parameters have a significant impact and on which of its values the engine shows a significantly different performance. The parameter values are given in the order of significantly decreasing performance. The values that perform similarly are grouped together. Some results are indicated for parameter p1 being fixed to a subset of its values.

Note that parameters p1, p3, and p6 have a different impact on each engine.

None of the engines is able to recognize the equivalence of all four syntactic patterns. MonetDB/XQuery is the most robust with respect to this task by performing similarly on three of the four patterns. What is more surprising is that only two of the four engines have the same performance on the *where* and *if* patterns, in spite of the fact that an equivalence preserving translation of the *where* pattern into the *if* pattern is given by the W3C [World Wide Web Consortium, 2007b].

Though the joins with *conjunctive* connectives showed visible negative impact on the engines' performance, all engines performed significantly different on joins with *disjunctive* connectives. One engine, SaxonB, showed better performance on disjunctive joins expressed via the *predicate* pattern. In all other cases, the disjunctive joins performed worse.

In general, we observed a tendency of Galax and MonetDB/XQuery to take more time to answer joins with larger selectivity. However, due to large variances of performance time measurements of queries within the same join selectivity class, the impact of this parameter is not always significant. SaxonB and MonetDB/XQuery showed significantly large performance differences on joins with the smallest selectivity, *XS*. We do not have an explanation for this (abnormal) behavior.

Parameters p2, p4, and p5 did not show a significant impact on any engine.

Parameter p2 has a visible impact on performance. Generally, the more join conditions the longer it takes to evaluate the join. But due to large variances in performance times of queries that contain conjunctions and those that contain disjunctions the influence is not significant. This parameter might show significant impact on those engines on which the impact of p3 is not as large.

Parameters p4 and p5 did not show any visible impact on any of the engines. This raises the question of whether the engines miss optimization opportunities or whether these parameters do not have an impact on value-based join processing as opposed to processing of joins in relational databases. The previous work on optimizing join processing techniques that we cite in Section 7.2.3 and the fact that other XQuery benchmarks also account for these parameters (Chapter 3), are a strong indication that these two parameters are interesting to consider. The results obtained on four engines are not yet convincing arguments against the importance of these parameters. Thus, we believe that the first conclusion is more likely and the engines do not optimize for these parameters.

7.4 Conclusions

Our aim in this chapter was to create a micro-benchmark that targets the performance of query processing techniques for value-based joins. The research questions we addressed are: *How to measure the performance of value-based joins expressed in XQuery? What is a suitable measure and which parameters are important to consider?*

In designing the micro-benchmark we followed the MemBeR methodology. In particular, the benchmark measures the impact of seven query and data parameters on the performance times of an engine. In choosing the benchmark parameters we drew inspiration from the observations that we made previously when analyzing the Michigan benchmark in Chapter 3 and from the work on join optimization techniques in relational databases. The benchmark query set

is carefully designed to allow for testing the impact of every parameter value in isolation. For example, for every parameter and for every two of its values, there are two queries in the set that differ only by these values. This guarantees the accuracy of the benchmark measure with respect to the tested parameters.

We tested our benchmark by analyzing the performance of four XQuery engines. As a result, we obtained a comprehensive overview of the performance of each engine when it comes to evaluating joins and we identified some shortcomings of the engines. Out of seven benchmark parameters, five parameters, *syntactic pattern*, *number of join conditions*, *Boolean connectives*, *join selectivity*, and *document size*, showed visible or significant impact on the performance of at least one engine. None of the engines showed impact of the remaining two parameters, *join-type* and *join-value data type*. We believe that this indicates a missed chance for the four engines to optimize for these parameters and that these parameters are still interesting to consider. We therefore conclude that the benchmark achieves its target and it is a useful tool for profiling the performance of XQuery engines on value-based joins.

In Part I of the thesis, we were concerned with developing methodology and tools for performance evaluation of XQuery processing techniques and engines. We discussed existing XQuery benchmarks, we investigated how to ensure repeatability of experimental studies, we developed a tool for automatic execution of benchmarks, and, finally, we proposed a micro-benchmarking methodology and a micro-benchmark for testing value-based joins expressed in XQuery.

In the next part of the thesis, Part II, we are concerned with optimizing recursion in XQuery. In Section 8.7, we use the tools and methodology developed in Part I to evaluate the optimization we propose.

Part II

Recursion

In this part of the thesis, we address the research questions referring to recursion in XQuery. In Chapter 8, we consider adding an inflationary fixed point operator to XQuery. We develop an optimization technique for processing this operator. Further, we implement this technique on top of MonetDB/XQuery, and evaluate its performance using the tools developed in Part I. In Chapter 9, we study the theoretical properties, decidability and expressivity, of this inflationary fixed point operator in the setting of Core XPath, the XML tree navigational core of XPath and XQuery.

Chapter 8

An Inflationary Fixed Point Operator for XQuery

In this chapter, we investigate a query processing technique for recursion in XQuery and use an experimental analysis to evaluate our approach. Our work is motivated by the current lack of declarative recursive operators in the language. We propose to introduce an *inflationary fixed point (IFP)* operator to XQuery and we present an efficient processing technique for it. This approach lifts the burden of optimizing recursive queries from the user's shoulders and shifts it to the automatic (algebraic) query optimizer.

This chapter is organized as follows: we begin by formally introducing an IFP operator in the setting of XQuery in Section 8.2; in Section 8.3, we present two evaluation algorithms for IFP, one is the direct implementation of the IFP semantics and the other is an optimized variant; we define a distributivity property that allows us to optimize the IFP evaluation in Section 8.4; in Sections 8.5 and 8.6, we provide sufficient syntactic and algebraic conditions for distributivity; we present our implementation of the IFP operator in MonetDB/XQuery and show the gains of the optimization in Section 8.7; in Section 8.8, we address related work on recursion in XQuery as well as on the relational side of the fence; and finally we conclude and discuss future work in Section 8.9.

This chapter is based on work previously published in [Afanasiev *et al.*, 2008, 2009].

8.1 Introduction

The backbone of the XML data model, namely *ordered, unranked trees*, is inherently recursive and it is natural to equip the associated query languages with constructs that can query such recursive structures. While XPath has a very restricted form of recursion via the recursive axes, *e.g.*, `ancestor` and `descendant`, XQuery's [World Wide Web Consortium, 2007] *recursive user-defined functions*

```
<!ELEMENT curriculum (course)*>
<!ELEMENT course prerequisites>
<!ATTLIST course code ID #REQUIRED>
<!ELEMENT prerequisites (pre_code)*>
<!ELEMENT pre_code #PCDATA>
```

Figure 8.1: Curriculum data (simplified DTD).

```
<curriculum>
  <course code="c1">
     <prerequisites>
        <pre_code>c2</pre_code>
     </prerequisites>
  </course>
  <course code="c2">
     <prerequisites>
        <pre_code>c3</pre_code>
        <pre_code>c4</pre_code>
     </prerequisites>
  </course>
  <course code="c3">
     <prerequisites/>
  </course>
  <course code="c4">
     <prerequisites>
        <pre_code>c3</pre_code>
        <pre_code>c1</pre_code>
     </prerequisites>
  </course>
</curriculum>
```

Figure 8.2: Curriculum data snippet "curriculum.xml".

(RUDFs) are the key ingredient of its Turing completeness. To obtain expressive power, the designers of the language took a giant leap, however. User-defined functions in XQuery admit *arbitrary* types of recursion, which makes recursion in the language procedural by nature—a construct that largely evades automatic optimization approaches beyond improvements like tail-recursion elimination or unfolding. This puts the burden of optimization on the user's shoulders.

To make matters concrete, let us consider a typical example of a recursive data and information need.

8.1.1. Example. The DTD of Figure 8.1 (taken from [Nentwich *et al.*, 2002]) describes recursive curriculum data, including courses, their lists of prerequisite courses, the prerequisites of the latter, and so on. Figure 8.2 shows an XML snippet of data that conforms to the given schema. A student wants to know what the courses are that (s)he needs to pass before being able to follow the course coded with `"c1"`. This query cannot be expressed in XPath 2.0, while in XQuery it can be done only by means of recursive user-defined functions. The XQuery expression of Figure 8.3, for instance, uses the `course` element node with code `"c1"` to seed a computation that recursively finds all prerequisite courses, direct or indirect, of course `"c1"`. For a given sequence `$x` of `course` nodes, function `fix(·)` calls out to `rec_body(·)` to find their prerequisites. As long as new nodes are encountered, `fix(·)` calls itself with the accumulated `course` node sequence.

```
1  declare function rec_body($cs) as node()*
2  { $cs/id(./prerequisites/pre_code)
3  };
4  ------------------------------------------------
5  declare function fix($x) as node()*           |
6  { let $res := rec_body($x)                    |
7    return if (empty($x except $res))           |
8           then $res                            |
9           else fix($res union $x)              |
10 };                                            |
11 ------------------------------------------------
12 let $seed := doc("curriculum.xml")
13                        //course[@code="c1"]
14 return fix(rec_body($seed))
```

Figure 8.3: An XQuery query for computing the prerequisites for the course `"c1"` (⌜⌝ marks the fixed point computation).

Note that `fix(·)` implements a generic *fixed point* computation: only the initialization (`let $seed := ···`) and the *recursion body* `rec_body(·)` are specific to the curriculum problem. The XQuery expression describes *how* to get the answer rather than *what* the answer is, i.e., the expression is *procedural* rather than *declarative*. The evaluation strategy encoded by `fix(·)` is not optimal, since it feeds already discovered `course` element nodes back into `rec_body(·)`. There are many ways to optimize a fixed point computation, but this task is left to the user—the query engine has little chance to recognize and optimize the particular recursion operator this expression describes.

Another difficulty with the RUDFs is that they do not seem to fit into the algebraic framework commonly adopted by the database community for query optimization (e.g., Natix Physical Algebra (NPA) [Fiebig *et al.*, 2002], or TAX, a tree algebra for XML used by the Timber engine [Jagadish *et al.*, 2001]). Most XQuery engines have an underlying algebra that facilitates optimizations, but since there is no proper algebraic correspondent for user-defined recursive functions, these optimizations cannot be used for recursive queries. On the other hand, working with operators allows an engine to uniformly apply the algebraic reasoning framework.

Thus, the question we are facing is:

8.1. Question. *What is a suitable declarative recursive operator in XQuery that is rich enough to cover interesting cases of recursion query needs and that allows for (algebraic) automatic optimizations?*

In this chapter, we consider adding a declarative, and thus controlled, form of recursion to XQuery. Our choice falls on the *inflationary fixed point (IFP)* operator, familiar from the context of relational databases [Abiteboul *et al.*, 1995].

While IFP imposes restrictions on the expressible types of recursion, it encompasses a family of widespread use cases of recursion in XQuery, including *structural recursion* (navigating recursively in one direction in the XML tree) and the more general and pervasive *transitive closure (TC)* on path expressions (capturing relations between XML nodes). In particular, XPath extended with IFP captures *Regular XPath* [ten Cate, 2006b], a version of XPath extended with a transitive closure operator. Most importantly, the IFP operator is susceptible to systematic optimizations.

Our goal is to define and implement an IFP operator in the setting of XQuery. We look at a standard optimization technique for IFP developed in the setting of relational databases and check whether it fits the new setting. The optimization consists of reducing the number of items that are fed into the recursion body, avoiding re-computation of items already obtained in previous steps of the recursion. Provided that the recursion body exhibits a specific *distributivity* property, this technique can be applied.

Unlike general user-defined XQuery functions, this account of recursion puts the query processor in control to decide whether the optimization may be safely applied. Distributivity may be assessed at a syntactic level—a non-invasive approach that can easily be realized on top of existing XQuery processors. Alternatively, if we adopt a version of relational algebra extended with a special tree aware operator and with the IFP operator for reasoning about XQuery queries (as in [Grust *et al.*, 2004]), the seemingly XQuery-specific distributivity notion turns out to be elegantly modeled at the algebraic level.

To assess the viability of our approach in practice, we integrated the IFP operator into *Pathfinder*,[1] an open-source XQuery compiler of the relational back-end MonetDB [Boncz *et al.*, 2006a]. The compiler is part of the *MonetDB/XQuery* system,[2] one of the fastest and most scalable XQuery engines today. Compliance with the restriction that IFP imposes on query formulation is rewarded by significant query runtime savings that the IFP-inherent optimization hook can offer. We document the effect for the XQuery processors *MonetDB/XQuery* [Boncz *et al.*, 2006a] and *Saxon* [Kay, 2009].

8.2 Defining an Inflationary Fixed Point operator for XQuery

In this section, we define an *inflationary fixed point operator* for XQuery expressions similar to the inflationary fixed point operator defined in the relational setting [Abiteboul *et al.*, 1995]. We consider the XQuery fragment without recursive user defined functions.

[1] `http://www.pathfinder-xquery.org/`
[2] `http://monetdb.cwi.nl/XQuery/`

Throughout this chapter, we regard an XQuery expression e_1 containing a free variable `$x` as a function of `$x`, denoted by $e_1(\texttt{\$x})$. We write $e_1(e_2)$ to denote a safe substitution $e_1[e_2/\texttt{\$x}]$, i.e., the uniform replacement of all free occurrences of `$x` in e_1 by e_2 making sure that no other free variables get accidentally bound. Finally, $e_1(X)$ denotes the result of $e_1(\texttt{\$x})$, evaluated on some given document, when `$x` is bound to the sequence of items X. It is always clear from the context which free variable we consider. The function $fv(e)$ returns the set of free variables of expression e.

Further, we introduce *set-equality* ($\overset{s}{=}$), a relaxed notion of equality for XQuery item sequences that disregards duplicate items and order, *e.g.*, `(1,"a")` $\overset{s}{=}$ `("a",1,1)`. For X_1, X_2 sequences of type `node()*`, we have

$$X_1 \overset{s}{=} X_2 \quad \Leftrightarrow \quad \texttt{fs:ddo}(X_1) = \texttt{fs:ddo}(X_2) \ , \tag{SetEq}$$

where `fs:ddo(·)` abbreviates the function `fs:distinct-doc-order(·)` of the XQuery Formal Semantics [World Wide Web Consortium, 2007b].

To streamline the discussion, in the following we *only* consider XQuery expressions and sequences of type `node()*`. An extension of our definitions and results to general sequences of type `item()*` is possible but requires the replacement of XQuery's node set operations that we use (`fs:ddo(·)`, `union` and `except`) with the corresponding operations on sequences of items.

8.2.1. Definition (Inflationary Fixed Point Operator). Let e_{seed} and $e_{body}(\texttt{\$x})$ be XQuery expressions. The *inflationary fixed point (IFP) operator* applied to $e_{body}(\texttt{\$x})$ and e_{seed} is the expression

$$\texttt{with \$x seeded by } e_{seed} \texttt{ recurse } e_{body}(\texttt{\$x}) \ . \tag{8.1}$$

The expressions e_{body}, e_{seed}, and `$x` are called, respectively, the recursion *body*, *seed*, and *variable* of the IFP operator.

The semantics of (8.1), called the *IFP of* $e_{body}(\texttt{\$x})$ *seeded by* e_{seed}, is the sequence Res_k obtained in the following manner:

$$\begin{array}{lcl} Res_0 & \leftarrow & e_{body}(e_{seed}) \\ Res_{i+1} & \leftarrow & e_{body}(Res_i) \texttt{ union } Res_i \ , \quad i \geqslant 0 \ , \end{array} \tag{IFP}$$

where $k \geqslant 1$ is the minimum number for which $Res_k \overset{s}{=} Res_{k-1}$. If no such k exists, the semantics of (8.1) is *undefined*.

Note that if expression e_{body} does *not* use node constructors (*e.g.*, `element` $\{\cdot\}$ $\{\cdot\}$ or `text` $\{\cdot\}$), expression (8.1) operates over a finite domain of nodes and its semantics is always defined. Otherwise, nodes might be created at each iteration and the semantics of (8.1) might be undefined. For example, `with $x seeded by () recurse <a>{$x}</a>` generates infinitely many distinct elements, thus it is undefined. When the result is defined, it is always a duplicate free and document ordered sequence of nodes, due to the semantics of the set operation `union`.

8.2.2. Example. Using the new operator we can express the query from Example 8.1.1 in a concise and elegant fashion:

```
with $x seeded by doc("curriculum.xml")//
                                        course[@code="c1"]      (Q1)
recurse $x/id(./prerequisites/pre_code)
```

In XQuery, each specific instance of the IFP operator can be expressed via the recursive user-defined function template `fix(·)` (shown in ⋯ in Figure 8.3). Since the IFP operator is a second-order construct taking an XQuery variable name and two XQuery expressions as arguments, the function `fix(·)` has to be interpreted as a template in which the recursion body `rec_body(·)` needs to be instantiated. Note that XQuery 1.0 does not support higher-order functions. Given this, Expression (8.1) is equivalent to the expression

$$\texttt{let \$x :=}\ e_{seed}\ \texttt{return fix(rec\_body(\$x))}.$$

8.2.3. Definition. XQuery$^{\text{-rudf}}$ is the XQuery fragment *without recursive user defined functions*, i.e., where the function dependency graph is acyclic. XQuery$^{\text{-rudf,+ifp}}$ is XQuery$^{\text{-rudf}}$ closed under the IFP operator.

8.2.1 Using IFP to compute Transitive Closure

Transitive closure is an archetype of recursive computation over relational data, as well as over XML instances. For example, Regular XPath [ten Cate, 2006b, Marx, 2004] extends the navigational fragment of XPath, Core XPath [Gottlob and Koch, 2002], with a transitive closure operator defined on location paths. We extend this definition to any XQuery expression of type `node()*`.

8.2.4. Definition (Transitive Closure). Let e be an XQuery expression. The *transitive closure (TC) operator* $(.)^+$ applied to e is the expression e^+. The semantics of e^+ is the result of

$$e\ \texttt{union}\ e/e\ \texttt{union}\ e/e/e\ \texttt{union}\ \ldots\ , \qquad \text{(TC)}$$

if it is a finite sequence. Otherwise, the semantics of e^+ is *undefined*.

Analogously to the IFP operator, e^+ might be *undefined* only if e contains node constructors. For example, `<a/>`$^+$ generates infinitely many distinct empty elements tagged with `a`, thus it is undefined.

8.2.5. Example. The TC operator applied to location paths expresses the transitive closure of paths in the tree:

```
(child::*)+          ≡  descendant::* ,
(child::*)+/self::a  ≡  descendant::a .
```

The TC operator applied to any expression of type `node()*` expresses the transitive closure of node relations: the query from Example 8.1.1 can be expressed as

```
doc("curriculum.xml")//course[@code="c1"]/
                        (id(./prerequisites/pre_code))+ .
```

8.2.6. Definition. XQuery$^{\text{-rudf,+tc}}$ is the XQuery fragment without recursive user-defined functions extended with the TC operator, i.e., is the XQuery$^{\text{-rudf}}$ fragment of XQuery closed under the TC operator.

Note that Regular XPath is a strict fragment of XQuery$^{\text{-rudf,+tc}}$—the expression in the scope of the TC operator in the curriculum example is a *data-value join* and cannot be expressed in Regular XPath.

8.2.7. Remark. For some expression e, the transitive closure of e can be expressed using the IFP operator as follows:

$$e^+ \equiv \texttt{with \$x seeded by . recurse \$x/}e \ ,$$

where '`.`' denotes the context node. In Section 8.4, we define a distributivity property for e that guarantees the correctness of this translation. We also show that all Regular XPath queries have this property, thus can be expressed in XQuery$^{\text{-rudf,+ifp}}$ using this translation.

8.2.2 Comparison with IFP in SQL:1999

The IFP operator is present in SQL in terms of the `WITH RECURSIVE` clause introduced in the ANSI/ISO SQL:1999 standard [Gulutzan and Pelzer, 1999]. The `WITH` clause defines a virtual table, while `RECURSIVE` specifies that the table is recursively defined. To exemplify this, consider the table `Curriculum`(`course`, `prerequisite`) as a relational representation of the curriculum data from Figure 8.2. The prerequisites `P`(`course_code`) of the course with code `'c1'` expressed in Datalog are:

$$\begin{aligned} \texttt{P(x)} &\leftarrow \texttt{Curriculum('c1', x)} \\ \texttt{P(x)} &\leftarrow \texttt{P(y), Curriculum(y, x)} \ . \end{aligned}$$

The equivalent SQL query is:

```
WITH RECURSIVE P(course_code) AS
  (SELECT prerequisite                          }
   FROM   Curriculum                            } seed
   WHERE  course = 'c1')                        }
 UNION ALL
  (SELECT Curriculum.prerequisite               }
   FROM   P, Curriculum                         } body
   WHERE  P.course_code = Curriculum.course)    }
SELECT DISTINCT * FROM P;
```

```
res ← e_body(e_seed);
do
  |
  |  res ← e_body(res) union res;
while res grows ;
return res;
```

(a) Algorithm *Naïve*

```
res ← e_body(e_seed);
Δ ← res;
do
  |  Δ ← e_body(Δ) except res;
  |  res ← Δ union res;
while res grows ;
return res;
```

(b) Algorithm *Delta*

Figure 8.4: Algorithms to evaluate the IFP of e_{body} seeded by e_{seed}. The result is *res*.

Analogously to the XQuery variant, the query is composed of a *seed* and a *body*. In the seed, table `P` is instantiated with the direct prerequisites of course `'c1'`. In the body, table `P` is joined with table `Curriculum` to obtain the direct prerequisites of the courses in `P`. The result is added to `P`. The computation of the body is iterated until `P` stops growing.

The SQL:1999 standard only requires engine support for *linear* recursion, i.e., each `RECURSIVE` definition contains at most one reference to a mutually recursively defined table. Note that the recursive table `P` in the example above is defined linearly: it is referenced only once in the `FROM` clause of the body. This syntactic restriction allows for efficient evaluation. The SQL:1999 `WITH RECURSIVE` clause without syntactic restrictions, we call *full* recursion.

Note that the IFP operator introduced in Definition 8.2.1 does not state any syntactic restriction on the recursive body. In this respect, the IFP in XQuery$^{\text{-rudf,+ifp}}$ corresponds to full recursion in SQL. In Section 8.5, we define a syntactic restriction for IFP expressions in XQuery$^{\text{-rudf,+ifp}}$ that is similar to linear recursion in SQL.

8.3 Algorithms for IFP

In this section, we describe two algorithms, *Naïve* [Bancilhon and Ramakrishnan, 1986] and *Delta* [Güntzer *et al.*, 1987], commonly used for evaluating IFP queries in the relational setting. *Delta* is more efficient than *Naïve*, but, unfortunately, *Delta* is not always correct for our IFP operator for XQuery.

8.3.1 *Naïve*

The semantics of the inflationary fixed point operator given in Definition 8.2.1 can be implemented straightforwardly. Figure 8.4(a) shows the resulting procedure, commonly referred to as *Naïve* [Bancilhon and Ramakrishnan, 1986]. At each iteration of the **while** loop, $e_{body}(\cdot)$ is executed on the intermediate result sequence

```
declare function delta($x,$res) as node()*
{
  let $delta := rec_body($x) except $res
  return if (empty($delta))
         then $res
         else delta($delta,$delta union $res)
};
```

Figure 8.5: An XQuery formulation of *Delta*.

res until no new nodes are added to it. Note that the recursive function `fix(·)` shown in Figure 8.3 is the XQuery equivalent of *Naïve*. Another remark is that the old nodes in *res* are fed into $e_{body}(\cdot)$ over and over again. Depending on the nature of $e_{body}(\cdot)$, *Naïve* may involve a substantial amount of redundant computation.

8.3.2 *Delta*

A folklore variation of *Naïve* is the *Delta* algorithm [Güntzer *et al.*, 1987] of Figure 8.4(b). In this variant, $e_{body}(\cdot)$ is invoked only for those nodes that have not been encountered in earlier iterations: the node sequence Δ is the difference between $e_{body}(\cdot)$'s last answer and the current result *res*. In general, $e_{body}(\cdot)$ will process fewer nodes. Thus, *Delta* introduces a significant potential for performance improvement, especially for large intermediate results and computationally expensive recursion bodies (see Section 8.7).

Figure 8.5 shows the corresponding XQuery user-defined function `delta(·,·)` which, for Example 8.1.1 and thus Query Q1, can serve as a drop-in replacement for the function `fix(·)`—line 14 needs to be replaced by

```
return delta(rec_body($seed),()).
```

Unfortunately, *Delta* is *not always* a valid optimization for the IFP operator in XQuery. Consider the following examples.

8.3.1. Example. Consider expression (Q2) below.

```
let $seed := (<a/>,<b><c><d/></c></b>)
return
   with $x seeded by $seed
   recurse
         if( some $i in $x satisfies $i/self::a )
         then $x/* else ()
```
(Q2)

While *Naïve* computes `(a,b,c,d)`, *Delta* computes `(a,b,c)`, where *a*, *b*, *c*, and *d* denote the elements constructed by the respective subexpressions of the seed. The table below illustrates the progress of the iterations performed by both algorithms.

Iteration	*Naïve*	*Delta*	
	res	*res*	Δ
0	(a,b)	(a,b)	(a,b)
1	(a,b,c)	(a,b,c)	(c)
2	(a,b,c,d)	(a,b,c)	$()$
3	(a,b,c,d)		

8.3.2. Example. Consider another expression:

```
with $x seeded by ()
recurse if( count($x) < 10 )
        then <a>{$x}</a>
        else ()
```

Naïve computes a sequence of 10 elements with the tag `a` of different structure: the first element is a tree containing one node, the last one is a tree of depth 9.[3] On the other hand, *Delta* falls into an infinite loop and thus the result is *undefined*.

Even though *Delta* does not always compute the IFP operator correctly, we can investigate for which IFP expressions *Delta* does compute the correct result and to apply it in those cases. In the next section, we provide a natural semantic property that allows us to trade *Naïve* for *Delta*.

8.4 Distributivity for XQuery

In this section, we define a *distributivity property* for XQuery expressions. We show that distributivity implies the correctness of *Delta*. We also show that distributivity allows for the elegant translation of the TC operator into the IFP operator discussed in Section 8.2.1. Unfortunately, determining whether an expression is distributive is undecidable. In the next section though, we present an efficient and expressively complete syntactic approximation of distributivity.

8.4.1 Defining distributivity

A function e defined on sets is *distributive* if for any non-empty sets X and Y, $e(X \cup Y) = e(X) \cup e(Y)$. This property suggests a divide-and-conquer evaluation strategy which consists of applying e to subsets of its input and taking union of the results. We define a similar property for XQuery expressions using the sequence set-equality defined by (SetEq) in Section 8.2. Recall that in this chapter we *only* consider XQuery expressions and sequences of type `node()*`.

[3] For the interested reader, this expression computes the tree encoding of the first 10 von Neumann numerals: the nodes represent sets of sets and the child relation represents the set membership.

8.4.1. Definition (Distributivity Property). Let $e($`$x`$)$ be an XQuery expression. Expression $e($`$x`$)$ is *distributive for* `$x` iff for any non-empty sequences X_1, X_2,

$$e(X_1 \texttt{ union } X_2) \quad \stackrel{s}{=} \quad e(X_1) \texttt{ union } e(X_2) \ . \tag{8.2}$$

Note that if e does not contain node constructors and if e is constant for `$x`, i.e., `$x` is not a free variable of e, then Eq. (8.2) always holds, thus e is distributive for `$x`.

8.4.2. Proposition. *Let e be an XQuery expression. Expression $e($`$x`$)$ is* distributive for `$x` *iff for any sequence $X \neq$* `()` *and any fresh variable* `$y`*,*

$$(\texttt{for \$y in \$x return } e(\texttt{\$y}))(\mathrm{X}) \quad \stackrel{s}{=} \quad e(X) \ . \tag{8.3}$$

Proof. Consider the following equality: for any sequence $X = (x_1, \ldots, x_n)$, $n \geq 1$,

$$(e(x_1) \texttt{ union } \ldots \texttt{ union } e(x_n)) \quad \stackrel{s}{=} \quad e(X) \ . \tag{8.4}$$

It is easy to see that for any partition X_1 and X_2 of X, i.e., $X_1 \cap X_2 = \emptyset$ and $X_1 \cup X_2 = X$, if Eq. (8.2) holds then Eq. (8.4) also holds for X, and vice versa. Thus Eq. (8.2) is equivalent to Eq. (8.4).

Conform XQuery formal semantics [World Wide Web Consortium, 2007b], for $X = (x_1, \ldots, x_n)$, $n \geq 1$, the left-hand side of Eq. (8.3) equals the sequence concatenation $(e(x_1), \ldots, e(x_n))$. The later sequence is set-equal to $(e(x_1)$ `union` $\ldots$ `union` $e(x_n))$, the left-hand side of Eq. (8.4). From the equivalence of Eq. (8.2) and (8.4), the equivalence of Eq. (8.2) and (8.3) follows. QED

We will use Eq. (8.3) as an alternative definition of distributivity.

8.4.3. Proposition (Distributivity of Path Expressions). *An XQuery expression of the form $e($`$x`$) =$ `$x`$/p$ is distributive for* `$x` *if the expression p neither contains (i) free occurrences of* `$x`*, nor (ii) calls to* `fn:position()` *or* `fn:last()` *that refer to the context item sequence bound to* `$x`*, nor (iii) node constructors.*

Proof. Consider the XQuery Core [World Wide Web Consortium, 2007b] equivalent of `$x/`p, `fs:ddo(for $fs:dot in $x return` p`)`, which is set-equal to `for $fs:dot in $x return` p, where `$fs:dot` is a built-in variable that represents the context item. Given conditions (i) to (iii), the left-hand side of Eq. (8.3), (`for $y in $x return (for $fs:dot in $y return` p`)`$)(X)$ is set-equal to the right-hand side, (`for $fs:dot in $x return` p`)`(X), for any non-empty X. QED

8.4.4. Example. Expressions of the form `$x/`p where p is a Core XPath or even Regular XPath expression are examples of distributive expressions in XQuery. Note that, by definition, all Core XPath and Regular XPath expressions satisfy conditions (i) to (iii) of Proposition 8.4.3 above.

8.4.5. Example. It is easy to see that `$x[1]` is not distributive for `$x`. For a counterexample, let `$x` be bound to (a, b), then `$x[1]` evaluates to (a), while `for $i in $x return $i[1]` evaluates to (a, b).

In the next section, we establish the main benefit of distributivity, namely that we can safely trade *Naïve* for *Delta* for computing distributive IFP expressions.

8.4.2 Trading *Naïve* for *Delta*

We say that *Delta* and *Naïve* are *equivalent for a given IFP expression*, if for any XML document (collection) both algorithms produce the same sequence of nodes or both fall into infinite loops.

8.4.6. Theorem (*Delta* computes IFP). *Consider the expression* `with $x seeded by` e_{seed} `recurse` e_{body}`($x)`. *If* e_{body}`($x)` *is distributive for* `$x`*, then the algorithm* Delta *correctly computes the IFP of* e_{body}`($x)` *seeded by* e_{seed}.

Proof. We show by inductive reasoning that *Delta* and *Naïve* have the same intermediate results, denoted by res_i^Δ and res_i, respectively. The equivalence of *Naïve* and *Delta* follows from this. The induction is on i, the iteration number of the **do** $\cdots$ **while** loops.

In its first iteration, *Naïve* yields $e_{rec}(e_{rec}(e_{seed}))$ `union` $e_{rec}(e_{seed})$ which is equivalent to *Delta*'s first intermediate result $(e_{rec}(e_{rec}(e_{seed}))$ `except` $e_{rec}(e_{seed}))$ `union` $e_{rec}(e_{seed})$.

Suppose that $res_k^\Delta = res_k$, for all $k \leq i$. We show that $res_{i+1}^\Delta = res_{i+1}$.

By the definition of *Naïve*, $res_{i+1} = e_{body}(res_i)$ `union` res_i. Since e_{body} is distributive for `$x`, we can apply Set-Eq. (8.2) to $e_{body}(res_i) = e_{body}\big((res_i$ `except` $\Delta_i)$ `union` $\Delta_i\big)$ and obtain

$$res_{i+1} = \big(e_{body}(res_i \texttt{ except } \Delta_i) \texttt{ union } e_{body}(\Delta_i)\big) \texttt{ union } res_i \ . \tag{8.5}$$

Note that we are allowed to replace set-equality with strict equality here, since both sequences are document ordered and duplicate free due to the semantics of `union`.

By induction, $res_i^\Delta = res_i$ and thus the right-hand side of (8.5) can be written as $\big(e_{body}(res_i^\Delta$ `except` $\Delta_i)$ `union` $e_{body}(\Delta_i)\big)$ `union` res_i^Δ. Note that res_i^Δ is the disjoin union of res_{i-1}^Δ and Δ_i. As a result, (8.5) becomes

$$res_{i+1} = e_{body}(res_{i-1}^\Delta) \texttt{ union } e_{body}(\Delta_i) \texttt{ union } res_i^\Delta \ . \tag{8.6}$$

By applying the induction step once more, we obtain

$$res_{i+1} = e_{body}(res_{i-1}) \texttt{ union } e_{body}(\Delta_i) \texttt{ union } res_i \ . \tag{8.7}$$

Since $e_{body}(res_{i-1})$ is contained in res_i, it follows that the left-hand side of (8.7) equals $e_{body}(\Delta_i)$ `union` res_i, which by induction equals $e_{body}(\Delta_i)$ `union` res_i^{Δ}. A final chain of equalities brings us the desired result:

$$\begin{aligned} res_{i+1} &= e_{body}(\Delta_i) \texttt{ union } res_i^{\Delta} \\ &= \left(e_{body}(\Delta_i) \texttt{ except } res_i^{\Delta}\right) \texttt{ union } res_i^{\Delta} \\ &= \Delta_{i+1} \texttt{ union } res_i^{\Delta} \\ &= res_{i+1}^{\Delta} \ . \end{aligned} \tag{8.8}$$

QED

We have proven that *Delta* can be correctly applied for the evaluation of a distributive IFP expression. In the next section, we discuss one more benefit of distributivity, namely the correctness of the straightforward translation of the TC operator into the IFP operator discussed in Section 8.2.1.

8.4.3 Translating TC into IFP

Distributivity is also a key to understanding the relation between the TC operator and the IFP operator in the setting of XQuery. Intuitively, if expression e is distributive for the context sequence, then e^+ is equivalent to `with $x seeded by . recurse $x/`e, where `$x`, a fresh variable, is a place holder for the context sequence.

8.4.7. Theorem. *Consider an XQuery expression* e *and a variable* `$x`*, such that* `$x`$\notin fv(e)$*. If* `$x/`e *is distributive for* `$x`*, then*

$$e^+ \quad = \quad \texttt{with \$x seeded by . recurse \$x/}e. \tag{TC2IFP}$$

Proof. First, we rewrite Definition 8.2.4 of the TC operator. It is not hard to see that the semantics given by (TC′) below it is equivalent to the semantics given by (TC)—it is merely a change in representation. Thus, we consider the semantics of e^+ to be the sequence of nodes Res'_k, if it exists, obtained in the following manner:

$$\begin{array}{lcl} \Theta_0 & \leftarrow & e \\ Res'_0 & \leftarrow & e \\ \Theta_{i+1} & \leftarrow & \Theta_i/e \\ Res'_{i+1} & \leftarrow & \Theta_{i+1} \texttt{ union } Res'_i, \quad i \geqslant 0, \end{array} \tag{TC′}$$

where $k \geqslant 1$ is the minimum number for which $Res'_k \overset{s}{=} Res'_{k-1}$. Otherwise, e^+ is *undefined*. Next, let us compare the Res'_i sequences with the sequences Res_i obtained conform Definition 8.2.1 while computing the IFP of `$x/`e `seeded by` `.` (the context node):

$$\begin{array}{lcl} Res_0 & \leftarrow & ./e \\ Res_{i+1} & \leftarrow & Res_i/e \texttt{ union } Res_i, \quad i \geqslant 0. \end{array}$$

If $\texttt{\$x/}e$ is distributive for $\texttt{\$x}$ then $Res'_{i+1} = \Theta_i/e$ `union` Res'_i is equal to Res_i/e `union` $Res_i = Res_{i+1}$. Below, we prove this equality rigorously. The correctness of the given translation of the TC operator into the IFP operator follows.

Let $e_{body} = \texttt{\$x/}e$ and $e_{seed} = .$ (the context node). Further, let Res_i, $i \geqslant 0$ be sequences obtained conform Definition 8.2.1 while computing the IFP of $e_{body}(\texttt{\$x})$ `seeded by` e_{seed}. Similarly, let Res'_i, $i \geqslant 0$ be sequences obtained conform (TC′) while computing e^+. We show by induction on i that Res_i equals Res'_i. This proof is similar to the proof of Theorem 8.4.6.

The base of the induction holds straightforwardly:

$$\begin{aligned} Res_1 &= e_{body}\big(e_{body}(e_{seed})\big) \texttt{ union } e_{body}(e_{seed}) \\ &= \texttt{./}e\texttt{/}e \texttt{ union ./}e \\ &= e\texttt{/}e \texttt{ union } e \\ &= Res'_1 \ . \end{aligned}$$

Suppose that $Res_j = Res'_j$, for all $j \leqslant i$. The equality $Res_{i+1} = Res'_{i+1}$ is easily achieved by applying several times the induction hypothesis and the distributivity of e_{body}:

$$\begin{aligned} Res_{i+1} &= e_{body}(Res_i) \texttt{ union } Res_i \\ &= e_{body}(Res'_i) \texttt{ union } Res_i \\ &= e_{body}(\Theta_i \texttt{ union } Res'_{i-1}) \texttt{ union } Res_i \\ &= e_{body}(\Theta_i) \texttt{ union } e_{body}(Res'_{i-1}) \texttt{ union } Res_i \\ &= e_{body}(\Theta_i) \texttt{ union } e_{body}(Res_{i-1}) \texttt{ union } Res_i \\ &= e_{body}(\Theta_i) \texttt{ union } Res_i \texttt{ union } Res_i \\ &= e_{body}(\Theta_i) \texttt{ union } Res_i \\ &= e_{body}(\Theta_i) \texttt{ union } Res'_i \\ &= Res'_{i+1} \ . \end{aligned} \tag{8.9}$$

Note that all the expressions above are unions of sequences of nodes, thus we can safely replace all set-equalities with strict equalities. QED

8.4.8. Remark. Any Regular XPath expression e^+ is equivalent to the IFP expression $\texttt{with \$x seeded by . recurse \$x/}e$. Moreover, *Delta* correctly evaluates this expression.

Proof. Any Regular XPath expression $\texttt{\$x/}e$ is of the form described by *(i)* to *(iii)* in Proposition 8.4.3, thus it is distributive for $\texttt{\$x}$. Then, by Theorem 8.4.7, e^+ is equivalent to $\texttt{with \$x seeded by . recurse \$x/}e$. By Theorem 8.4.6, *Delta* correctly computes e^+. QED

Distributivity gives us a clean translation of the TC operator into the IFP operator. In the general case, we do not know whether there is a translation from XQuery$^{\text{-rudf,+tc}}$ into XQuery$^{\text{-rudf,+ifp}}$ and vice versa. In Section 8.8, we will discuss

related work on the expressive power of the TC and IFP operators in the context of XPath.

We have seen two benefits of distributivity: distributive TC expressions can be translated into distributive IFP expressions, and then *Delta* can be applied for their evaluation. Further, we need to be able to determine which expressions are distributive. Unfortunately, in the next section, we show that the distributivity property is undecidable.

8.4.4 Undecidability of the distributivity property

When we plan the evaluation of `with $x seeded by` e_{seed} `recurse` e_{body}, knowing the answer to "*Is* e_{body} *distributive for* `$x`*?*" allows us to apply *Delta* for computing the answer. Can we always answer this question, i.e., is there an implementable characterization—sufficient and necessary conditions—for the distributivity property? The answer is *no*.

8.4.9. THEOREM. *The problem of determining whether a given XQuery expression* e(`$x`) *is distributive for* `$x` *is undecidable.*

Proof. Consider two arbitrary expressions e_1, e_2, in which `$x` does not occur free. If an XQuery processor could assess whether `if (deep-equal(`e_1`,`e_2`)) then $x else $x[1]` is distributive for `$x`, it could also decide the equivalence of e_1 and e_2. Since the equivalence problem for XQuery is undecidable (as follows from the Turing-completeness of the language [Kepser, 2004]), determining whether an expression is distributive with respect to some variable is also undecidable. QED

In spite of this negative result, in practice, safe approximations of distributivity are still worth while to consider. In the next two sections, we present two such approximations, one at the syntactic level of XQuery, and the other at the level of an algebra underlying XQuery.

8.5 A syntactic approximation of distributivity

In this section, we define a syntactic fragment of XQuery parametrized by a variable `$x`, called *the distributivity-safe fragment*. This syntactic fragment bares analogy with the syntactic fragment defined by the linearity condition in SQL:1999 (see Section 8.2.2). We show that all expressions that are distributivity-safe for a variable `$x` are distributive for that variable. Moreover, membership in this fragment can be determined in linear time with respect to the size of the expression. Since distributivity is undecidable (see Section 8.4.4), the distributivity-safe fragment does not contain all distributive expressions. We give an example of a distributive expression that is not distributivity-safe. Nevertheless, we show that this fragment is *expressively complete* for distributivity, i.e., any distributive

XQuery expression is expressible in the distributivity-safe fragment of XQuery. Moreover, membership in this fragment can be determined in linear time with respect to the size of the expression.

In Section 8.5.1, we define the distributivity-safe fragment. We state the soundness of the fragment with respect to distributivity. We also state and prove the expressive completeness. In Section 8.5.2, we provide the proof of soundness.

8.5.1 The distributivity-safe fragment of XQuery

In the following, we define the distributivity-safe fragment of XQuery that implies distributivity. We consider LiXQuery [Hidders *et al.*, 2004], a simplified version of XQuery. Then we define an auxiliary fragment of LiXQuery, the *position and last guarded* fragment. Finally, we define the *distributivity-safe* fragment as a fragment of LiXQuery.

LiXQuery is a simplified version of XQuery that preserves Turing-completeness. LiXQuery has a simpler syntax and data model than XQuery. It includes the most important language constructs, 3 basic types of items: `xs:boolean`, `xs:string`, and `xs:integer`; and 4 types of nodes: `element()`, `attribute()`, `text()`, and `document-node()`. The syntax of LiXQuery is given in Figure A.1 in Appendix A. This language has a well-defined semantics and it was designed as a convenient tool for studying properties of the XQuery language. We consider *static-type*$(\cdot)$ to be the mapping of LiXQuery expressions to their *static type*, conform XQuery's formal semantics [World Wide Web Consortium, 2007b].

Let LiXQuery^{-nc} be the fragment of LiXQuery without node constructors. We define an auxiliary fragment of LiXQuery^{-nc} that contains the built-in functions `position()` and `last()` only as subexpressions of the second argument of the path and filter operators. In Section 8.5.2, we will relate this fragment to a semantic notion of *context position and size independence*.

8.5.1. Definition (Position and Last Guarded). An XQuery expression e is called *position and last guarded*, *plg*(e), if it can be generated using the syntactic rules in Figure 8.6.

The inference rules Atomic and Closure in Figure 8.6 define the LiXQuery^{-nc} fragment that does not contain `position()` and `last()` at all, while the rules Path and Filter allow these two functions as subexpressions of the second argument of the path and filter operators.

Using the position and last guarded fragment, we define the *distributivy-safe* fragment of LiXQuery^{-nc}. But first, we give an intuition of this fragment.

Intuitively, we may apply a divide-and-conquer evaluation strategy for an expression $e($`$x`$), if any subexpression of e processes the nodes in `$x` one by one. The most simple example of such a subexpression is `for $y in $x return` $e($`$y`$)$, where e is in LiXQuery^{-nc} and `$x` does not occur free in e. On the other hand, we may *not* apply a divide-and-conquer evaluation strategy if any subexpression

$$\frac{e\text{–atomic} \quad e \neq \texttt{position()} \quad e \neq \texttt{last()}}{plg(e)}(\textsc{Atomic})$$

$$\frac{\oslash \in \{\texttt{/}, \texttt{//}\} \quad plg(e_1)}{plg(e_1 \oslash e_2)}(\textsc{Path}) \qquad \frac{plg(e_1)}{plg(e_1\texttt{[}e_2\texttt{]})}(\textsc{Filter})$$

$$\frac{\odot\text{–any operator or function name} \quad plg(e_i), 1 \leq i \leq n}{plg(\odot\,(e_1, \ldots, e_n))}(\textsc{Closure})$$

Figure 8.6: Position and last guarded $plg(\cdot)$: A fragment of LiXQuery^{-nc} that contains `position()` and `last()` only in the second argument of the path and filter operators.

$$\frac{}{ds_{\$\texttt{x}}\,(\$\texttt{x})}(\textsc{Var}) \quad \frac{\$\texttt{x} \notin fv(e)}{ds_{\$\texttt{x}}\,(e)}(\textsc{Const}) \quad \frac{\oplus \in \{\texttt{,}, \texttt{|}\} \quad ds_{\$\texttt{x}}\,(e_1) \quad ds_{\$\texttt{x}}\,(e_2)}{ds_{\$\texttt{x}}\,(e_1 \oplus e_2)}(\textsc{Concat})$$

$$\frac{\$\texttt{x} \notin fv(e_1) \quad ds_{\$\texttt{x}}\,(e_2) \quad ds_{\$\texttt{x}}\,(e_3)}{ds_{\$\texttt{x}}\,(\texttt{if (}e_1\texttt{) then } e_2 \texttt{ else } e_3)}(\textsc{If})$$

$$\frac{\$\texttt{x} \notin fv(e_1) \quad ds_{\$\texttt{x}}\,(e_2)}{ds_{\$\texttt{x}}\,(\texttt{for } \$v \texttt{ at } \$p \texttt{ in } e_1 \texttt{ return } e_2)}(\textsc{For1}) \quad \frac{ds_{\$\texttt{x}}\,(e_1) \quad \$\texttt{x} \notin fv(e_2)}{ds_{\$\texttt{x}}\,(\texttt{for } \$v \texttt{ in } e_1 \texttt{ return } e_2)}(\textsc{For2})$$

$$\frac{\$\texttt{x} \notin fv(e_0) \quad ds_{\$\texttt{x}}\,(e_i)_{1\leq i\leq n+1}}{ds_{\$\texttt{x}}\left(\begin{array}{l}\texttt{typeswitch}\,(e_0)\\ \quad\texttt{case } \tau_1 \texttt{ return } e_1\\ \qquad\vdots\\ \quad\texttt{case } \tau_n \texttt{ return } e_n\\ \quad\texttt{default return } e_{n+1}\end{array}\right)}(\textsc{TypeSw})$$

$$\frac{\$\texttt{x} \notin fv(e_1) \quad ds_{\$\texttt{x}}\,(e_2)}{ds_{\$\texttt{x}}\,(\texttt{let } \$v \texttt{ := } e_1 \texttt{ return } e_2)}(\textsc{Let1})$$

$$\frac{ds_{\$\texttt{x}}\,(e_1) \quad \$\texttt{x} \notin fv(e_2) \quad ds_{\$v}\,(e_2)}{ds_{\$\texttt{x}}\,(\texttt{let } \$v \texttt{ := } e_1 \texttt{ return } e_2)}(\textsc{Let2})$$

$$\frac{\oslash \in \{\texttt{/}, \texttt{//}\} \quad \$\texttt{x} \notin fv(e_1) \quad ds_{\$\texttt{x}}\,(e_2)}{ds_{\$\texttt{x}}\,(e_1 \oslash e_2)}(\textsc{Path1})$$

$$\frac{\oslash \in \{\texttt{/}, \texttt{//}\} \quad ds_{\$\texttt{x}}\,(e_1) \quad \$\texttt{x} \notin fv(e_2) \quad plg(e_2)}{ds_{\$\texttt{x}}\,(e_1 \oslash e_2)}(\textsc{Path2})$$

$$\frac{ds_{\$\texttt{x}}\,(e_1) \quad \$\texttt{x} \notin fv(e_2) \quad plg(e_2) \quad \textit{static-type}(e_2) = \texttt{xs:boolean}}{ds_{\$\texttt{x}}\,(e_1\texttt{[}e_2\texttt{]})}(\textsc{Filter})$$

$$\frac{\begin{array}{c}\texttt{declare function } f\texttt{(}\$v_1\texttt{,}\ldots\texttt{,}\$v_n\texttt{)}\,\{\,e_0\,\}\\ ds_{\$\texttt{x}}\,(e_i) \wedge (\$\texttt{x} \in fv(e_i) \text{ implies } ds_{\$v_i}\,(e_0)), \text{for each } 1 \leq i \leq n\end{array}}{ds_{\$\texttt{x}}\,(f\texttt{(}e_1\texttt{,}\ldots\texttt{,}e_n\texttt{)})}(\textsc{FunCall})$$

Figure 8.7: Distributivity-safety $ds_{\$\texttt{x}}\,(\cdot)$: A syntactic approximation of the distributivity property for LiXQuery^{-nc} expressions.

of e inspects `$x` as a whole. Examples of such problematic subexpressions are `count($x)` and `$x[1]`, but also the general comparison `$x = 10` which involves existential quantification over the sequence bound to `$x`.

Subexpressions whose value is *independent* of `$x`, on the other hand, are distributive. The only exception of this rule are XQuery's node constructors, *e.g.*, `element` $\{\cdot\}\,\{\cdot\}$, which create new node identities upon each invocation. With `$x` bound to `(<a/>,<b/>)`, for example,

$$\texttt{element}\,\{\,\texttt{"c"}\,\}\,\{\,\texttt{()}\,\} \overset{\mathcal{S}}{\neq} \texttt{for \$y in \$x return element}\,\{\,\texttt{"c"}\,\}\,\{\,\texttt{()}\,\}\ ,$$

since the right-hand side will yield a sequence of two distinct element nodes. This is the reason why we consider the distributivity-safe fragment inside LiXQuery^{-nc}.

8.5.2. Definition (Distributivity-safety). An XQuery expression e is said to be *distributivity-safe for* `$x`, indicated by $ds_{\texttt{\$x}}\,(e)$, if it can be generated using the syntactic rules in Figure 8.7.

The rules Var and Const in Figure 8.7 define the base of the fragment, while the other inference rules define the closure with respect to a number of language operators. Note that the rules For1 and For2 ensure that the recursion variable `$x` occurs either in the body e_2 or in the range expression e_1 of a `for`-iteration but not both. This condition is closely related with the linearity constraint of SQL:1999 (for an in-depth discussion on this, see Section 8.8). A similar remark applies to Rules Let1, Let2, Path1 and Path2. One condition of Filter involves $plg(\cdot)$ defined in Definition 8.5.1, another asks e_2 to be of static type `xs:boolean`. In order to check the latter, an engine should implement *static type checking*. This is an optional functionality (note that the expressive completeness does not depend on this rule). Also note how the rule FunCall requires the distributivity for every function argument of the function body if the recursion variable occurs free in that argument.

8.5.3. Remark. All the rules of Definitions 8.5.1 and 8.5.2 can be checked in parallel with a single traversal of the parse tree of a LiXQuery^{-nc} expression. Checking membership of LiXQuery can thus be done in linear time with respect to the size of an XQuery expression (the size of an expression is the number of subexpressions, which equals the size of the parse tree).

Finally, we can state the property that we most desire of the distributivity-safe fragment, the soundness with respect to distributivity.

8.5.4. Theorem (Soundness). *Any XQuery expression e that is distributivity-safe for a variable* `$x`*, i.e., for which* $ds_{\texttt{\$x}}\,(e)$ *holds, is also distributive for* `$x`.

A formal proof in the settings of LiXQuery is given in Section 8.5.2. The proof is by induction on the structure of the expression.

The distributive-safe fragment does not contain all distributive expressions. For example, `count($x) >= 1` is not distributivity-safe, but still distributive for `$x`. However, it is interesting to note that the distributivity-safe fragment is expressively complete for distributivity.

8.5.5. Proposition (Expressive completeness). *If an XQuery expression* e(`$x`) is distributive for `$x` *and it does not contain node constructors as subexpressions, then it is set-equal to* `for $y in $x return` e(`$y`)*, which is distributivity-safe for* `$x`*.*

Proof. This is a direct consequence of the rule For2 (Figure 8.7) and Proposition 8.4.2. QED

Thus, at the expense of a slight reformulation of the query, we may provide a "syntactic distributivity hint" to an XQuery processor.

In the next section, we provide the proof of Theorem 8.5.4 in the setting of LiXQuery.

8.5.2 Distributivity-safety implies distributivity

In this section we prove the soundness of the distributivity-safety rules with respect to the distributivity property in the context of LiXQuery. The result transfers directly to XQuery, since LiXQuery is its fragment. Before proceeding with the proof of Theorem 8.5.4, we first cover the basics of the LiXQuery semantics. The complete definition can be found in [Hidders *et al.*, 2004].

LiXQuery in nutshell

An *XML store*, denoted by St, contains the XML documents and collections that are queried, and also the XML fragments that are created during the evaluation of an expression. The *query evaluation environment* of an XML store, denoted by $En = (\mathbf{a}, \mathbf{b}, \mathbf{v}, \mathbf{x}, \mathbf{k}, \mathbf{m})$, consists of:

- a partial function **a** that maps a function name to its formal arguments;
- a partial function **b** that maps a function name to its body;
- a partial function **v** that maps variable names to their values;
- **x**, which is undefined or an item of the XML store, and indicates the context item;
- **k**, which is undefined or an integer and gives the position of the context item in the context sequence;

- **m**, which is undefined or an integer and gives the size of the context sequence.

We use $En[\mathbf{a}(n) \mapsto y]$ ($En[\mathbf{b}(n) \mapsto y]$, $En[\mathbf{v}(n) \mapsto y]$) to denote the environment that is equal to En except that the function **a** (**b**, **v**) maps the name n to the item y. Similarly, we let $En[\mathbf{x} \mapsto y]$ ($En[\mathbf{k} \mapsto z]$, $En[\mathbf{m} \mapsto z]$) denote changing the environment En only by attributing **x** a new item y (attributing **k**, **m** a new integer value z).

We denote a sequence of items by $S = (y_1, y_2, \ldots, y_n)$, the empty sequence by $()$, and the concatenation of two sequences S_1 and S_2 by $S_1 \circ S_2$. The set of items in a sequence S is $\mathbf{Set}(S)$. Given a sequence of nodes S in an XML store St, we denote $\mathbf{Ord}_{St}(S)$ to be the unique sequence $S' = (y'_1, y'_2, \ldots, y'_m)$, such that $\mathbf{Set}(S) = \mathbf{Set}(S')$ and $y'_1 \ll_{St} \cdots \ll_{St} y'_m$, where $\ll_{St}$ is a total order on the nodes of the store St denoting the *document order*. Using this notation, we can rewrite the equivalence (SetEq) into: $S \overset{s}{=} S'$ iff $\mathbf{Ord}_{St}\big(\mathbf{Set}(S)\big) = \mathbf{Ord}_{St}\big(\mathbf{Set}(S')\big)$.

Further, the semantics of LiXQuery is defined by a set of *semantic rules*. We write $St, En \vdash e \Rightarrow V$ to denote that the evaluation of expression e against the XML store St and environment En of St results in a sequence V of values of St.[4] For an example of a semantic rule, let us take the Concatenation Rule [Hidders *et al.*, 2004]:

$$\frac{St, En \vdash e_1 \ \Rightarrow V' \quad St', En \vdash e_2 \ \Rightarrow V''}{St, En \vdash e_1\,\texttt{,}\,e_2 \ \Rightarrow V' \circ V''}$$

Given this, we can write the definition of distributivity for `$x` in terms of LiXQuery semantics. Let St be a store, $En[\mathbf{v}(x) \mapsto (x_1, \ldots, x_n)]$ an environment that binds `$x` to a non-empty sequence of items $(x_1, \ldots, x_n)$. Applying the LiXQuery semantic rules on both sides of Eq. (8.2) of Definition 8.4.1 we obtain the following: $V_1 \circ \cdots \circ V_n \overset{s}{=} V$, where $St, En[\mathbf{v}(x) \mapsto (x_1, \ldots, x_n)] \vdash e \Rightarrow V$ and $St, En[\mathbf{v}(x) \mapsto x_i] \vdash e \Rightarrow V_i$, for $1 \leq i \leq n$. Thus, e is *distributive for* `$x` if $\mathbf{Ord}_{St}\big(\mathbf{Set}(V_1 \circ \cdots \circ V_n)\big) = \mathbf{Ord}_{St}\big(\mathbf{Set}(V)\big)$.

One last remark is that the path operators `/` and `//` are defined to be *left associative*, i.e., $e_1/e_2/e_3$ means $(e_1/e_2)/e_3$.

Proving the soundness

Before giving the proof of Theorem 8.5.4 we define a notion of *context position and size independence* and prove two lemmas.

8.5.6. Definition (Context Position and Size Independence). A LiXQuery expression e is *context position and size independent (c.p. and s. ind.)* if

[4] In fact, the semantic rules of LiXQuery are of the form $St, En \vdash e \Rightarrow (St', V)$, where St' might be a new XML store and V is a sequence of values of St'. But since we consider LiXQuery without node constructors, the evaluation of expression e against the XML store St and the environment En results always in the same XML store St and a sequence V of values of St.

for any XML store St, environment En, and any sequence of items V, we have $St, En \vdash e \Rightarrow V$ if and only if $St, En[\mathbf{m} \mapsto 1][\mathbf{k} \mapsto 1] \vdash e \Rightarrow V$.

In other words, e is c.p. and s. ind. if no matter what the values for the environment parameters **m** and **k** are, e evaluates to the same result as if **m** and **k** are set to 1. An expression that is c.p. and s. ind. does not use other information about the context sequence than the context item. We can interpret this property as distributivity for the context sequence.

The expression `a/b/c` is obviously c.p. and s. ind., while `position()>1` is not, if the context sequence contains more than one item. In fact, the *position and last guarded* fragment defined by $plg(\cdot)$ (Definition 8.5.1) is an approximation of the *context position and size independent* fragment of LiXQuery:

8.5.7. Lemma. *Any LiXQuery expression e that is position and last guarded, i.e., $plg(e)$, is also c.p. and s. independent.*

Proof. The proof goes by induction on the structure of e. The base case consists of checking the implication for atomic expressions satisfying the rule Atomic in Figure 8.6. Below, we list all atomic expressions in LiXQuery, grouped by clause in the BNF definition of the language:

$\langle Var \rangle$:	"`$`" $\langle Name \rangle$
$\langle BuiltIn \rangle$:	"`true()`", "`false()`", "`position()`", "`last()`"
$\langle Step \rangle$:	"`.`", "`..`", $\langle Name \rangle$, "`@`" $\langle Name \rangle$, "`*`", "`@*`", "`text()`"
$\langle Literal \rangle$:	$\langle String \rangle, \langle Integer \rangle$
$\langle EmpSeq \rangle$:	"`()`"

By the conditions of Rule Atomic, e cannot be `position()` or `last()`. None of the semantic rules for the remaining atomic expressions refer to **k** or **m**, thus e evaluates to the same sequence of items irrespective of the value of these parameters: $St, En \vdash e \Rightarrow V$ and $St, En[\mathbf{k} \mapsto 1][\mathbf{m} \mapsto 1] \vdash e \Rightarrow V$, for any St and En.

Next, we prove the induction step for the expressions defined by Rules Path, Filter and Closure.

Path and Filter expressions. Let $e = e_1 \oslash e_2$, where $\oslash \in \{\texttt{/}, \texttt{//}, \texttt{[]}\}$, and let $plg(e)$. By the rules Path and Filter, $plg(e_1)$. Suppose also that e_1 is c.p. and s. independent. We prove that $St, En \vdash e_1 \oslash e_2$ is equivalent with $St, En[\mathbf{k} \mapsto 1][\mathbf{m} \mapsto 1] \vdash e_1 \oslash e_2$. By the semantic rules for both path and filter expressions (see Rules (18) and (17), in [Hidders *et al.*, 2004]), e_1 is evaluated first: $St, En \vdash e_1 \Rightarrow (x_1, \ldots, x_m)$. Since e_1 is c.p. and s. independent, $St, En[\mathbf{k} \mapsto 1][\mathbf{m} \mapsto 1] \vdash e_1 \Rightarrow (x_1, \ldots, x_m)$. Further, e_2 is evaluated for each item in the result sequence of e_1: $St, En[\mathbf{x} \mapsto x_i][\mathbf{k} \mapsto i][\mathbf{m} \mapsto m] \vdash e_2$, $1 \leq i \leq m$. Note that the values of **k** and **m** are changed by the semantics of these operators and that the result of the evaluation of e_2 does not depend on the initial context position

and size. Thus, no matter which of the three operators we consider, the end result of e evaluated against St and En is the same as evaluated against St and $En[\mathbf{k} \mapsto 1][\mathbf{m} \mapsto 1]$.

Other expressions. Let $e = \odot(e_1, \ldots, e_n)$, $n \geq 1$ be a complex expression, where $\odot(\cdot, \ldots, \cdot)$ is any operator or function in the language. Suppose $plg(e)$ then by the rule CLOSURE, $plg(e_i)$, $1 \leq i \leq n$. Suppose also that e_i is c.p. and s. independent. For $\odot$ equal to one of the path operators or the filter operator, we have already proved that e is c.p. and s. independent. The semantic rules of the remaining operators and functions in the languages do not refer to the parameters $\mathbf{k}$ and $\mathbf{m}$, thus e is trivially c.p. and s. independent. QED

8.5.8. LEMMA. *For any LiXQuery expression e and variable $\$x \notin fv(e)$, e is distributive for $\$x$.*

Proof. Let St be an XML store and En an environment that binds $\$x$ to a non-empty sequence of size n. Suppose that $St, En \vdash e \Rightarrow (x_1, \ldots, x_m)$. In this case, the result of the corresponding for-expression is a sequence constructed by concatenating $(x_1, \ldots, x_m)$ n times: $St, En \vdash$ `for $y in $x return` $e \Rightarrow (x_1, \ldots, x_m) \circ (x_1, \ldots, x_m) \circ \cdots \circ (x_1, \ldots, x_m)$, which is set-equal to $(x_1, \ldots, x_m)$. Note that $\$y \notin fv(e)$ and the result of e does not depend on the binding of $\$y$. QED

Proof of Theorem 8.5.4. As before, the proof goes by induction on the structure of e.

The base case consists of checking the implication for atomic expressions and for expressions that do not contain $\$x$ as a free variable (constant w.r.t. $\$x$). First, suppose e is an expression for which $\$x \notin fv(e)$. By the rule CONST in Figure 8.7, e is distributivity-safe for $\$x$ and by Lemma 8.5.8, e is distributive for $\$x$. Second, suppose $e = \$x$, distributivity-safe for $\$x$ by the rule VAR. Let St be a store and En an environment that binds $\$x$ to the non-empty sequence of items $(x_1, \ldots, x_n)$, then $St, En \vdash \$x \Rightarrow (x_1, \ldots, x_n)$ and $St, En \vdash$ `for $y in $x return $y` $\Rightarrow (x_1) \circ \cdots \circ (x_n)$. Thus expression e is distributive for $\$x$.

The induction step consists of checking the implication for the complex expressions defined by the rest of the distributivity-safety rules in Figure 8.7. The induction hypothesis (IH) is: any distributive-safe for $\$x$ subexpression of e is distributive for $\$x$. We suppose that $\$x \in fv(e)$, otherwise e was already considered in the base case. Further, let St be a store, En an environment that binds $\$x$ to the non-empty sequence of items $(x_1, \ldots, x_n)$.

If expressions. Suppose $e =$ `if (`e_1`) then` e_2 `else` e_3 and $ds_{\$x}(e)$, then by the rule IF: $ds_{\$x}(e_2)$, $ds_{\$x}(e_3)$, and $\$x \notin fv(e_1)$. Let $St, En \vdash e_1 \Rightarrow b$, $St, En \vdash e_2 \Rightarrow V$ and $St, En \vdash e_3 \Rightarrow V'$, where b is a boolean value, V and V' are sequences of items.

Suppose b is *true*, then $St, En \vdash e \Rightarrow V$ (otherwise, $St, En \vdash e \Rightarrow V'$). By Lemma 8.5.8, e_1 is distributive for $\$x$, thus it yields the same boolean value for any

binding of $x to a singleton x_i: $St, En[\mathbf{v}(x) \mapsto x_i] \vdash e_1 \Rightarrow true$, $1 \leq i \leq n$. From this we obtain: if $St, En[\mathbf{v}(x) \mapsto x_i] \vdash e_2 \Rightarrow V_i$ then $St, En[\mathbf{v}(x) \mapsto x_i] \vdash e \Rightarrow V_i$, for any $1 \leq i \leq n$. By the IH, e_2 is distributive for $x, so $V \overset{s}{=} V_1 \circ \cdots \circ V_n$. The reasoning is identical when b is $false$, thus e is distributive for $x.

Type-switch expressions. The proof for the type-switch expressions that are defined by Rule (TypeSw) is similar to the proof for if-expressions.

Path expressions. Suppose $e = e_1/e_2$ (the case for '//' is identical) and $ds_{\$x}(e)$, then e must satisfy Rule Path1 or Rule Path2.

Suppose e satisfies Rule Path1, then $x \notin fv(e_1)$ and $ds_{\$x}(e_2)$. First, let $St, En \vdash e_1 \Rightarrow (y_1, \ldots, y_m)$ and, since e_1 is constant w.r.t. $x, $St, En[\mathbf{v}(x) \mapsto x_i] \vdash e_1 \Rightarrow (y_1, \ldots, y_m)$, $1 \leq i \leq n$. Second, let $St, En[\mathbf{x} \mapsto y_j] \vdash e_2 \Rightarrow V_j$ and $St, En[\mathbf{v}(x) \mapsto x_i][\mathbf{x} \mapsto y_j] \vdash e_2 \Rightarrow V_{i,j}$, $1 \leq i \leq n$, $1 \leq j \leq m$. By the IH, e_2 is distributive for $x and thus, $V_j \overset{s}{=} V_{1,j} \circ \cdots \circ V_{n,j}$, for $1 \leq j \leq m$. Following the semantic rule for path expressions in [Hidders *et al.*, 2004], $St, En \vdash e \Rightarrow \mathbf{Ord}_{St}\big(\bigcup_{1\leq j\leq m} \mathbf{Set}(V_j)\big)$ and $St, En[\mathbf{v}(x) \mapsto x_i] \vdash e \Rightarrow \mathbf{Ord}_{St}\big(\bigcup_{1\leq j\leq m} \mathbf{Set}(V_{i,j})\big)$. Finally, from the distributivity for $x of e_2, it follows that

$$\mathbf{Ord}_{St}\Big(\bigcup_{1\leq j\leq m} \mathbf{Set}(V_j)\Big) \overset{s}{=} \mathbf{Ord}_{St}\Big(\bigcup_{1\leq j\leq m} \mathbf{Set}(V_{1,j})\Big) \circ \cdots \circ \mathbf{Ord}_{St}\Big(\bigcup_{1\leq j\leq m} \mathbf{Set}(V_{n,j})\Big),$$

which means that e is distributive for $x.

In the other case, if e satisfies Rule Path2, then $ds_{\$x}(e_1)$, $x \notin fv(e_2)$ and $plg(e_2)$. Let $St, En \vdash e_1 \Rightarrow (y_1, \ldots, y_m)$ and $St, En[\mathbf{v}(x) \mapsto x_i] \vdash e_1 \Rightarrow (y_1^i, \ldots, y_{m_i}^i)$, $1 \leq i \leq n$. By the IH, e_1 is distributive for $x, thus $(y_1, \ldots, y_m) \overset{s}{=} (y_1^1, \ldots, y_{m_1}^1) \circ \cdots \circ (y_1^n, \ldots, y_{m_n}^n)$, which means that for any $1 \leq k \leq m$, exists $1 \leq i \leq n$ and $1 \leq j \leq m_i$, such that $y_k = y_j^i$, and vice versa, for any $1 \leq i \leq n$ and $1 \leq j \leq m_i$, exists $1 \leq k \leq m$ with the same property. Next, let $St, En[\mathbf{x} \mapsto y_k] \vdash e_2 \Rightarrow V_k$ and $St, En[\mathbf{x} \mapsto y_j^i] \vdash e_2 \Rightarrow V_{i,j}$, for respective k, i, and j. Note that during the evaluation of e_2 we disregard the values of the parameters $\mathbf{k}$ and $\mathbf{m}$, and the binding of the variable $x: the former is allowed by the fact that $plg(e_2)$ and, by Lemma 8.5.7, e_2 is c.p. and s. independent; the latter is allowed by the fact that e_2 is constant w.r.t. $x. Finally, observe that for any $1 \leq k \leq m$, there is $1 \leq i \leq n$ and $1 \leq j \leq m_i$, such that $V_k = V_{i,j}$, and vice versa. This implies that $\mathbf{Ord}_{St}\big(\bigcup_{1\leq k\leq m} \mathbf{Set}(V_k)\big) \overset{s}{=} \mathbf{Ord}_{St}\big(\bigcup_{1\leq j\leq m_1} \mathbf{Set}(V_{1,j})\big) \circ \cdots \circ \mathbf{Ord}_{St}\big(\bigcup_{1\leq j\leq m_n} \mathbf{Set}(V_{n,j})\big)$, which means that e is distributive for $x.

For expressions. Suppose e is a for-expression and $ds_{\$x}(e)$, then e must satisfy Rule For1 or Rule For2 in Figure 8.7. If e satisfies Rule For1, then the proof is similar to the proof for the path expressions satisfying Rule Path1. And if e satisfies Rule For2, then the proof is similar to the one for path expressions satisfying Rule Path2. Note that such expressions do not contain the positional variable "`at $p`". This condition forms the counterpart of the position and last guarded condition in the case of path expressions.

Let expressions. Suppose $e =$ `let $v :=` e_1 `return` e_2 and $ds_{\$x}(e)$, then e must satisfy Rule LET1 or Rule LET2.

Suppose e satisfies Rule LET1, then $\$x \notin fv(e_1)$ and $ds_{\$x}(e_2)$. By the IH, e_2 is distributive for `$x` and the distributivity for `$x` of e follows straightforwardly.

Suppose e satisfies Rule LET2, then $ds_{\$x}(e_1)$, $\$x \notin fv(e_2)$ and $ds_{\$v}(e_2)$. By the IH, e_1 is distributive for `$x` and e_2 is distributive for `$v`. Let $St, En \vdash e_1 \Rightarrow (y_1, \ldots, y_m)$ and $St, En[\mathbf{v}(x) \mapsto x_i] \vdash e_1 \Rightarrow (y_1^i, \ldots, y_{m_i}^i)$, $1 \leq i \leq n$. Since e_2 is distributive for `$x`, $(y_1, \ldots, y_m) \overset{s}{=} (y_1^1, \ldots, y_{m_1}^1) \circ \cdots \circ (y_1^n, \ldots, y_{m_n}^n)$, which means that for any $1 \leq k \leq m$, exists $1 \leq i \leq n$ and $1 \leq j \leq m_i$, such that $y_k = y_j^i$, and vice versa, for any $1 \leq i \leq n$ and $1 \leq j \leq m_i$, exists $1 \leq k \leq m$ with the same property. Further, let $St, En[\mathbf{v}(v) \mapsto (y_1, \ldots, y_m)] \vdash e_2 \Rightarrow V$ and $St, En[\mathbf{v}(v) \mapsto y_k] \vdash e_2 \Rightarrow V_k$, $1 \leq k \leq m$. Since e_2 is distributive for `$v`, it follows that $V \overset{s}{=} V_1 \circ \cdots \circ V_m$. Last, let $St, En[\mathbf{v}(x) \mapsto x_i][\mathbf{v}(v) \mapsto (y_1^i, \ldots, y_{m_i}^i)] \vdash e_2 \Rightarrow V^i$ and $St, En[\mathbf{v}(x) \mapsto x_i][\mathbf{v}(v) \mapsto y_j^i] \vdash e_2 \Rightarrow V_j^i$, for $1 \leq i \leq n$, $1 \leq j \leq m_i$. Note that since e_2 is constant w.r.t. `$x`, the binding of this variable does not influence the result of e_2. And again, since e_2 is distributive for `$v`, it follows that $V^i \overset{s}{=} V_1^i \circ \cdots \circ V_{m_i}^i$, $1 \leq i \leq n$. We saw before that for any k, there exist i and j, such that $y_k = y_j^i$, which implies $V_k = V_j^i$, and vice versa, for any i and j, there exists a k, with the same property. This, finally, implies that $V \overset{s}{=} V_1 \circ \cdots \circ V_m \overset{s}{=} V_1^1 \circ \cdots \circ V_{m_1}^1 \circ \cdots \circ V_1^n \circ \cdots \circ V_{m_n}^n \overset{s}{=} V^1 \circ \cdots \circ V^n$, which means that e is distributive for `$x`.

Filter expressions. Suppose $e = e_1$`[`e_2`]` and $ds_{\$x}(e)$, then by the rule FILTER: $ds_{\$x}(e_1)$, $\$x \notin fv(e_2)$, $plg(e_2)$ and e_2 is static type `xs:boolean`. Let $St, En \vdash e_1 \Rightarrow (y_1, \ldots, y_m)$ and $St, En[\mathbf{v}(x) \mapsto x_i] \vdash e_1 \Rightarrow (y_1^i, \ldots, y_{m_i}^i)$, $1 \leq i \leq n$. By the IH, e_1 is distributive for `$x`, thus $(y_1, \ldots, y_m) \overset{s}{=} (y_1^1, \ldots, y_{m_1}^1) \circ \cdots \circ (y_1^n, \ldots, y_{m_n}^n)$, which means that for any $1 \leq k \leq m$, there exists $1 \leq i \leq n$ and $1 \leq j \leq m_i$, such that $y_k = y_j^i$, and vice versa, for any $1 \leq i \leq n$ and $1 \leq j \leq m_i$, there exists $1 \leq k \leq m$ with the same property. Next, let $St, En[\mathbf{x} \mapsto y_k] \vdash e_2 \Rightarrow b_k$ and $St, En[\mathbf{x} \mapsto y_j^i] \vdash e_2 \Rightarrow b_j^i$, where b_k and b_j^i are booleans, for all k, i, and j. Note that during the evaluation of e_2 we disregard the values of the parameters $\mathbf{k}$ and $\mathbf{m}$, and the binding of the variable `$x`: the former is allowed by the fact that $plg(e_2)$ and, by Lemma 8.5.7, e_2 is c.p. and s. independent; the latter is allowed by the fact that e_2 is constant w.r.t. `$x`. It is clear that if $y_k = y_j^i$ then $b_k = b_j^i$. This means that y_k is contained in the result of e evaluated against St and En, iff y_j^i is contained in the concatenation of the results of e evaluated against St and $En[\mathbf{v}(x) \mapsto x_i]$. Thus, the respective result sequences are set-equal and e is distributive for `$x`.

Other expressions. Suppose $e = e_1 \oplus e_2$, where $\oplus \in \{$`,`, `|`$\}$ or $e = f(e_1, \ldots, e_l)$, a function call, and $ds_{\$x}(e)$. By the rules CONCAT and FUNCALL, $ds_{\$x}(e_i)$, for all $1 \leq i \leq l$. Then the distributivity for `$x` of e follows directly from the distributivity for `$x` of e_i, which follows from the IH. QED

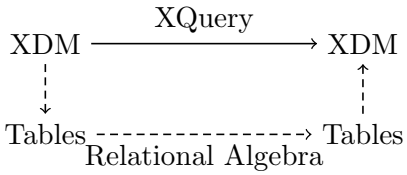


Figure 8.8: Relational (algebra based) approach to XQuery processing.

8.6 An algebraic approximation of distributivity

XQuery is a syntactically rich language with many equivalent ways of expressing the same information need. Reasoning about the queries at the syntactic level is cumbersome. Most XQuery engines adopt algebras as a convenient formalism for query normalization and optimization. One would also expect that reasoning about distributivity is more elegant at the algebraic level.

In this section, we follow an algebraic route for checking the applicability of *Delta* for the evaluation of the IFP of an XQuery expression e_{body}. We adopt a relational approach to XML data modeling and XQuery evaluation, and instead of performing the distributivity test at the syntactic level, we inspect the *relational algebraic query plan* compiled for e_{body}. As we would expect, the algebraic representation of e_{body} renders the check for the distributivity property particularly robust and simple.

In the following, we first sketch the relational approach to XQuery that we follow. Then we define an algebraic distributivity property that is equivalent to the XQuery distributivity property and present an incomplete but effective test for it. We also discuss this approach in comparison with the syntactic approach presented in the previous section.

Relational XQuery. The alternative route we take in this section builds on the *Pathfinder* project, which fully implements a purely relational approach to XQuery. *Pathfinder* compiles instances of the XQuery Data Model (XDM) and XQuery expressions into relational tables and algebraic plans over these tables, respectively, and thus follows the dashed path in Figure 8.8. The translation strategy (*i*) preserves the XQuery semantics (including compositionality, node identity, iteration and sequence order), and (*ii*) yields relational plans which rely on regular relational query engine technology [Grust *et al.*, 2004].

The compiler emits a dialect of relational algebra that mimics the capabilities of modern SQL query engines. The algebra operators are presented in Table 8.1. The row numbering operator $\varrho_{\mathsf{a}:\langle \mathsf{b}_1,\ldots,\mathsf{b}_n\rangle/\mathsf{p}}$ compares with SQL:1999's `ROW_NUMBER() OVER (PARTITION BY p ORDER BY` $\mathsf{b}_1, \ldots, \mathsf{b}_n$`)` and correctly implements the order semantics of XQuery on the (unordered) algebra. Other non-

Operator	Semantics
$\pi_{\mathsf{a}_1:\mathsf{b}_1,\ldots,\mathsf{a}_n:\mathsf{b}_n}$	project onto col.s a_i, rename b_i into a_i
σ_{b}	select rows with column $\mathsf{b} = \mathit{true}$
$\bowtie_p$	join with predicate p
$\dot{\bowtie}_q$	iterated evaluation of rhs argument (`APPLY`)
$\times$	Cartesian product
$\cup$	union
$\setminus$	difference
$\mathtt{count}_{\mathsf{a}:/\mathsf{b}}$	aggregates (group by b, result in a)
$\circledcirc_{\mathsf{a}:\langle \mathsf{b}_1,\ldots,\mathsf{b}_n\rangle}$	n-ary arithmetic/comparison operator $\circ$
$\varrho_{\mathsf{a}:\langle \mathsf{b}_1,\ldots,\mathsf{b}_n\rangle/\mathsf{p}}$	ordered row numbering (by $\mathsf{b}_1,\ldots,\mathsf{b}_n$)
$\text{⏍}_{\alpha::n}$	XPath step join (axis α, node test n)
$\varepsilon, \tau, \ldots$	node constructors
μ, μ^{Δ}	fixpoint operators

Table 8.1: Relational algebra dialect emitted by the Pathfinder compiler.

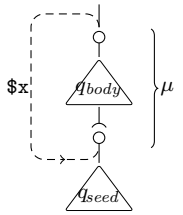
textbook operators, like ε or ⏍, are merely macros representing "micro plans" composed of standard relational operators: expanding $\text{⏍}_{\alpha::n}(q)$, for example, would reveal $\mathsf{doc} \bowtie_p q$, where p is a conjunctive range predicate that realizes the semantics of an XPath location step along axis α with node test n between the context nodes in q and the encoded XML document doc. Dependent joins $\dot{\bowtie}$—also named `CROSS APPLY` in Microsoft SQL Server's SQL dialect *Transact-SQL*—like ⏍ are a logical concept and can be replaced by standard relational operators [Galindo-Legaria and Joshi, 2001].

The plans operate over relational encodings of XQuery item sequences held in flat (1NF) tables with an iter|pos|item schema. In these tables, columns iter and pos are used to properly reflect `for`-iteration and sequence order, respectively. Column item carries encodings of XQuery items, i.e., atomic values or nodes. The inference rules driving the translation procedure from XQuery expressions into algebraic query plans are described in [Grust *et al.*, 2004] and [Afanasiev *et al.*, 2009]. The result is a DAG-shaped query plan where the sharing of sub-plans primarily coincides with repeated references to the same variable in the input XQuery expression. Further details of Relational XQuery do not affect our present discussion of distributivity or IFP evaluation and may be found in [Grust *et al.*, 2004, Afanasiev *et al.*, 2009].

In the following, we extend the algebra with two algebraic fixed point operators corresponding to the fixed point computation algorithms discussed in Section 8.3 and assess distributivity based purely on algebraic equivalences.

8.6.1 An algebraic account of distributivity

An occurrence of the new `with $x seeded by` e_{seed} `recurse` e_{body} form in a source XQuery expression will be compiled into a plan fragment as shown here on the right. In the following, let q denote the algebraic query plan that has been compiled for XQuery expression e. Operator μ, the algebraic representation of the algorithm *Naïve* (Figure 8.4(a)), iterates the evaluation of the algebraic plan for e_{body} and feeds its output ♀ back to its input ⊥ until the IFP is reached. If we can guarantee that the plan for e_{body} is distributive, we may safely trade μ for its *Delta*-based variant μ^{Δ} which, in general, will feed significantly less items back in each iteration (see Figure 8.4(b) and Section 8.7).



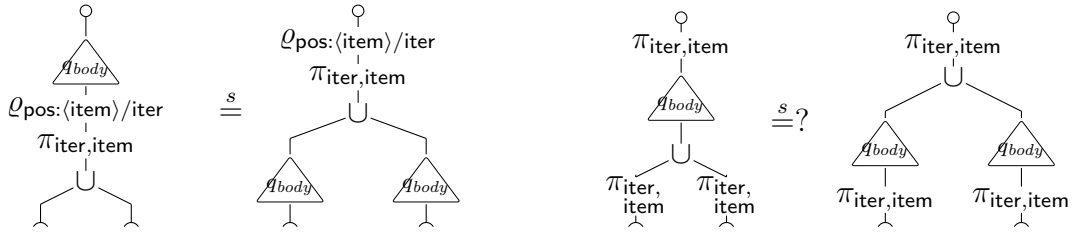
In Section 8.4, we defined the distributivity property of XQuery expressions based on the XQuery operator `union` (see Definition 8.4.1). In the algebraic setting, the XQuery `union` operation is compiled to the following expression that implements the XQuery order requirements—for each iteration the result is ordered by the node rank in column item (see [Afanasiev *et al.*, 2009] for the compilation rule):

$$e_1 \texttt{ union } e_2 \Rightarrow \varrho_{\mathsf{pos}:\langle\mathsf{item}\rangle/\mathsf{iter}}\left(\pi_{\mathsf{iter},\mathsf{item}}\left(q_1 \cup q_2\right)\right) .$$

A straightforward application of this translation to Definition 8.4.1 allows us to express the distributivity criterion based on the equivalence of relational plans. If we can prove the set-equality of the two plans in Figure 8.9(a), we know that the XQuery expression q_{body} must be distributive. This equality is the algebraic expression of the divide-and-conquer evaluation strategy: evaluating e_{body} over a composite input (left-hand side, $\langle \cup \rangle$) yields the same result as the union of the evaluation of e_{body} over a partitioned input (right-hand side).

Given that the distributivity property is undecidable (see Theorem 8.4.9), we propose an effective approximation to distributivity. First, we loosen up the condition expressed in Figure 8.9(a). One prerequisite for distributivity is that the recursion body q_{body} does not inspect sequence positions in its input. Thus, for a distributive q_{body} it must be legal to omit the row-numbering operator $\varrho_{\mathsf{pos}:\langle\mathsf{item}\rangle/\mathsf{iter}}$ in the left-hand side of Figure 8.9(a) and discard all position information in the inputs of sub-plan q_{body} (using $\pi_{\mathsf{iter},\mathsf{item}}$). Further, since the set-equality (used to define distributivity) is indifferent to sequence order, we are also free to disregard the row-numbering operator on top of the right-hand-side plan and place a projection $\pi_{\mathsf{iter},\mathsf{item}}$ on top of both plans to make the order indifference explicit. Proving the equivalence illustrated in Figure 8.9(b), therefore, is sufficient to decide distributivity.

Further, we propose an assessment of distributivity based on algebraic rewrites.



(a) Algebraic equivalent of the distributivity property: e_{body} is distributive if and only if its algebraic counterpart q_{body} satisfies this set-equality.

(b) Distributivity assessment agnostic to sequence order.

Figure 8.9: Algebraic distributivity assessment.

$$\frac{\otimes \in \{\pi, \sigma, \circledcirc, \boxdot\}}{\otimes\,(q_1 \uplus q_2) \rightarrow (\otimes\,(q_1)) \uplus (\otimes\,(q_2))} \quad \text{(Unary)}$$

$$\frac{\otimes \in \{\cup, \times, \bowtie, \overset{:=}{\bowtie}\}}{(q_1 \uplus q_2) \otimes q_3 \rightarrow (q_1 \otimes q_3) \uplus (q_2 \otimes q_3)} \quad \text{(Binary1)}$$

$$\frac{\otimes \in \{\cup, \times, \bowtie, \overset{:=}{\bowtie}\}}{q_1 \otimes (q_2 \uplus q_3) \rightarrow (q_1 \otimes q_2) \uplus (q_1 \otimes q_3)} \quad \text{(Binary2)}$$

$$\frac{}{(q_1 \uplus q_2) \cup (q_3 \uplus q_4) \rightarrow (q_1 \cup q_3) \uplus (q_2 \cup q_4)} \quad \text{(Union)}$$

Figure 8.10: An algebraic approximation of the distributivity property for arbitrary XQuery expressions.

If we can successfully "push" a union operator $\cup$ through the sub-plan q_{body} in the left-hand side of Figure 8.9(b) to obtain the right-hand side, its corresponding XQuery expression e_{body} must be distributive and we can safely trade μ for μ^{Δ} to compute the fixed point.

To this end, we use a set of algebraic *rewrite rules* (Figure 8.10) that try to move a union operator upwards through the DAG. To avoid ambiguity or infinite loops during the rewrite process, we *mark* the union operator (indicated as $\uplus$) in the left-hand-side plan q_{left} of Figure 8.9(b), before we start rewriting. We then exhaustively apply the rule set in Figure 8.10 to each sub-plan in q_{left} in a bottom-up fashion. Since each rule in the set strictly moves the marked union operator upwards inside the plan, termination of the process is guaranteed. Further, the number of operators n in q_{body} is an upper bound for the number of rewrites needed to push $\uplus$ through q_{body}; n itself is bounded by the size of e_{body} (we have seen the same complexity bound for the syntactic analysis of Section 8.5).

Once the rule set does not permit any further rewrites, we compare the rewritten plan q'_{left} with the right-hand side plan q_{right} of Figure 8.9(b) for structural equality. This type of equality guarantees the equivalence of both plans and, hence, the distributivity of e_{body}.

Figure 8.11 shows the rewrites involved to determine the distributivity of e_{body} for Query Q1 (Section 8.2). We place a marked union operator $\uplus$ as the input to the algebraic plan q_{body} obtained for the recursion body of Query Q1. The resulting plan corresponds to the left-hand side of Figure 8.9(b). Applying the equivalence rules UNARY, BINARY1, and again Rule UNARY pushes $\uplus$ up to the plan root, as illustrated in Figures 8.11(b), 8.11(c), and 8.11(d), respectively. The final plan (Figure 8.11(d)) is structurally identical to the right-hand side of Figure 8.9(b), with q_{body} instantiated with the recursion body in Query Q1. We can conclude distributivity for q_{body} and, consequently, for the recursion body in Query Q1.

To prove the soundness of this approach it is enough to acknowledge the correctness of the rewrite rules in Figure 8.10. Once union has been pushed through the algebraic plan of e_{body} and the equality in Figure 8.9(b) holds, we can conclude that the expression is distributive and apply *Delta* for its evaluation. For more details, we refer to [Afanasiev *et al.*, 2009].

8.6.2 Algebraic vs. syntactic approximation

Compared to the syntactic approximation $ds(\cdot)$, the above algebraic account of distributivity draws its conciseness from the fact that the rather involved XQuery semantics and substantial number of built-in functions nevertheless map to a small number of algebraic primitives (given suitable relational encodings of the XDM). Further, for these primitives, the algebraic distributivity property is readily decided.

To make this point, consider this equivalent slight variation of Query Q1 in

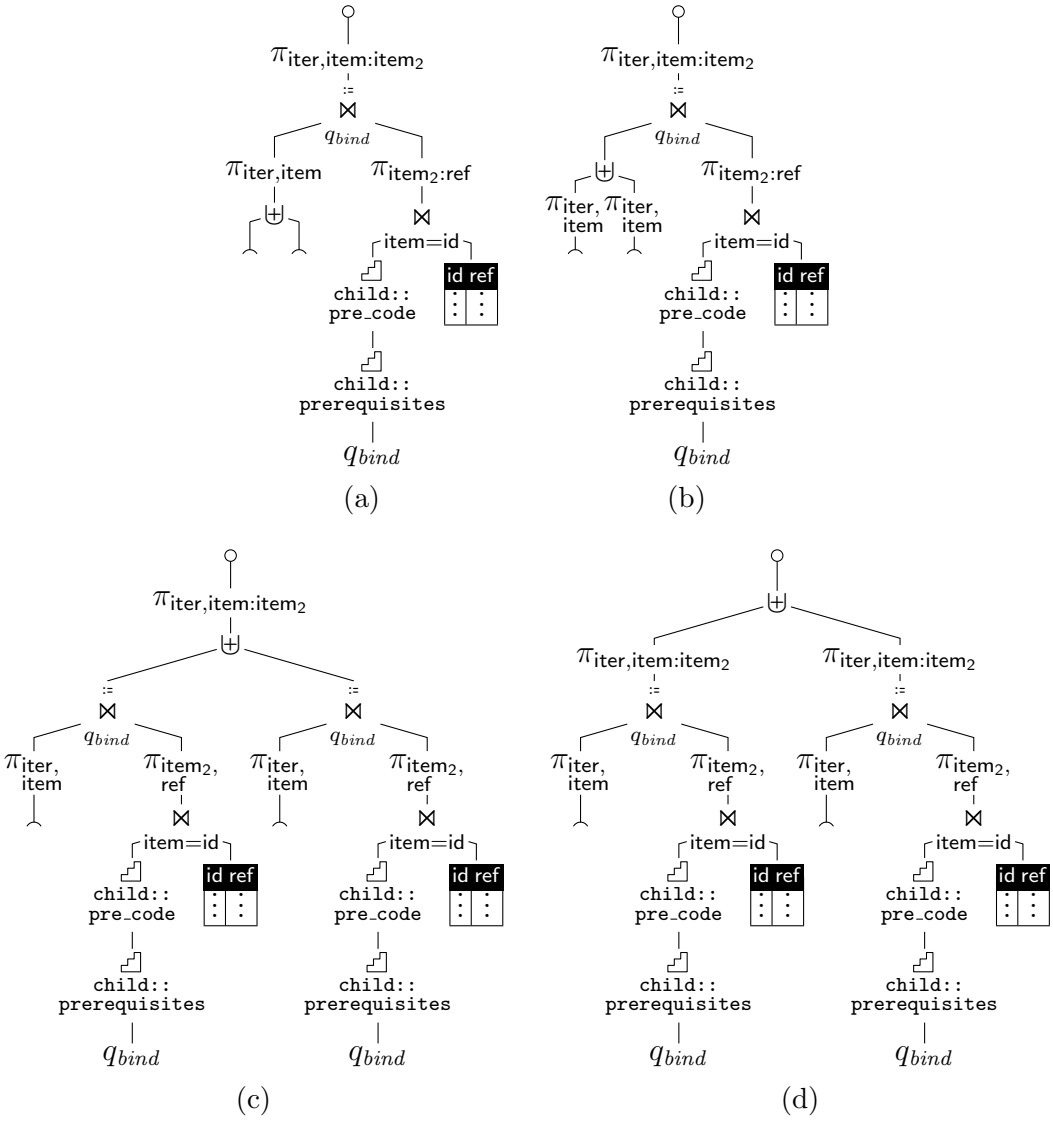


Figure 8.11: The query plan transformation involved in determining the distributivity of e_{body} for Query Q1. The union operator $\uplus$ marks the input to the algebraic plan q_{body} obtained for the recursion body of Query Q1. Applying the equivalence rules Unary, Binary1, and again Rule Unary pushes $\uplus$ up to the plan root, as illustrated in Figures 8.11(b), 8.11(c), and 8.11(d), respectively.

which variable `$x` now occurs free in the argument of function `id(·)`:

```
with $x seeded by
        doc("curriculum.xml")/course[@code="c1"]
recurse id($x/prerequisites/pre_code)  .
```

If we unfold the implementation of the XQuery built-in function `id(·)` (effectively, this expansion is performed when Rule FunCall recursively invokes $ds_{\$x}(\cdot)$ to assess the distributivity of the function body of `id(·)`), we obtain

```
with $x seeded by
        doc("curriculum.xml")/course[@code="c1"]
recurse
     for $c in doc("curriculum.xml")/course
     where $c/@code = $x/prerequisite/pre_code
     return $c  .
```

The syntactic approximation will flag the recursion body as non-distributive because of the presence of the `where` clause (Section 8.5). Even if we rewrite the filter condition using an if-construct, the expression still remains not distributivity-safe due to the occurrence of the variable `$x` in the condition. While the algebraic approach is not affected by the two variations, the rule set of Figure 8.7 needs to be extended with a specific rule for `id(·)` to be able to infer its actual distributivity.

For each syntactic rule in Figure 8.7 we can prove that the corresponding algebraic plan passes the test for distributivity. Thus, the algebraic approach determines a larger fragment of distributive expressions and it is more succinct and easier to work with than the syntactic approach.

In spite of the fact that the approximation is bound to a particular (relational) algebra, we believe that this approach can easily be adapted for other algebras for XQuery.

8.7 Practical impact of distributivity and *Delta*

Exchanging RUDFs for the IFP operator limits the expressive power of the language. However, it also puts the query optimizer in control while the user is spared the trouble of deciding which algorithm should be used for the fix point computation. Trading *Naïve* for *Delta* is a promising optimization and in the previous sections we showed that it can be effectively decided. In this section, we provide experimental evidence that significant gains can indeed be realized, much like in the relational domain.

To quantify the impact, we implemented the two fixed point operator variants μ and μ^{Δ} in *MonetDB/XQuery 0.18* [Boncz *et al.*, 2006a], an efficient and scalable XQuery processor that implements the Relational XQuery approach (Section 8.6). Its algebraic compiler front-end *Pathfinder* has been enhanced (i) to

Query	***MonetDB/XQuery***		***Saxon-SA 8.9***		**Total # of nodes fed to body**		**Recursion depth**
	Naïve	*Delta*	*Naïve*	*Delta*	*Naïve*	*Delta*	
Bidder network (small)	362 ms	165 ms	2,307 ms	1,872 ms	40,254	9,319	10
Bidder network (medium)	5,010 ms	1,995 ms	15,027 ms	7,284 ms	683,225	122,532	16
Bidder network (large)	40,785 ms	13,805 ms	123,316 ms	52,436 ms	5,694,390	961,356	15
Bidder network (huge)	9 m 46 s	176,890 ms	32 m 40 s	12 m 04 s	87,528,919	9,799,342	24
Romeo and Juliet	6,795 ms	1,260 ms	1,150 ms	818 ms	37,841	5,638	33
Curriculum (medium)	183 ms	135 ms	1,308 ms	1,040 ms	12,301	3,044	18
Curriculum (large)	1,466 ms	646 ms	3,485 ms	2,176 ms	127,992	19,780	35
Hospital (medium)	734 ms	497 ms	1,301 ms	1,290 ms	99,381	50,000	5

Table 8.2: Naïve vs. Delta: Comparison of query evaluation times and total number of nodes fed to the recursion body as input.

```
declare variable $doc := doc("auction.xml");

declare function bidder ($in as node()*) as node()*
{ for $id in $in/@id
  let $b := $doc//open_auction[seller/@person = $id]
              /bidder/personref
  return $doc//people/person[@id = $b/@person]
};

for $p in $doc//people/person
return <person>
        { $p/@id }
        { data ((with $x seeded by $p
                 recurse bidder ($x))/@id) }
       </person>
```

Figure 8.12: XMark bidder network query.

process the syntactic form `with` $\cdots$ `seeded by` $\cdots$ `recurse`, and (*ii*) to implement the algebraic distributivity check. All queries in this section were recognized as being distributive by *Pathfinder*. To demonstrate that any XQuery processor can benefit from optimized IFP evaluation in the presence of distributivity, we also performed the transition from *Naïve* to *Delta* on the XQuery source level and let *Saxon-SA 8.9* [Kay, 2009] process the resulting user-defined recursive queries (cf. Figures 8.3 and 8.5). All experiments were conducted on a Linux-based host (64 bit), with two 3.2 GHz Intel Xeon® CPUs, 8 GB of primary and 280 GB SCSI disk-based secondary memory.

In accordance with the micro-benchmarking methodology developed in Part I, we identify which query parameters might influence the performance of the IFP computation. There are three such parameters: (*i*) the complexity of the recursion body, (*ii*) the size of the input fed to the recursion body during the queries computation, and (*iii*) the depth of the recursion. Our goal is to measure the practical gains of the proposed optimization on real-life examples, rather than a thorough investigation of the precise impact of these parameters. Thus, for our experiment, we chose four queries on different XML data sets that are both natural (borrowed from related literature) and that cover different values of these parameters. We leave a more thorough investigation in the style of the MemBeR micro-benchmarking for future work.

Table 8.2 summarizes our measurements of query evaluation time, total size of the input fed to the recursion body during the recursive computation, and recursion depth, for the four queries. We varied the data instance sizes to test for scalability. Note that varying the data instance size we influence both the recursion body input size (ranging from 12K to 87M nodes for *Naïve* and from 3K to 9M nodes for *Delta*) and the recursion depth (ranging from 5 to 33). Below we describe each query and its performance.

```
let $lengths :=
    for $speech in doc("r_and_j.xml")//SPEECH
    let $rec :=
        with $x seeded by (: pair of speeches :)
                ($speech/preceding-sibling::SPEECH[1], $speech)
        recurse $x/following-sibling::SPEECH[1]
                [SPEAKER = preceding-sibling::SPEECH[2]/SPEAKER]
    return count($rec)
return max($lengths)
```

Figure 8.13: Romeo and Juliet dialogs query.

XMark Bidder Network. The first query computes a bidder network—recursively connecting the sellers and bidders of auctions—over XMark [Schmidt *et al.*, 2002] XML data (see Figure 8.12). We vary the data size from small (1MB, scale factor 0.01) to huge (37MB, scale factor 0.33). If *Delta* is used to compute the IFP of this network, *MonetDB/XQuery* as well as *Saxon* benefit significantly: 2.2 to 3.3 times faster and 1.2 to 2.7 times faster, respectively. Note that the number of nodes in the network (the same as the total number of nodes fed to body) grows quadratically with the input document size. Algorithm *Delta* feeds significantly fewer nodes to the recursion body, `bidder(·)`, at each recursion level which positively impacts the complexity of the value-based join inside the function: for the huge network, *Delta* feeds exactly those 10 million nodes into `bidder(·)` that make up the result, while *Naïve* repeatedly revisits intermediate results and processes 9 times as many nodes.

Romeo and Juliet Dialogs. Far less nodes are processed by a recursive expression that queries XML markup of Shakespeare's Romeo and Juliet[5] to determine the maximum length of any uninterrupted dialog (see Figure 8.13). Seeded with `SPEECH` element nodes, each level of the recursion expands the currently considered dialog sequences by a single `SPEECH` node given that the associated `SPEAKER`s are found to alternate. This query expresses horizontal structural recursion along the `following-sibling` axis. Although the recursion is shallow (depth 6 on average), Table 8.2 shows how both, *MonetDB/XQuery* and *Saxon*, completed evaluation up to 5 times faster because the query had been specified in a distributive fashion.

Curriculum. The following query, (Q1), was first presented in Example 8.1.1 and served as the leading example throughout the chapter. This query is borrowed directly from related work [Nentwich *et al.*, 2002] (Rule 5 in the Curriculum Case Study in Appendix B). It implements a consistency check over the curriculum data (cf. Figure 8.1) and finds courses that are among their own prerequisites.

[5] `http://www.ibiblio.org/xml/examples/shakespeare/`

```
let $hospital := doc("hospital.xml")/hospital
for $patient in $hospital/patient
where
     (with $x seeded by $patient
      recurse $x/parent/patient)/visit/treatment/test
     and
     $patient/visit/treatment[contains(medication, "headache")]
return $patient/pname
```

Figure 8.14: Hospital records query.

We generated the data instances for this query with the help of ToXgene [Barbosa *et al.*, 2002].

Much like for the bidder network query, the larger the query input (medium instance: 800 courses, large: 4,000 courses), the bigger the benefit of *Delta*, for both query engines.

Hospital records. The last query explores 50,000 hospital patient records to investigate a hereditary disease. The query, shown in Figure 8.14, is taken from [Fan *et al.*, 2006]. We generated the corresponding data instances with the help of ToXgene [Barbosa *et al.*, 2002]. In this case, the recursion follows the hierarchical structure of the XML input (from patient to parents), recursing into subtrees of a maximum depth of five. Again, *Delta* makes a notable difference even for this computationally rather "light" query.

In conclusion, this experiment renders the particular controlled form of XQuery recursion that we propose and its associated distributivity notion attractive, even for processors that do not implement a dedicated fixed point operator (like *Saxon*).

8.8 Related work

Achieving adequate support for recursion in XQuery is an important research topic. Recursion exists at different levels of the language, starting with the essential recursive XPath axes (*e.g.*, `descendant` or `ancestor`) and ending with the recursive user-defined functions. While efficient evaluation of the recursive axes is well understood by now [Al-Khalifa *et al.*, 2002, Grust *et al.*, 2003], the optimization of recursive user-deṪned functions has been found to be tractable only in the presence of restrictions: [Park *et al.*, 2002, Grinev and Lizorkin, 2004] propose exhaustive inlining of functions but require that functions are *structurally* recursive (use axes `child` and `descendant` to navigate into subtrees only) over *acyclic* schemata to guarantee that inlining terminates. Beyond inlining, the recursive user-defined functions do not come packaged with an effective optimization hook comparable to what the inflationary fixed point operator offers.

A prototypical use case for inflationary fixed point computation is transitive closure of arbitrary path expressions. This is also reflected by the advent of XPath dialects like Regular XPath [ten Cate, 2006b] and the inclusion of a dedicated `dyn:closure(·)` construct in the EXSLT function library [EXSLT, 2006]. In Section 8.7, we have seen two applications relying on transitive closure [Nentwich *et al.*, 2002, Fan *et al.*, 2006] and recent work on data integration and XML views adds to this [Fan *et al.*, 2007].

The adoption of inflationary fixed point semantics by Datalog and SQL:1999 with its `WITH RECURSIVE` clause (Section 8.2) led to an intense search for efficient evaluation techniques for inflationary fixed point operators in the domain of relational query languages. The *Naïve* algorithm implements the inflationary fixed semantics directly and it is the most widely described algorithm [Bancilhon and Ramakrishnan, 1986]. Its optimized *Delta* variant, in focus since the 1980's, has been coined *delta iteration* [Güntzer *et al.*, 1987], *semi-naïve* [Bancilhon and Ramakrishnan, 1986], or *wavefront* [Han *et al.*, 1988] strategy in earlier work. Our work rests on the adaptation of these algorithms to the XQuery Data Model and language.

While *Naïve* is applicable to all accounts of inflationary fixed points, *Delta* is mainly applicable under syntactic restrictions, such as *linear* recursion. For stratified Datalog programs [Abiteboul *et al.*, 1995], *Delta* is applicable in *all* cases, since positive Datalog maps onto the distributive operators of relational algebra (π, σ, $\bowtie$, $\cup$, $\cap$) while stratification yields partial applications of the difference operator $x \setminus R$ in which R is fixed ($f(x) = x \setminus R$ is distributive). SQL:1999, on the other hand, imposes rigid *syntactic* restrictions [Melton and Simon, 2002] on the iterative fullselect (recursion body) inside `WITH RECURSIVE` that make *Delta* applicable: grouping, ordering, usage of column functions (aggregates), and nested subqueries are ruled out, as are repeated references to the virtual table computed by the recursion. The distributivity-safe syntactic fragment introduced in Section 8.5.1 is essentially the XQuery counterpart of the linearity condition. We saw in Section 8.6, that replacing this coarse syntactic check by an elegant algebraic distributivity assessment renders a larger class of queries admissible for efficient fixed point computation.

Another well-known algorithm in the relational world, called *Smart*, is presented again only for linear recursion [Ioannidis, 1986]. *Smart* targets the inflationary fixed point computation of *relational operators* specifically and it performs better than *Delta* on shallow recursion. In the settings of XQuery, were the recursion body is any expression, *Smart* is less applicable.

8.9 Conclusions and discussions

The problem we faced in this chapter is the *lack of declarative recursive operators in XQuery that allow for (algebraic) automatic optimizations.* As a solution, we

introduced a declarative IFP operator for XQuery, borrowed from the context of relational databases. This operator covers a family of widespread use cases of recursion in XQuery, including the transitive closure of path expressions, while also being susceptible to systematic optimizations. We adopt an optimization technique widely used in relational databases and adapt it to the XQuery settings. This optimization relies on a distributivity property of XQuery expressions that can be effectively detected at the syntactic level. Furthermore, if we adopt a relational approach to XQuery evaluation, then distributivity can be detected more conveniently and effectively at the underlying algebraic level. Nevertheless, the IFP operator and the optimization technique that we propose can be easily implemented on top of any XQuery engine.

We integrated the IFP operator into the *MonetDB/XQuery* system and assessed the practical gain of our approach on real-life use cases. *MonetDB/XQuery* implements a relational approach to XQuery query evaluation and it is one of the fastest and most scalable XQuery engines today. We also experimented with *Saxon*, a popular open-source XQuery engine implementing a native approach to query evaluation. Our experiments showed significant performance gain (up to five times faster query evaluation times) on both engines. The main advantage of our approach—relying on a declarative recursive operator—is that this gain is obtained automatically, thus lifting the burden put on the user by the RUDFs.

While the empirical evidence is there, a foundational question remains: how feasible it is to do static analysis for recursive queries specified by means of the IFP operator. Specifically, are there substantial fragments of XQuery with the IFP operator for which static analysis tasks such as satisfiability are decidable? We address this question in the next chapter, Chapter 9.

Our choice of declarative recursive operator fell naturally on the IFP operator due to its success in relational databases. As we have shown, its good properties transfer to the XQuery setting. Nevertheless, there are other recursive operators, including other types of fixed points, such as the *least fixed point* operator, worth investigating. For example, a good understanding of the theoretical properties of the IFP operator for XQuery, such as its expressive power, is still missing. In Chapter 9, we study the theoretical properties of the IFP operator in the setting of the navigational core of XPath.

In spite of the fact that IFP covers a large class of recursive query needs in XQuery, some natural recursive operations cannot be expressed with it or it is very cumbersome, e.g., *recursive XML construction* (XML transformations) and *recursive aggregates*. It remains an open question what set of declarative recursive operators would be most natural to implement in the XQuery settings. This set should: (*i*) cover the most useful, commonly used, recursive query needs, and (*ii*) be easily implementable and susceptible to automatic optimizations.

Chapter 9

Core XPath with Inflationary Fixed Points

In the previous chapter, we proposed to introduce an *inflationary fixed point* (IFP) operator in XQuery. We presented an efficient processing technique for it and argued for its practical advantage, both on theoretical and experimental grounds. In this chapter, we continue with our theoretical investigation of the IFP operator in the context of Core XPath 1.0 (CXP) [Gottlob and Koch, 2002], the core navigational fragment of XPath and thus of XQuery.

We prove that the satisfiability problem of CXP extended with the IFP operator is undecidable. In fact, the fragment containing only the *self* and *descendant* axes is already undecidable. This means that a complete static analysis of recursive queries specified by means of the inflationary fixed point operator is not feasible. As a by-product of our result, we also obtain that CXP extended with IFP is strictly more expressive than CXP extended with the *transitive closure* (TC) operator, also known as Regular XPath [Marx, 2004].

This chapter is organized as follows: in Section 9.2, we define two languages, Core XPath extended with IFP (CXP+IFP) and Modal Logic extended with IFP (ML+IFP). In Section 9.3, we relate the two languages and give a truth-preserving translation from ML+IFP into CXP+IFP. In Section 9.4, we establish that the satisfiability of ML+IFP is undecidable on finite trees by presenting an encoding of successful runs of 2-register machines in ML+IFP. In Section 9.5, we discuss the implications of this result, the remaining open questions, and conclude.

9.1 Introduction

In the previous chapter, an extension of XQuery with an inflationary fixed point operator was proposed and studied. The motivation for this study stemmed from a practical need for declarative recursion operators. The existing mechanism in XQuery for expressive recursive queries (i.e., user defined recursive functions)

is procedural in nature, which makes queries both hard to write and hard to optimize. The inflationary fixed point operator provides a declarative means to specify recursive queries, and is more amenable to query optimization since it blends in naturally with algebra-based query optimization frameworks such as the one of MonetDB/XQuery [Boncz *et al.*, 2006a]. Indeed, we showed that a significant performance gain can be achieved in this way.

While the empirical evidence is there, a foundational question remains:

9.1. Question. *How feasible is it to do static analysis for recursive queries specified by means of the fixed point operator. Specifically, are there substantial fragments of XQuery with the fixed point operator for which static analysis tasks such as satisfiability are decidable?*

In this chapter, we give a strong negative answer. Our main result states that, already for the downward-looking fragment of Core XPath 1.0 with the inflationary fixed point operator (CXP+IFP), satisfiability is undecidable. The proof is based on a reduction from the undecidable halting problem for 2-register machines (cf. [Börger *et al.*, 1997]), and borrows ideas from the work of Dawar et al. [2004] on the Modal Iteration Calculus (MIC), an extension of modal logic with inflationary fixed points.

As a by-product of our investigation, we establish a relationship between CXP+IFP and MIC. While similar in spirit, it turns out that the two formalisms differ in subtle and important ways. Nevertheless, we obtain a translation from 1MIC (the fragment of MIC that does not involve simultaneous induction) to CXP+IFP node expressions.

In [Dawar *et al.*, 2004], after showing that the satisfiability problem for MIC on arbitrary structures is highly undecidable, the authors ask whether there are fragments that are still interesting to consider, and also whether the logic has any relevance for practical applications. Our results shed some light on these questions. We obtain as part of our investigation that the satisfiability problem for 1MIC is already undecidable on finite trees, and the relationship between MIC and CXP+IFP adds relevance to the study of MIC.

Another implication of our encoding of the halting problem for 2-register machines is the fact that CXP extended with IFP is strictly more expressive than CXP extended with the *transitive closure* (TC) operator, also known as Regular XPath [Marx, 2004]. The result follows from the ability of CXP+IFP to define a non-regular string language, while in Regular XPath this language cannot be defined.

9.2 Preliminaries

9.2.1 Core XPath 1.0 extended with IFP (CXP+IFP)

Core XPath 1.0 (CXP) was introduced in [Gottlob and Koch, 2002] to capture the navigational core of XPath 1.0. The definition that we use here differs slightly from the one of [Gottlob and Koch, 2002]. We consider only the downward axes *child* and *descendant* (plus the *self* axis), both in order to facilitate the comparison with MIC, and because this will already suffice for our undecidability result. We will briefly comment on the other axes later. Other differences with [Gottlob and Koch, 2002] are that we allow filters and unions to be applied to arbitrary expressions.

We consider the extension of CXP, which we call CXP+IFP, with an inflationary fixed-point operator. This inflationary fixed-point operator was first proposed in Chapter 8 in the context of XQuery, and is naturally adapted here to the setting of CXP. We first give the syntax and semantics of CXP+IFP, and then discuss the intuition behind the operator.

9.2.1. Definition (Syntax and Semantics of CXP+IFP). Let Σ be a set of labels and VAR a set of variables. The CXP+IFP expressions are defined as follows:

$$\begin{array}{rcl} axis & ::= & \mathsf{self} \mid \mathsf{child} \mid \mathsf{desc} \\ step & ::= & axis{::}l \mid axis{::}\mathsf{*} \\ \alpha & ::= & step \mid \alpha_1/\alpha_2 \mid \alpha_1 \cup \alpha_2 \mid \ \mid \alpha[\phi] \mid X \mid \mathsf{with}\ X\ \mathsf{in}\ \alpha_1\ \mathsf{recurse}\ \alpha_2 \\ \phi & ::= & \mathsf{false} \mid \langle\alpha\rangle \mid \neg\phi \mid \phi_1 \wedge \phi_2 \mid \phi_1 \vee \phi_2 \mid X\ , \end{array}$$

where $l \in \Sigma$ and $X \in VAR$. The α expressions are called *path expressions*, the ϕ expressions are called *node expressions*. The $\mathsf{with}\ \ldots\ \mathsf{in}\ \ldots\ \mathsf{recurse}\ \ldots$ operator is called the WITH operator, while X, α_1, and α_2 in the expression $\mathsf{with}\ X\ \mathsf{in}\ \alpha_1\ \mathsf{recurse}\ \alpha_2$ are called the *variable*, the *seed*, and the *body* of the recursion.

The CXP+IFP expressions are evaluated on *finite node-labeled trees*. Let $T = (N, R, L)$ be a finite node-labeled tree, where N is a finite set of nodes, $R \subset N \times N$ is the child relation in the tree, and L is a function from N to a set of labels. Let $g(\cdot)$ be an assignment function from variables to sets nodes, $g : VAR \to \wp(N)$. Then the semantics of CXP+IFP expressions are as follows:

$$\begin{array}{rcl} [\![\mathsf{self}]\!]_{T,g} & = & \{(u,u) \mid u \in N\} \\ [\![\mathsf{child}]\!]_{T,g} & = & R \\ [\![axis{::}l]\!]_{T,g} & = & \{(u,v) \in [\![axis]\!]_T \mid L(u) = l\} \\ [\![axis{::}\mathsf{*}]\!]_{T,g} & = & [\![axis]\!]_T \end{array}$$

$$
\begin{aligned}
[\![\alpha_1/\alpha_2]\!]_{T,g} &= \{(u,v) \mid \exists w.(u,w) \in [\![\alpha_1]\!]_{T,g} \wedge (w,v) \in [\![\alpha_2]\!]_{T,g}\} \\
[\![\alpha_1 \cup \alpha_2]\!]_{T,g} &= [\![\alpha_1]\!]_{T,g} \cup [\![\alpha_2]\!]_{T,g} \\
[\![\alpha[\phi]]\!]_{T,g} &= \{(u,v) \in [\![\alpha]\!]_{T,g} \mid v \in [\![\phi]\!]_{T,g}\} \\
[\![X]\!]_{T,g} &= N \times g(X), X \in VAR \\
[\![\mathsf{with}\ X\ \mathsf{in}\ \alpha_1 & \\
\quad \mathsf{recurse}\ \alpha_2]\!]_{T,g} &= \text{union of all sets } \{w\} \times g_k(X), \text{ for } w \in N, \\
& \quad \text{where } g_k \text{ is obtained in the following manner, for } i \geq 1: \\
& \quad g_1 := g[X \mapsto \{v \in N \mid (w,v) \in [\![\alpha_1]\!]_{T,g}\}], \\
& \quad g_{i+1} := g_i[X \mapsto g_i(X) \cup \{v \in N \mid (w,v) \in [\![\alpha_2]\!]_{T,g_i}\}], \\
& \quad \text{and } k \text{ is the least natural number for which } g_{k+1}{=}g_k.
\end{aligned}
$$

$$
\begin{aligned}
\langle\!|\mathsf{false}|\!\rangle_{T,g} &= \emptyset \\
\langle\!|\langle\alpha\rangle|\!\rangle_{T,g} &= \{u \in N \mid (u,v) \in [\![\alpha]\!]_{T,g}\} \\
\langle\!|\neg\phi|\!\rangle_{T,g} &= N \setminus \langle\!|\phi|\!\rangle_{T,g} \\
\langle\!|\phi_1 \wedge \phi_2|\!\rangle_{T,g} &= \langle\!|\phi_1|\!\rangle_{T,g} \cap \langle\!|\phi_2|\!\rangle_{T,g} \\
\langle\!|\phi_1 \vee \phi_2|\!\rangle_{T,g} &= \langle\!|\phi_1|\!\rangle_{T,g} \cup \langle\!|\phi_2|\!\rangle_{T,g} \\
\langle\!|X|\!\rangle_{T,g} &= g(X), X \in VAR
\end{aligned}
$$

While the semantics $[\![\alpha]\!]_{T,g}$ of a path expression α is defined as a binary relation, it is natural to think of it as a function mapping each node u to a set of nodes $\{v \mid (u,v) \in [\![\alpha]\!]_{T,g}\}$, which we denote by $Result^g_u(\alpha)$. It represents the result of evaluating α in the context node u (using the assignment g). The semantics of the variables and of the WITH operator is most naturally understood from this perspective, and can be equivalently stated as follows:

- $Result^g_u(X) = g(X)$, i.e., when X is used as a path expression, it evaluates to $g(X)$ regardless of the context node.
- $Result^g_u(\mathsf{with}\ X\ \mathsf{in}\ \alpha_1\ \mathsf{recurse}\ \alpha_2) = X_k$, where $X_1 = Result^{g[X\mapsto\emptyset]}_u(\alpha_1)$, $X_{i+1} = X_i \cup Result^{g[X\mapsto X_i]}_u(\alpha_2)$ for $i \geq 1$, and k is the smallest number such that $X_k = X_{k+1}$.

Note that, at each iteration, the context node of the evaluation of α_1 or α_2 remains u.

When a variable X is used as a node expression, it simply tests whether the current node belongs to the set assigned to X.

The example query below yields the set of nodes that can be reached from the context node by following the transitive closure of the child::a relation.

$$\mathsf{with}\ X\ \mathsf{in}\ \mathsf{child}::a\ \mathsf{recurse}\ X/\mathsf{child}::a$$

The query below yields the set of nodes that are labeled with a and are at an even distance from the context node.

$$(\mathsf{with}\ X\ \mathsf{in}\ \mathsf{self}::*\ \mathsf{recurse}\ X/\mathsf{child}::*/\mathsf{child}::*)/\mathsf{self}::a$$

It is important to note that (unlike MIC) the language provides no way to test whether a given node belongs to the result of $\mathsf{with}\ X\ \mathsf{in}\ \alpha_1\ \mathsf{recurse}\ \alpha_2$, it only allows us to *go to* a node belonging to the result set. From the point of view of XQuery and XPath, it is very natural to define the inflationary fixed point operator in this way, i.e., as an operator on path expressions. However, this has some subtle consequences, as we explain next.

The semantics of the WITH operator we give here differs slighly from the original semantics used in Chapter 8. According to the original semantics, when $Result_u^g(\mathsf{with}\ \alpha_1\ \mathsf{in}\ \alpha_2\ \mathsf{recurse}\ )$ is computed, the result of α_1 is only used as a seed of the recursion but is not itself added to the fixed point set. In other words, $Result_u^g(\mathsf{with}\ X\ \mathsf{in}\ \alpha_1\ \mathsf{recurse}\ \alpha_2)$ was defined there as X_k, where $X_0 = Result_u^{g[X\mapsto\emptyset]}(\alpha_1)$, $X_1 = Result_u^{g[X\mapsto X_0]}(\alpha_2)$, $X_{i+1} = X_i \cup Result_u^{g[X\mapsto X_i]}(\alpha_2)$ for $i \geq 1$, and k is the least number such that $X_k = X_{k+1}$. The semantics we use here is arguably mathematically cleaner and more intuitive since it is truly inflationary: all the nodes assigned to the recursion variable during fixed-point computation end up in the result.

9.2.2 Propositional Modal Logic extended with IFP (ML+IFP)

The language ML+IFP we consider is an extension of Propositional Modal Logic (ML) [Blackburn *et al.*, 2002] with a monadic IFP operator. It is also known as 1MIC, the fragment of Modal Iteration Calculus (MIC) that does not involve simultaneous induction, and it was first introduced in [Dawar *et al.*, 2004], where it was also shown that its satisfiability problem is undecidable on arbitrary structures.

9.2.2. Definition (ML+IFP). Let Σ be a set of labels and VAR a set of variables. Then the syntax of ML+IFP is defined as follows:

$$\phi \ ::= \ \bot \mid l \mid X \mid \Diamond\phi \mid \neg\phi \mid \phi_1 \wedge \phi_2 \mid \phi_1 \vee \phi_2 \mid \big(\mathsf{ifp}\ X \leftarrow \phi\big)$$

where $l \in \Sigma$, $X \in VAR$.

The semantics of ML+IFP is given in terms of Kripke models. To facilitate the comparison with CXP+IFP, we will assume that the Kripke models assign a unique label to each node, rather than a set of labels. This is not essential, since for a finite set of labels Σ this property can be expressed with a Modal Logic formula. Let $T = (N, R, L)$ be a Kripke model, where N is a set of nodes, $R \subseteq N \times N$ is a binary relation on the nodes in N, and L is a valuation function that assigns a label from Σ to to each in N. Let $g(\cdot)$ be an assignment function from variables to sets of nodes, $g : VAR \to \wp(N)$. Then the semantics of ML+IFP

formulas are as follows:

$$
\begin{array}{rcl}
[\![\bot]\!]_{T,g} & = & \emptyset \\
[\![l]\!]_{T,g} & = & \{n \in N \mid L(n) = l\} \\
[\![X]\!]_{T,g} & = & g(X) \\
[\![\Diamond\phi]\!]_{T,g} & = & \{u \mid \exists v.(u,v) \in R \wedge v \in [\![\phi]\!]_{T,g}\} \\
[\![\neg\phi]\!]_{T,g} & = & N \setminus [\![\phi]\!]_{T,g} \\
[\![\phi_1 \wedge \phi_2]\!]_{T,g} & = & [\![\phi_1]\!]_{T,g} \cap [\![\phi_2]\!]_{T,g} \\
[\![\phi_1 \vee \phi_2]\!]_{T,g} & = & [\![\phi_1]\!]_{T,g} \cup [\![\phi_2]\!]_{T,g} \\
[\![\mathsf{ifp}\ X \leftarrow \phi]\!]_{T,g} & = & g_k(X), \text{ where } g_k \text{ is obtained in the following manner:} \\
 & & g_0 := g[X \mapsto \emptyset], \\
 & & g_{i+1} := g_i[X \mapsto g_i(X) \cup [\![\phi]\!]_{T,g_i}], \text{ for } i \geq 0, \\
 & & \text{where } k \text{ is the minimum number for which } g_{k+1}{=}g_k.
\end{array}
$$

We write $T, g, u \Vdash \phi$ if $v \in [\![\phi]\!]_{T,g}$. If a formula has no free variables, we may leave out the assignment and write $T, u \Vdash \phi$ or $u \in [\![\phi]\!]_T$.

It was shown in [Dawar *et al.*, 2004] that the satisfiability problem for ML+IFP on arbitrary Kripke models is highly undecidable. As we will show below, it is undecidable on finite trees as well.

9.3 ML+IFP vs. CXP+IFP

In this section, we give a truth-preserving translation from ML+IFP to CXP+IFP. In fact, the translation yields CXP+IFP expressions that use only the *self* and *descendant* axes. It follows that this fragment of CXP+IFP already has (at least) the expressive power of ML+IFP.

One of the main differences between ML+IFP and CXP+IFP is that, in the former, fixed-point expressions are node expressions that test whether the current node belongs to the fixed point of a formula, while in the latter, fixed-point expressions are path expressions that travel to nodes belonging to the fixed point of a formula. Another difference is that, in CXP+IFP, during the entire fixed point computation, the expressions are evaluated from a fixed context node, whereas in ML+IFP, whether a node is added to the set at some stage of the fixed point computation is determined by local properties of the subtree below that node.

In our translation from ML+IFP to CXP+IFP we have to overcome these differences. The main idea for the translation of ML+IFP formulas of the form $\mathsf{ifp}\ X \leftarrow \phi$ will be that, during the fixed point computation, we treat leaf nodes in a special way, never adding them to the fixed point set but keeping track of them separately. More precisely, we first compute the set Y of all leaf nodes satisfying $\mathsf{ifp}\ X \leftarrow \phi$. Next, we let $X_0 = \emptyset$ and X_{i+1} is computed as $X_i \cup ([\![\phi]\!]_{T,g[X \mapsto X_i \cup Y]} - Y)$. Observe how the nodes in Y are added to the input and substracted again from the output. Let X_k be the fixed point of the sequence $X_0 \subseteq X_1 \subseteq \cdots$. Then we

have that $[\![\mathsf{ifp}\ X \leftarrow \phi]\!]_{T,g} = X_k \cup Y$. The advantage of this construction is that, since the leaves are never added during the fixed point computation, they can be freely used for signalling that the context node was added to the set X: if the context node is added at some stage, we add a leaf node as well, and the presence of a leaf node in the result set will be used as a sign that we test for afterwards.

Before we give the details of the construction, we first note that when computing the inflationary fixed point of an ML+IFP formula, any leaf node that is added to the fixed point set is in fact already added at the first stage of the fixed point computation. This is expressed by the following lemma.

9.3.1. Lemma. *Let u be any node in a Kripke model T, and let $\phi(X)$ be any ML+IFP formula and g an assignment. If u has no successors, then $u \in [\![\mathsf{ifp}\ X \leftarrow \phi]\!]_{T,g}$ iff $u \in [\![\phi]\!]_{T,g[X\mapsto\emptyset]}$.*

Proof. Follows from the fact that the modal formula ϕ only speaks about the submodel generated by u, i.e., the submodel consisting only of the node u itself. QED

In what follows we will use $\oslash$ as shorthand for $\mathsf{self}::*[\mathsf{false}]$, $\mathsf{desc\text{-}or\text{-}self}::*$ as shorthand for $\mathsf{desc}::* \cup \mathsf{self}::*$, and leaf as shorthand for $\neg\langle\mathsf{child}::*\rangle$. Also, for node expressions ϕ, ψ and a variable X, such that X only occurs in ϕ in the form of node tests, we will denote by $\phi^{X/\psi}$ the node expression obtained from by replacing all free occurrences of X in ϕ by the node expression ψ.

The translation $\tau(\cdot)$ from ML+IFP formulas to CXP+IFP node expressions is given by Equation (9.1).

$$
\begin{array}{rcl}
\tau(\bot) & = & \mathsf{false} \\
\tau(l) & = & \langle \mathsf{self}::l \rangle \\
\tau(\phi_1 \wedge \phi_2) & = & \tau(\phi_1) \wedge \tau(\phi_2) \\
\tau(\phi_1 \vee \phi_2) & = & \tau(\phi_1) \vee \tau(\phi_2) \\
\tau(\neg\phi) & = & \neg\tau(\phi) \\
\tau(X) & = & X \\
\tau(\Diamond\phi) & = & \langle \mathsf{child}::*[\tau(\phi)] \rangle \\
\tau\big(\mathsf{ifp}\ X \leftarrow \phi\big) & = & \langle \big(\mathsf{with}\ X\ \mathsf{in}\ \mathsf{desc\text{-}or\text{-}self}::*[\tau(\phi)^{X/\mathsf{false}} \wedge \neg\mathsf{leaf}]\ \mathsf{recurse} \\
 & & \quad \mathsf{desc\text{-}or\text{-}self}::*[\tau(\phi)^{X/(X\vee\tau(\phi)_{\mathsf{leaf}})} \wedge \neg\mathsf{leaf}] \cup \\
 & & \quad \mathsf{self}::*[X \vee \tau(\phi)_{\mathsf{leaf}}]/\mathsf{desc}::* \big)[\mathsf{leaf}] \rangle \\
 & & \text{where } \tau(\phi)_{\mathsf{leaf}} = \tau(\phi)^{X/\mathsf{false}} \wedge \mathsf{leaf}
\end{array}
\tag{9.1}
$$

9.3.2. Theorem. *Let $T = (N, R, L)$ be a node-labeled finite tree, g an assignment, and u a node in T. Then $T, g, u \Vdash \phi$ iff $T, g, u \Vdash \tau(\phi)$.*

Proof. The proof is by simultaneous induction on the form of the formula ϕ. The cases for $\bot$, l, $\psi_1 \wedge \psi_2$, $\psi_1 \vee \psi_2$, $\neg\psi$, X, and $\Diamond\psi$ are immediate. Therefore, let $\phi = \big(\mathsf{ifp}\ X \leftarrow \psi\big)$.

Since ML+IFP formulas and CXP+IFP node expressions can only see the subtree of the context node, we may assume without loss of generality that u is the root of the tree T. We write T_u instead of T, to make this explicit.

Let g_i, $0 \leq i \leq k$, be the variable assignments computed for ϕ in accordance with the semantics of the IFP operator (see Definition 9.2.2) on T_u, where $g_0 = g[X \mapsto \emptyset]$ and where k is the least natural number such that either $u \in g_k(X)$ or $g_k(X) = g_{k+1}(X)$, whichever happens first. Similarly, let g'_i, $1 \leq i \leq k$, be the first k variable assignments computed for the recursive sub-expression of $\tau(\phi)$ in accordance with the semantics of the WITH operator (see Definition 9.2.1) for u as the context.

Let $Y = [\![\tau(\phi)_{\mathsf{leaf}}]\!]_{T,g}$. It follows from the induction hypothesis, together with Lemma 9.3.1, that Y is precisely the set of all leaf nodes belonging to $g_k(X)$, and moreover, for every $1 \leq i \leq k$, Y is precisely the set of all leaf nodes belonging to $g_i(X)$.

Now, a straightforward induction shows that, for each $i \leq k$, $g_i(X)$ is exactly the disjoint union of the sets $g'_i(X)$ and Y (note that we use here the fact that $u \notin g_{k-1}(X)$).

Now, there are two possibilities: either $u \in g_k(X)$ (in which case u satisfies $\big(\mathsf{ifp}\ X \leftarrow \phi\big)$) or $u \notin g_k(X)$ (in which case u does not satisfy $\big(\mathsf{ifp}\ X \leftarrow \phi\big)$). In the first case, it is easy to see that $g'_{k+1}(X)$ contains all nodes below u, and in particular, contains a leaf node, and therefore $\tau(\big(\mathsf{ifp}\ X \leftarrow \phi\big))$ is satisfied. In the second case, $g'_{k+1}(X) = g'_k(X)$, and therefore u does not satisfy $\tau(\big(\mathsf{ifp}\ X \leftarrow \phi\big))$. This concludes the proof. QED

We can conclude that CXP+IFP node expressions have (at least) the expressive power of ML+IFP. Since the desc axis is definable from the child axis, the same holds of course for the fragment of CXP+IFP without the desc axis. What is more surprising is that the same holds for the fragment of CXP+IFP without the child axis. The next lemma shows that the use of the child axis in the above translation can be avoided (provided that we keep, of course, the desc axis). Note that the child axis was only used in the translation of formulas of the form $\Diamond\phi$.

9.3.3. Proposition. *For any node expression ϕ, $\langle \mathsf{child}::*[\phi]\rangle$ is equivalent to the following node expression (which does not use the* child *axis):*

$$\begin{array}{c}\langle\Big(\mathsf{with}\ X\ \mathsf{in}\ \mathsf{desc}::*/\mathsf{desc}::*[\mathsf{leaf}]\ \mathsf{recurse}\\ \quad \mathsf{self}::*[\langle\mathsf{desc}::*[\mathsf{leaf}\wedge\neg X\wedge\phi]\rangle]\Big)[\neg\mathsf{leaf}]\rangle\\ \vee\\ \langle\Big(\mathsf{with}\ X\ \mathsf{in}\ \mathsf{desc}::*/\mathsf{desc}::*[\neg\mathsf{leaf}]\ \mathsf{recurse}\\ \quad \mathsf{desc}::*[\neg\mathsf{leaf}\wedge\neg X\wedge\phi]/\mathsf{desc}::*\Big)[\mathsf{leaf}]\rangle\end{array}$$

Proof. Let $T = (N, R, L)$ be a finite nodel-labeled tree, $u \in N$ a node, and $g : VAR \to \wp(N)$ an assignment. We will show that the first disjunct of the node expression is true at u under the assignment g if and only if u has a child satisfying ϕ (under g) that is a leaf. Similarly, it can be shown that the second disjunct is true if and only if u has a child satisfying ϕ that is not a leaf.

Thus, let us consider the first disjunct. In the first step of the inflationary fixed point computation, all leaf nodes below u are added to the set X except those that are a child of u. Next, u itself is added to the set X just in case it has a descendant satisfying ϕ that is a leaf and that was not marked with X already. After these two steps, the fixed point is reached. It is easy to see that the set X obtained in this way contains a non-leaf node if and only if it contains the node u, which holds if and only if u has a descendant satisfying ϕ that is a leaf and that was nor marked by X in the first step of the fixed point computation. The latter holds if and only if u has a child that is a leaf and that satisfies ϕ.

In the same way, for the second disjunct of the node expression, it can be shown that the inflationary fixed point set X contains a leaf node if and only if u has a child satisfying ϕ that is not a leaf. QED

9.4 CXP+IFP and ML+IFP are undecidable on finite trees

We show that the satisfiability problem for ML+IFP on finite trees is undecidable, and therefore also (by our earlier translation), the satisfiability problem for CXP+IFP.

9.4.1. Theorem. *The satisfiability problem of ML+IFP on finite trees is undecidable.*

9.4.2. Corollary. *The satisfiability problem of CXP+IFP is undecidable, even if the* child *axis is disallowed.*

The proof, given in Section 9.4.2, is based on a reduction from the halting problem for 2-register machines (cf. [Börger *et al.*, 1997]).

9.4.1 2-Register machines

A 2-register machine is a very simple kind of deterministic automaton without input and output. It has two registers containing integer values, and instructions for incrementing and decrementing the content of the registers. These 2-register automata form one of the simplest types of machines for which the halting problem is undecidable. The formal definition is as follows:

A *2-register machine* M is a tuple $M = (Q, \delta, q_0, q_f)$, where Q is a finite set of *states*, δ is a *transition function* from Q to a set of instructions I, defined below,

and q_0, q_f are designated states in Q, called *initial and final states*, respectively [Börger *et al.*, 1997].

The set of *instructions* I consists of four kinds of instructions:

- $INC_A(q')$: increment the value stored in A and move to state q';
- $INC_B(q')$: increment the value stored in B and move to state q';
- $DEC_A(q', q'')$: if the value stored in A is bigger than 0 then decrement it with one and move to state q', otherwise move to state q'' without changing the value in A nor B; and
- $DEC_B(q', q'')$: if the value stored in B is bigger than 0 then decrement it with one and move to state q', otherwise move to state q'' without changing the value in A nor B.

A *configuration* of the machine M is a triple $S = (q, a, b)$, where q is a state in Q, and a, b are non-negative integers that correspond to the numbers stored in the registers A and B, respectively. The configuration $S_0 = (q_0, 0, 0)$ is called the initial configuration, and the configuration $S_f = (q_f, 0, 0)$ is called the final configuration.

A *successful run* of the machine is a sequence of configurations, $S_i = (q_i, a_i, b_i)$, $0 \leq i \leq n$, $n > 0$, such that:

- the sequence starts with the initial configuration, S_0, and it ends with the final configuration S_f, and
- any pair of consecutive configurations in the sequence, S_i and S_{i+1}, *satisfies* δ, i.e., the state and the register values in the successor configuration correspond to the instruction attributed to the state in the predecessor configuration by δ:
  - if $\delta(q_i) = INC_A(q'_i)$, then $S_{i+1} = (q'_i, a_i + 1, b_i)$;
  - if $\delta(q_i) = INC_B(q'_i)$, then $S_{i+1} = (q'_i, a_i, b_i + 1)$;
  - if $\delta(q_i) = DEC_A(q'_i, q''_i)$, then $S_{i+1} = (q'_i, a_i - 1, b_i)$, if $a_i > 0$, else $S_{i+1} = (q''_i, a_i, b_i)$;
  - if $\delta(q_i) = DEC_B(q'_i, q''_i)$, then $S_{i+1} = (q'_i, a_i, b_i - 1)$, if $b_i > 0$, else $S_{i+1} = (q''_i, a_i, b_i)$.

9.4.3. Theorem ([Börger *et al.*, 1997]). *The following question is known to be undecidable:* given a 2-register machine, is there a successful run of this machine?

Note that, since a 2-register machine is deterministic without input, it can have only one minimal successful run, and any other successful other run must contain the first one as a prefix. We may in fact assume without loss of generality that the machine does not pass through the final configuration $(q_0, 0, 0)$ more than once, and hence has at most one successful run. Two further assumptions we can safely make are: (*i*) the initial and final states are distinct (if $q_0 = q_f$ then the machine trivially has a successful run), and (*ii*) no two subsequent configurations on any run of the machine have the same state (this can be ensured by adding additional intermediate states if necessary).

9.4.2 The reduction

In this section, we construct a ML+IFP formula that is satisfied in the root of a finite labeled tree *if and only if* all the paths from the root to the leaves represent a successful run of a given 2-register machine. If a successful run exists, then there is a tree that satisfies this formula, and vice versa, if there is no successful run, then there is no tree that satisfies the formula.

For the remainder of this section, fix a 2-register machine $M = (Q, \delta, q_0, q_f)$. The set of labels used in the formula will be $\Sigma = Q \cup \{a, b, \$\}$, where Q is the set of states of the 2-register machine, a, b are symbols used for representing the register content in each configuration, and \$ is another symbol used for marking the end of the encoding of the successful run. It is convenient in what follows to treat these labels as mutually exclusive. In other words, when we write a symbol such as a, we actually mean $a \wedge \bigwedge_{c \in \Sigma \setminus \{a\}} \neg c$.

We model the registers A and B of a 2-register machine with paths labeled with a and b, respectively. The number of nodes in the path corresponds to the integer number stored in the respective register. Then we prove that we can express the equality of two register values in ML+IFP. This is needed in order to express that two configurations of the machine satisfy the transition function δ. Once we can express that two configurations satisfy the transition function, we construct a formula that forces the existence of a sequence of configurations that forms a successful run.

It will be convenient to consider regular expressions describing paths in the tree. By a *path* in a tree, we mean a sequence of nodes $v_1, \ldots, v_n$ $(n \geq 1)$ such that any two consecutive nodes satisfy the child relation, i.e., $(v_i, v_{i+1}) \in R$, for $1 \leq i \leq n-1$. A path that ends with a leaf node is called a *branch*. A prefix of a path $v_1, \ldots, v_n$ is any path $v_1, \ldots, v_i$ with $i \leq n$. In order to describe paths, we will use expressions built up inductively from ML+IFP formulas using the regular operations of composition, union $(+)$, and transitive closure $(\cdot^+)$ as well as reflexive transitive closure $(\cdot^*)$. We call such expressions *regular path expressions*. For example, $a(\neg a)$ is a regular path expression that is satisfied by paths of length two whose first node satisfies a and whose second node does not, and $(\top\top)^*$ is a regular path expression that is satisfied by paths of even length.

We want to build a formula that describes a successful run of a given 2-register machine. For this purpose, we encode a configuration of this machine, $S = (q, n, m)$, $n, m \geq 0$, with a path that satisfies $qa^{n+1}b^{m+1}$, i.e, we represent the values n and m stored in the A and B registers with a sequence of $n+1$ a-labels and a sequence of $m+1$ b-labels. A sequence of configurations $S_1, \ldots, S_k$ is encoded by a path that satisfies $q_1 a^{n_1+1} b^{m_1+1} \ldots q_k a^{n_k+1} b^{m_k+1}\$$, where \$ is a special label indicating the end of the sequence. In order to describe a successful run, we first build a formula describing that a pair of configurations satisfy the transition function δ of the given machine, then we build a formula that ensures that every consecutive pair of configurations satisfies δ. In order to describe a pair of configurations that satisfy δ, we need to be able to express the equality constraints that δ enforces on the register values before and after a transition. For example, for $\delta(q) = INC_A(q')$ and two configurations that satisfy δ, $S = (q, n, m)$ and $S' = (q', n+1, m)$, $n, m > 0$, we need to express that a path satisfies $qa^n b^m q' a^{n+1} b^m$.

Below, in Lemma 9.4.5, we show a generic formula that expresses the equality of the lengths of two a-labeled sequences. But first, we prove an auxiliary lemma that helps with the notations and interpretation of the formulas.

9.4.4. Lemma. *Let α be any regular path expression. Then there are* ML+IFP *formulas $\phi^{\exists}_{\alpha}$ and $\phi^{\forall}_{\alpha}$ such that for any finite labeled tree T and node v,*

1. *$T, v \vdash \phi^{\exists}_{\alpha}$ iff there is a path starting with v satisfying α, and*
2. *$T, v \vdash \phi^{\forall}_{\alpha}$ iff every branch starting with v has a prefix-path that satisfies α.*

Proof. We know that the statement holds for the modal mu-calculus (it follows from the fact that the modal mu-calculus is the bisimulation invariant fragment of MSO on finite trees [Janin and Walukiewicz, 1996]). To see that it holds also for ML+IFP we proceed as follows:

Let an expression α be given. First we replace each ML+IFP formula ψ ocurring in α by a new corresponding fresh propositional variable p_ψ. Let the resulting expression be α'. Then, clearly, α' is an expression built up from formulas of the modal mu-calculus using concatenation, union, and star. Hence, there are formulas $\phi^{\exists}_{\alpha'}$ and $\phi^{\forall}_{\alpha'}$ of the modal mu-calculus satisfying the required conditions with respect to α'. Now, replace each p_ψ back by the original formula ψ (making sure that no free occurrences of variables in ψ accidentally get bound by a fixed point operator during the substitution—this can be ensured by renaming bound variables appropriately). It follows that the resulting formulas satisfy the required conditions with respect to α. QED

In the following, we rely heavily on this lemma.

9.4.5. Lemma. *There is a formula, $\phi^{\forall}_{a^n b a^n c}$, such that for any finite labeled tree T and a node v in this tree, $T, v \Vdash \phi^{\forall}_{a^n b a^n c}$ iff there is a $k > 0$ such that every branch starting with v has a prefix-path that satisfies $a^k b a^k c$.*

Proof. In [Dawar *et al.*, 2004], a formula was constructed that, on finite strings (i.e., finite trees in which each node has at most one child), defines the language $a^n b^{\geq n}$. Our construction below differs from the one in [Dawar *et al.*, 2004] in that our formula expresses exact equality, and, more importantly, in the fact that it works on arbitrary finite trees, which makes it a non-trivial generalization.

We define $\phi^{\forall}_{a^n b a^n c}$ as in Equation (9.2).

$$\phi^{\forall}_{a^n b a^n c} \quad := \quad \phi^{\forall}_{a^n b a^{\geq n} c} \wedge \neg \phi^{\exists}_{a^n b a^{>n} c + a a^+ c} \quad , \tag{9.2}$$

where

$$\begin{aligned} \phi^{\forall}_{a^n b a^{\geq n} c} &:= \phi^{\forall}_{a^* b a^* c} \wedge \left(\mathsf{ifp}\ X \leftarrow \phi^{\forall}_{a(a \wedge X)^* b (a \wedge \neg X) a^* c} \vee \phi^{\exists}_{a(a \wedge X)^* c} \right) \\ \phi^{\exists}_{a^n b a^{>n} c + a a^+ c} &:= \left(\mathsf{ifp}\ X \leftarrow \phi^{\exists}_{a(a \wedge X)^* b (a \wedge \neg X) a^* a c} \vee \phi^{\exists}_{a(a \wedge X)^* a c} \right) \end{aligned}$$

The idea behind the two conjuncts of the formula is that $\phi^{\forall}_{a^n b a^{\geq n} c}$ expresses something slightly too weak, since it only enforces the second sequence of a-nodes on each path to be *at least* as long as the first sequence. The second conjunct $\neg \phi^{\exists}_{a^n b a^{>n} c + a a^+ c}$ corrects for this by enforcing that there is no path on which the second sequence of a-nodes is strictly longer than the first sequence. For technical reasons, the formula $\phi^{\exists}_{a^n b a^{>n} c + a a^+ c}$ in question expresses something weaker than the existence of a path satisfying $a^n b a^{>n} c$: it also accepts nodes where a path starts satisfying $a a^+ c$. However, this clearly does not affect the correctness of the overall formula $\phi^{\forall}_{a^n b a^n c}$.

Below, we prove that the formulas $\phi^{\forall}_{a^n b a^{\geq n} c}$ and $\phi^{\exists}_{a^n b a^{>n} c + a a^+ c}$ do indeed have the intended meaning, which is captured by the following claims:

1. $T, v \Vdash \phi^{\forall}_{a^n b a^{\geq n} c}$ iff $\exists k$, such that every branch starting with v has a prefix-path that satisfies $a^{\leq k} b a^{\geq k} c$.

2. $T, v \Vdash \phi^{\exists}_{a^n b a^{>n} c + a a^+ c}$ iff there exists a path starting with v that satisfies $a^n b a^m c$, for some $m > n > 0$, or that satisfies $a^n c$, for some $n \geq 2$.

It is clear from the above discussion that the lemma follows directly from (1) and (2). Below we give the proof for (1). The proof for (2) is similar. Note that we rely on Lemma 9.4.4 for the existence and the semantics of the formulas $\phi^{\exists}_{a(a \wedge X)^* b (a \wedge \neg X)}$, $\phi^{\forall}_{a(a \wedge X)^* b (a \wedge \neg X)}$, $\phi^{\exists}_{a(a \wedge X)^* a c}$, and $\phi^{\forall}_{a^* b a^* c}$.

Let $g(\cdot)$ be a variable assignment and let $g_i(\cdot)$, $0 \leq i$, be the variable assignments obtained for $\left(\mathsf{ifp}\ X \leftarrow \phi^{\forall}_{a(a \wedge X)^* b (a \wedge \neg X)} \vee \phi^{\exists}_{a(a \wedge X)^* c} \right)$ in accordance with the semantics of the IFP operator (Definition 9.2.2), where $g_0 = g[X \mapsto \emptyset]$. First, we show, by induction on i, $0 < i$, the following:

i. A node in $[\![\phi^{\exists}_{a^+ c}]\!]_{T,g}$ is added to $g_i(X)$ at the recursion step i iff there is a path from that node that satisfies $a^i c$ and there is no other path from that node that satisfies $a^{<i} c$ (i.e., that satisfies a^j for some $j < i$).

Here, by "is added to $g_i(X)$ at the recursion step i", we mean that the node belongs to $g_i(X)$ and not to $g_{i-1}(X)$. Then, using this equivalence, again by induction on i, $0 < i$, we show the following:

ii. A node in $[\![\phi^{\forall}_{a^*ba^*c}]\!]_{T,g}$ is added to $g_i(X)$ at the recursion step i iff every branch starting with that node has a prefix-path that satisfies $a^{\leq i}ba^*a^ic$ and i is the least number with this property.

Suppose $u \in [\![\phi^{\exists}_{a^+c}]\!]_{T,g}$. It is easy to see that u is added to $g_1(X)$ iff $u \in [\![\phi^{\exists}_{a(a\wedge X)^*c}]\!]_{T,g[X\mapsto\emptyset]}$ iff there is a path starting with u that satisfies ac. Next, suppose that (i) holds for some $i \geq 1$. We show that (i) holds for $i+1$.

Suppose $u \in [\![\phi^{\exists}_{a^+c}]\!]_{T,g}$ and u is added to $g_{i+1}(X)$. Then $u \in [\![\phi^{\exists}_{a(a\wedge X)^*c}]\!]_{T,g_i}$. From this it follows that u is labeled with a and there is a successor w labeled with c or $w \in g_i(X)$. Note that $w \in [\![\phi^{\exists}_{a^*c}]\!]_{T,g}$. In the first case, by induction hypothesis, u was added already to $g_1(X)$, which contradicts our assumption. In the second case, $w \in g_i(X)$ and w was added to $g_i(X)$ at the recursion step i, otherwise, by the same argument, u would be already in $g_i(X)$. By induction hypothesis, there is a path starting with w that satisfies a^ic and thus, there is a path starting with u that satisfies $a^{i+1}c$.

Conversely, suppose that there is a path from u that satisfies $a^{i+1}c$ and there is no other path from that node that satisfies $a^{<i+1}c$. Let w be the successor of u on that path. Then there is a path from w that satisfies a^ic and there is no other path from w that satisfies $a^{<i}c$. By induction hypothesis, w was added to $g_i(X)$ and thus, u must be added to $g_{i+1}(X)$.

This concludes the proof of (i). We now proceed with the proof of (ii).

Suppose $u \in [\![\phi^{\forall}_{a^*ba^*c}]\!]_{T,g}$. Again it is easy to see that u is added to $g_1(X)$ iff $u \in [\![\phi^{\forall}_{a(a\wedge X)^*b(a\wedge\neg X)a^*c}]\!]_{T,g[X\mapsto\emptyset]}$ iff every branch starting with u has a prefix-path that satisfies $abaa^*c$. Further, suppose that (ii) holds for i, $0 < i$. We show that (ii) holds for $i+1$.

Suppose $u \in [\![\phi^{\forall}_{a^*ba^*c}]\!]_{T,g}$ and u is added to $g_{i+1}(X)$. Then we know that $u \in [\![\phi^{\forall}_{a(a\wedge X)^*b(a\wedge\neg X)a^*c}]\!]_{T,g_i}$ and thus, every successor w of u (there is at least one successor) is in $g_i(X)$. Suppose that w was added to $g_j(X)$ at iteration step $j \leq i$. By induction hypothesis, every branch from w has a prefix-path that satisfies $a^{\leq j}ba^*a^jc$, thus it satisfies $a^{\leq i}ba^*c$. By statement (i) proven above, every branch from u has prefix-path that satisfies $a^*ba^*a^{i+1}c$. From the last two statements it follows that every branch from u has a prefix-path that satisfies $a^{\leq i+1}ba^*a^{i+1}c$. Note that $i+1$ is the least number with this property, otherwise u would have been added at an earlier iteration step.

For the other direction, suppose that every branch from u has a prefix-path that satisfies $a^{\leq i+1}ba^*a^{i+1}c$ and $i+1$ is the least number with this property. From this, it follows that every branch from a successor w of u, has a prefix-path that satisfies $a^{\leq i}ba^*a^{i+1}c$, and hence $a^{\leq i}ba^*a^ic$. Let j be the least number with this property for w. Then, by induction hypothesis, it follows that w was added to

$g_j(X)$ at iteration step j. Let j_0 be the least number such that $g_{j_0}(X)$ contains all successors of u. Note that j_0 equals i, otherwise every branch from u has a prefix-path that satisfies $a^{\leq i}ba^*a^ic$, which contradicts our initial assumption. From this, it follows that $u \in [\![\phi^{\forall}_{a(a\wedge X)^{\leq i}ba^*a^ic}]\!]_{T,g_i}$ and i is the least number with this property. From the fact that every branch from u has a prefix-path that satisfies $a^{\leq i+1}ba^*a^{i+1}c$ and conform the statement (i) proven above, it follows that $u \in [\![\phi^{\forall}_{a^*b(a\wedge\neg X)a^*(a\wedge X)^ic}]\!]_{T,g_i}$. From the last two statements it follows that u is added to $g_{i+1}(X)$ at iteration step $i+1$.

Now, based on (ii) we can prove the statement of (1): $T, v \Vdash \phi^{\forall}_{a^nba^{\geq n}c}$ iff $v \in g_k(X) \cap [\![\phi^{\forall}_{a^*ba^*c}]\!]_{T,g}$, where k is the recursive step at which the computation of the IFP operator ends $\iff$ $\exists n_0, 0 < n_0 \leq k$, such that v was added to $g_{n_0}(X)$ at the iteration n_0 and $v \in [\![\phi^{\forall}_{a^*ba^*c}]\!]_{T,g}$ $\iff$ $\exists n_0, 0 < n_0 \leq k$, every branch starting with that node has a prefix-path that satisfies a^nba^mc, for some $0 < n \leq n_0 \leq m$.

The proof for (2) is similar to the proof for (i). QED

Lemma 9.4.5 can be extended to other formulas of the form $\phi^{\forall}_{\alpha a^n\beta a^n\gamma}$, where α, β, and γ are a certain kind of regular expressions denoting sequences of labels. We do not attempt a general result along these lines, but we consider instances of this general pattern for which it is clear that the proof of Lemma 9.4.5 works. In particular, in order for the proof of Lemma 9.4.5 to work, it is important that β and γ are incompatible, in the sense that they cannot be satisfied by the same path. Below, we give two examples of such formulas and state their semantics. We use these examples further on for our reduction.

Intuitively, the first example is a formula that describes a transition of the 2-register machine, in which register A is incremented. The second example describes a transition in which register A is decremented. The variable Y in these formulas is intended to represent that the remaining part of the run is already known to be correct (and will be bound by an IFP-operator).

Let $q, q', q'' \in Q$ such that $q \neq q'$ and $q \neq q''$, and let $Q'_Y := q' \wedge Y$, $Q''_Y := q'' \wedge Y$, and $E_Y := Y \vee \$$, where Y is a variable in VAR.

9.4.6. Example. Consider $\phi^{\forall}_{qa^nb^*Q'_Ya^{n+1}b^*E_Y}$ defined by Equation (9.3). Similarly to Lemma 9.4.5, we can prove that $\phi^{\forall}_{qa^nb^*Q'_Ya^{n+1}b^*E_Y}$ is valid in a node v of a finite labeled tree T iff every branch starting with v has a prefix-path that satisfies $qa^nb^*Q'_Ya^{n+1}b^*E_Y$, for some $n > 0$.

$$
\begin{array}{rcl}
\phi^{\forall}_{qa^nb^*Q'_Ya^{n+1}b^*E_Y} & := & \phi^{\forall}_{qa^nb^*Q'_Ya^{\geq n+1}b^*E_Y} \wedge \\
 & & \neg\phi^{\exists}_{qa^nb^*Q'_Ya^{>n+1}b^*E_Y + a^*a^3b^*E_Y} \\
\phi^{\forall}_{qa^nb^*Q'_Ya^{\geq n+1}b^*E_Y} & := & \phi^{\forall}_{qa^*b^*Q'_Ya^*b^*E_Y} \wedge \\
 & & \Box\big(\mathsf{ifp}\ X \leftarrow \phi^{\forall}_{a(a\wedge X)^*b^*Q'_Y(a\wedge\neg X)a^*ab^*E_Y} \vee \\
 & & \phi^{\exists}_{a(a\wedge X)^*(a\wedge\neg X)b^*E_Y}\big) \\
\phi^{\exists}_{qa^nb^*Q'_Ya^{>n+1}b^*E_Y + a^*a^3b^*E_Y} & := & q \wedge \Diamond\big(\mathsf{ifp}\ X \leftarrow \phi^{\exists}_{a(a\wedge X)^*b^*Q'_Y(a\wedge\neg X)a^*a^2b^*E_Y} \vee \\
 & & \phi^{\exists}_{a(a\wedge X)^*(a\wedge\neg X)^2b^*E_Y}\big)
\end{array}
\tag{9.3}
$$

9.4.7. Example. For convenience, consider the following alternative notations for the until formulas $\psi_1\ \mathsf{EU}\ \psi_2 := \phi^{\exists}_{\psi_1^*\psi_2}$ and $\psi_1\ \mathsf{AU}\ \psi_2 := \phi^{\forall}_{\psi_1^*\psi_2}$. Consider $\phi^{\forall}_{qa^*b^{n+1}Q'_Ya^*b^nE_Y}$ defined by Equation (9.4) and let $g(\cdot)$ be a variable assignment that covers Y. Similarly to Lemma 9.4.5, we can prove that $\phi^{\forall}_{qa^*b^{n+1}Q'_Ya^*b^nE_Y}$ is valid in a node v of a finite labeled tree T and given $g(\cdot)$ $\iff$ every branch starting with v has a prefix-path that satisfies $qa^*b^{n+1}Q'_Ya^*b^nE_Y$, for some $n > 0$.

$$
\begin{array}{rcl}
\phi^{\forall}_{qa^*b^{n+1}Q'_Ya^*b^nE_Y} & := & \phi^{\forall}_{qa^*b^{n+1}Q'_Ya^*b^{\geq n}E_Y} \wedge \\
 & & \neg\phi^{\exists}_{qa^*b^{n+1}Q'_Ya^*b^{>n}E_Y + b^*b^2E_Y} \\
\phi^{\forall}_{qa^*b^{n+1}Q'_Ya^*b^{\geq n}E_Y} & := & \phi^{\forall}_{qa^*b^*Q'_Ya^*b^*E_Y} \wedge \big(a\ \mathsf{AU}\ (b\wedge \\
 & & \big(\mathsf{ifp}\ X \leftarrow \phi^{\forall}_{b(b\wedge X)^*Q'_Ya^*(b\wedge\neg X)b^*E_Y} \vee \phi^{\exists}_{b(b\wedge X)^*E_Y}\big)\big)\big) \\
\phi^{\exists}_{qa^*b^{n+1}Q'_Ya^*b^{>n}E_Y + b^*b^2E_Y} & := & q \wedge \big(a\ \mathsf{EU}\ (b\wedge \\
 & & \big(\mathsf{ifp}\ X \leftarrow \phi^{\exists}_{b(b\wedge X)^*Q'_Ya^*(b\wedge\neg X)b^*bE_Y} \vee \\
 & & \phi^{\exists}_{b(b\wedge X)^*(b\wedge\neg X)E_Y}\big)\big)\big)
\end{array}
\tag{9.4}
$$

In a similar fashion, we construct the following formulas:

$$
\begin{array}{llll}
\phi^{\forall}_{qa^nb^*Q'_Ya^nb^*E_Y} & \phi^{\forall}_{qa^nb^*Q'_Ya^{n+1}b^*E_Y} & \phi^{\forall}_{qa^{n+1}b^*Q'_Ya^nb^*E_Y} & \phi^{\forall}_{qa^nbQ''_Ya^nbE_Y} \\
\phi^{\forall}_{qa^*b^mQ'_Ya^*b^mE_Y} & \phi^{\forall}_{qa^*b^mQ'_Ya^*b^{m+1}E_Y} & \phi^{\forall}_{qa^*b^{m+1}Q'_Ya^*b^mE_Y} & \phi^{\forall}_{qab^nQ''_Yab^nE_Y}
\end{array}
$$

An equivalent of Lemma 9.4.5 can be proven for each of these formulas. Finally, we are ready to define the transition function of a given 2-register machine.

$$
\begin{array}{rcl}
\phi^{\forall}_{qa^nb^mQ'_Ya^{n+1}b^mE_Y} & := & \phi^{\forall}_{qa^nb^*Q'_Ya^{n+1}b^*E_Y} \wedge \phi^{\forall}_{qa^*b^mQ'_Ya^*b^mE_Y} \\
\phi^{\forall}_{qa^nb^mQ'_Ya^nb^{m+1}E_Y} & := & \phi^{\forall}_{qa^nb^*Q'_Ya^nb^*E_Y} \wedge \phi^{\forall}_{qa^*b^mQ'_Ya^*b^{m+1}E_Y} \\
\phi^{\forall}_{qa^{n+1}b^mQ'_Ya^nb^mE_Y} & := & \phi^{\forall}_{qa^{n+1}b^*Q'_Ya^nb^*E_Y} \wedge \phi^{\forall}_{qa^*b^mQ'_Ya^*b^mE_Y} \\
\phi^{\forall}_{qa^nb^{m+1}Q'_Ya^nb^mE_Y} & := & \phi^{\forall}_{qa^nb^*Q'_Ya^nb^*E_Y} \wedge \phi^{\forall}_{qa^*b^{m+1}Q'_Ya^*b^mE_Y}
\end{array}
\tag{9.5}
$$

$$
\begin{array}{rcl}
Tr_q(Y) & := & \begin{cases}
\phi^{\forall}_{qa^nb^mQ'_Ya^{n+1}b^mE_Y} & \text{if } \delta(q) = INC_A(q'), \\
\phi^{\forall}_{qa^nb^mQ'_Ya^nb^{m+1}E_Y} & \text{if } \delta(q) = INC_B(q'), \\
\phi^{\forall}_{qa^{n+1}b^mQ'_Ya^nb^mE_Y} \vee \phi^{\forall}_{qab^nQ''_Yab^nE_Y} & \text{if } \delta(q) = DEC_A(q', q''), \\
\phi^{\forall}_{qa^nb^{m+1}Q'_Ya^nb^mE_Y} \vee \phi^{\forall}_{qa^nbQ''_Ya^nbE_Y} & \text{if } \delta(q) = DEC_B(q', q'')
\end{cases} \\
Tr(Y) & := & \bigvee_{q\in Q} Tr_q(Y)
\end{array}
\tag{9.6}
$$

Recall from Section 9.4.1 that we assume without loss of generality that the 2-register machine M is such that no two subsequent configurations on a run have the same state and therefore q' and q'' are always distinct from q in the formulas in (9.6). Thus, generalizing from the above examples and the proof of Lemma 9.4.5, we have the following.

9.4.8. Lemma. *Let $T = (N, R, L)$ be a finite labeled tree and $g(\cdot)$ be a variable assignment that covers the free variable Y. Then $T, v, g \Vdash Tr(Y)$ iff every branch starting with v has a prefix-path that satisfies $qa^nb^mq'a^{n'}b^{m'}E_Y$, for some pair of triples, $S = (q, n, m)$ and $S' = (q', n', m')$, that satisfies δ $(n, n', m, m' > 0)$.*

Having the formula that describes a transition, we can build the formula ϕ_{run}, below, that describes a successful run of a given 2-register machine; ϕ_{run} enforces that every branch starting from a node in the tree represents a successful run of the given machine.

$$
\begin{array}{rcl}
Q_s & := & \phi^{\forall}_{q_sab(\bigvee_{q\in Q} q)} \\
Q_f & := & \phi^{\forall}_{q_fab\$} \\
\phi_{run} & := & Q_s \wedge \big(\mathsf{ifp}\ Y \leftarrow Tr(Y) \vee Q_f\big)
\end{array}
\tag{9.7}
$$

9.4.9. Theorem. *The formula ϕ_{run} is satisfiable iff the 2-register machine M has a successful run.*

Proof.($\Longleftarrow$) Suppose that the sequence of configurations, $S_1, \ldots, S_n$, $n > 0$, is a successful run of M, with $S_i = (q_i, k_i, \ell_i)$. In particular, $S_1 = (q_0, 0, 0)$ is the initial configuration and $S_n = (q_f, 0, 0)$ is the final configuration. Also, as explained in Section 9.4.1, we may assume that $n > 1$, and that $q_i \neq q_{i+1}$ for $1 \leq i < n$. Let T be the tree consisting of a single branch, such that the sequence of labels of the nodes on the branch forms the string $q_1a^{k_1+1}b^{\ell_1+1} \cdots q_na^{k_n+1}b^{k_n+1}\$$. Let u_i (for $1 \leq i \leq n$) be the i-th node on the branch whose label belongs to Q. It is clear that the root of the tree, u_1, satisfies Q_s, and that u_n satisfies Q_f. Furthermore, for each $i \leq n$, $T, u_i \Vdash \big(\mathsf{ifp}\ Y \leftarrow Tr(Y) \vee Q_f\big)$ as can be shown by a straightforward induction on $n - i$. It follows that $T, u_1 \Vdash \phi_{run}$.

($\Longrightarrow$) Suppose $T, v \Vdash \phi_{run}$. Let $g(\cdot)$ be a variable assignment. Since $T, v \Vdash \big(\mathsf{ifp}\ Y \leftarrow Tr(Y) \vee Q_f\big)$ we have that $v \in g_k(Y)$, where $g_k(\cdot)$ is the last variable

assignment obtained conform the definition of the IFP operator (Definition 9.2.2) for $\big(\mathsf{ifp}\ Y \leftarrow Tr(Y) \vee Q_f\big)$. One can show by a straightforward induction on i, $1 \leq i \leq k$, that for every branch starting with a node $u \in g_i(Y)$, the sequence of labels of the nodes on this branch, up to the first node satisfying \$, forms an encoding of a run of the 2-register machine, starting in some (not necessarily initial) configuration and ending in the final configuration. In particular, since $v \in g_k(Y)$, and also $T, v \Vdash Q_s$, we then have that for every branch starting at v, the sequence of labels of the nodes on this branch, up to the first node saytisfying \$, forms an encoding of a successful run of the 2-register machine (starting in the initial configuration). QED

We have shown that the undecidable halting problem for 2-register machines reduces to the satisfiability problem for ML+IFP on finite trees. Theorem 9.4.1 now follows.

9.5 Discussions and conclusions

We proved that the fragment of CXP+IFP with only *self* and *descendant* axes is undecidable. This implies the undecidability of CXP+IFP with all the axes. Moreover, since the undecidability proof for ML+IFP, as well as the translation from ML+IFP into CXP+IFP, works on strings too, no matter what axis one takes (along with the *self* axis) the fragment of CXP+IFP with only that axis is undecidable. Recall that the transitive axes (e.g., *descendant*, *ancestor*, *following-sibling*, *preceding-sibling*) are easily defined from the corresponding non-transitive axes using the IFP operator.

This result means that a complete static analysis of recursive queries specified by means of the IFP operator is not feasible. In other words, we cannot do better than implementing sound-but-not-complete query optimizations, such as the distributivity-based optimization presented in Chapter 8.

Another recursion operator that has been studied extensively in the context of CXP, is the *transitive closure* (TC) of path expressions and the language is known as Regular XPath [Marx, 2004]. Note that we can express the transitive closure of a path expression α by using the IFP operator as follows:

$$\alpha^+ = \mathsf{with}\ X\ \mathsf{in}\ \alpha\ \mathsf{recurse}\ X/\alpha$$

Regular XPath falls within monadic second-order logic (MSO) [ten Cate, 2006a], while by Lemma 9.4.5, CXP+IFP can define (among all finite strings) the strings that satisfy $a^n b^n c$, $n > 0$, which is not a regular string language and thus not definable in MSO [Thomas, 1997]. From this it follows that CXP+IFP is strictly more expressive than Regular XPath.

Note that the definition of the TC operator via IFP does not use negation on the recursion variable. Thus the TC operator can be expressed also via a *least*

fixed point (LFP) operator, which is a non-inflationary fixed point operator that does not allow the recursion variable to occur under an odd number of negations. If we consider CXP extended with LFP, then this language still falls within MSO and is at least as expressive as Regular XPath but strictly less expressive than CXP+IFP on finite trees.

In conclusion, when choosing a recursion operator to extend CXP, one should keep in mind that the inflationary fixed point operator is the most expressive and expensive operator (with undecidable static analysis problems) of the three recursion operators discussed above.

9.5.1 Remaining questions

One natural follow-up question is whether CXP+IFP node expressions are strictly more expressive than ML+IFP formulas.

Other natural follow-up questions concern fragments of CXP+IFP. Recall that in CXP+IFP, the variables can be used both as atomic path expressions and as atomic node expressions. The former is the most natural, but the translation we gave from ML+IFP to CXP+IFP crucially uses the latter. Our conjecture is that the fragment of CXP+IFP in which variables are only allowed as atomic path expressions is also undecidable.

It is also natural to consider CXP+IFP expressions where the fixed point variables occur only under an even number of negations, so that the WITH-operator computes the least fixed point of a monotone operation. Note that this fragment is decidable, since it is contained in monadic second-order logic. Thus, further questions like the complexity of the static analysis problems and the expressive power of this language are open to investigation.

In Part II of this thesis, we showed how to optimize recursion in XQuery by extending the language with an Inflationary Fixed Point operator. We implemented the operator, and the optimization technique, in MonetDB/XQuery, and we provided experimental evidence for its advantages in practice. We also investigated the theoretical properties of the operator in the context of Core XPath.

Chapter 10

Conclusions

In this chapter, we revisit the research questions that we pursed in this thesis and summarize our answers. We then wrap-up with a discussion of future directions.

10.1 Answering the research questions

In this thesis, we pursued two main research themes: *How to evaluate the performance of XML query processing?* and *How to optimize recursion in XQuery?* These general questions boiled down to more concrete questions that we addressed in the different chapters of the thesis. In this section, we summarize our answers to each of these questions.

Developing benchmarking methodology

While pursuing the first general research question, we focused our investigation on the XQuery language and started with an analysis of existing XQuery benchmarks: XMach-1, XMark, X007, MBench, and XBench. The main conclusion of this analysis is that that the benchmarks are very useful for exploratory performance studies, but not adequate for rigorous performance evaluations of XML query processors. The three detailed questions that we addressed, Question 3.1, 3.2, and 3.3, and their answers follow.

Question 3.1: What do the benchmarks measure?

We analyzed and compared the workloads (i.e., the data and query sets) and measures of the five benchmarks. The benchmarks differ in their target and performance measure. Our comparison showed 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. 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, where the benchmark measure is the query processing time. X007, XMark, and XBench are application benchmarks that test the performance of an XML query processor on a relatively small set of complex queries. The latter benchmarks differ from each other in the document scenario they test: X007, XMark, and XBench TC/SD and DC/SD test a single-document scenario, while XBench TC/MD and DC/MD test a multi-document scenario.

All benchmark queries have a common characteristic: they are designed to test important language features. However, we observed that a single query usually contains more than one language feature and an engine's performance on that query cannot be attributed to only one of them. From this, we concluded that the benchmark queries have an exploratory nature rather than precise nature.

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 processing scenarios and parameters. Several advanced XML/XQuery features, such as typed data, namespaces, recursion, etc., are poorly covered. Also, 90% of all benchmark queries can already be expressed in XPath 1.0 or 2.0, provided that we consider only the element retrieval functionality and ignore the XML construction functionality of XQuery (the other 10% of the queries test two XQuery constructs: sorting and recursive user-defined functions).

Question 3.2: How are the benchmarks used?

We conducted a survey of scientific articles that contain experimental studies of XML processing and were reported in the 2004 and 2005 proceedings of the ICDE, SIGMOD and VLDB conferences, 41 papers in total. The survey showed 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, we concluded that with the exception of XMark and XBench, the benchmarks are *not* used. A 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.

Question 3.3: What can we learn from using the benchmarks?

To answer Question 3.3, we ran the benchmarks on four XQuery engines: Galax, SaxonB, Qizx/Open, and MonetDB/XQuery and compared their performance. We observed that: (*i*) the engines produce errors and suffer from crashes (even on workloads of small size); (*ii*) the relative performance of the engines varies on different benchmarks; and (*iii*) the engines' performance differs per benchmark

even for queries intended to test the same language feature. We concluded that the tested engines (or at least the versions we tested) are still immature and that no engine can be crowned as a winner. The results further indicate that implicit benchmark parameters have a big impact on the performance. The benchmarks, even applied together, cannot be used to obtain a comprehensive understanding of the performance of an engine.

Since MBench was designed as a micro-benchmark, we tested whether MBench is suitable for a rigorous analysis of a language feature it targets, namely attribute-value joins. Based on the benchmarks results obtained on Qizx/Open, we concluded that the set of four queries designed for micro-benchmarking joins is insufficient for drawing sound conclusions about its performance. Thus, even though MBench provides a good starting point for micro-benchmarking, it is incomplete, which leads to inconclusive results.

Question 4.1: How to ensure the repeatability of experimental studies of database systems? This question incorporates two sub-questions: (*i*) What is a proper methodology for designing and reporting on experimental studies that facilitates their repeatability? and (*ii*) What is a proper mechanism for evaluating and improving the repeatability of experimental studies presented in scientific research?

At the SIGMOD 2008 conference, by way of an experiment, submitted papers were subjected to a repeatability reviewing process. Although repeatability reviewing was optional, most authors participated. In order to facilitate the task of the reviewers, the authors of the papers were requested to follow a specific methodology in describing and reporting on their experiments. All in all, this mechanism provides a solution for ensuring repeatability of experimental studies of database systems, and it shows that repeatability may be achieved and measured. Based on our experience as a member of the repeatability reviewing committee, we were able to address the above questions.

Concerning Question 4.1 (i), the methodology used for describing and reporting experiments that the repeatability committee developed was enough to cover the 289 papers that were submitted for the repeatability review. Out of 64 papers that were assessed by the repeatability committee, 33 (52%) achieved the repeatability of all presented experiments and 20 (31%) achieved repeatability of some of the experiments. Considering that we strive for all experiments to be repeatable, 52% is a small number. Nevertheless, we consider these results to be a good start towards achieving the repeatability of experimental studies in the database research field.

Concerning Question 4.1 (ii), the high percentage of participation in the optional review, 66% of the total submissions to the conference (289 out of 436), hints at the perceived usefulness of a peer reviewing process. The positive feedback from the authors of the papers recorded by the survey also confirms that

such a review is considered useful for the community: 80% of the surveyed authors found the process useful, while 84% would participate in such a process in the future.

There are still some problems that need to be addressed, though, if ensuring repeatability is to become a common practice in the database community. One of the problems is the amount of effort that the reviewing process takes.

Question 5.1: Is it possible to build a generic tool for automating the following three tasks: (*i*) running a performance benchmark, (*ii*) documenting the benchmark experiment, and (*iii*) analyzing the benchmark results? What are the design choices needed to be made?

We presented XCheck, a tool for running performance benchmarks that measure execution times on sets of XML documents and sets of queries, formulated in an XML query language, such as XPath and XQuery. Given a benchmark and a set of engines, XCheck runs the benchmark on these engines, collects performance times, query results, and the testing environment configuration. XCheck was used for running all experiments in this thesis, and, as of September 2009, it has been used in at least 9 scientific papers.

During the development of XCheck we had to address several issues. First, we had to decide how XCheck should communicate with the tested engines. The command line adapter design that XCheck implements is elegant and easily implementable—many of the XML query engines have a command line interface. Second, we had to decide what atomic metric XCheck should implement. Based on what the current XQuery benchmarks measure, namely performance times of a set of queries on a set of documents/collections, the atomic metric deployed by XCheck is the total execution time of processing one query on a document/collection. If the engines provide more detailed performance times, e.g., document processing, query compilation, etc., XCheck also collects these times. Finally, we had to decide how to store and present the performance results. XCheck uses XML to store the raw measurement data, and it uses HTML and plots to present it to the user in an easily readable format.

Question 6.1: What is a suitable methodology for precise and comprehensive performance evaluations of XML query processing techniques and systems?

As a result of investigating Questions 3.1–3.3, we identified a lack of suitable tools for precise and comprehensive performance evaluations. As a solution to this problem, we proposed MemBeR, an open-ended, community driven, repository of micro-benchmarks. We endowed the repository with micro-benchmarking design principles and methodology, with a fixed micro-benchmark structure, with suggestions for potentially interesting parameters, and tools for generating parametrized data sets. In Chapter 7, we presented a concrete micro-benchmark for evaluating value-based joins processing techniques that follows the MemBeR methodology.

With MemBeR we aim to consolidate the experience of individual researchers that spend time and effort in designing micro-benchmarks for performance evaluation of their query optimization and processing techniques. We hope MemBeR will provide the necessary performance evaluation tools and methodology and will be used in the XML data management community.

Question 7.1: How to measure the performance of value-based joins expressed in XQuery? What is a suitable measure and which parameters are important to consider?

We designed a micro-benchmark for value-based joins in XQuery, following the MemBeR methodology. The benchmark measures the impact of seven query and data parameters on the performance times of an engine. The benchmark query set is carefully designed to allow for testing the impact of every parameter value in isolation. We validated our benchmark by analyzing the performance of four XQuery engines. We obtained a comprehensive overview of the performance of each engine when it comes to evaluating joins, and we identified some shortcomings of the engines, as well as some missed opportunities for optimization. We concluded that the benchmark achieves its target and it is a useful tool for profiling the performance of XQuery engines on value-based joins with respect to the tested parameters.

Recursion in XQuery

Next, we answer the research questions pursued in the second part of the thesis, referring to recursion in XQuery.

Question 8.1: What is a suitable declarative recursive operator in XQuery that is rich enough to cover interesting cases of recursion query needs and that allows for (algebraic) automatic optimizations?

As a solution to this question we proposed an inflationary fixed point (IFP) operator for XQuery. This operator covers important use cases of recursion in XQuery, such as the transitive closure of path expressions, while also being susceptible to systematic optimizations. We also propose an optimization technique for this operator. This optimization relies on a distributivity property of XQuery expressions that can be effectively detected at the syntactic level. Furthermore, if we adopt a relational approach to XQuery evaluation, then distributivity can be detected more conveniently and effectively at the underlying algebraic level. We integrated the IFP operator into the MonetDB/XQuery system and assessed the practical gain of our approach on real-life use cases using the benchmarking tools developed in Part I.

Question 9.1: How feasible is it to do static analysis for recursive queries specified by means of the fixed point operator? Specifically, are there substantial fragments of XQuery with the fixed point operator for which static analysis tasks such as satisfiability are decidable?

We investigated the theoretical aspects of the IFP operator in the context of Core XPath (CXP) [Gottlob and Koch, 2002], the navigational core of XPath. We proved that the satisfiability problem of CXP extended with the IFP operator is undecidable. In fact, the fragment containing only the `self` and `descendant` axes is already undecidable. This means that a complete static analysis of recursive queries specified by means of the inflationary fixed point operator is not feasible. In other words, we cannot do better than implementing sound-but-not-complete query optimizations, such as the distributivity-based optimization presented in Chapter 8. As a by-product of the undecidability result, we also obtained that CXP extended with IFP is strictly more expressive than CXP extended with the *transitive closure* (TC) operator, also known as Regular XPath [Marx, 2004].

10.2 Outlook and further directions

While answering the above research questions, we have raised new questions and identified unsolved problems. In this section, we list these questions and problems for each research theme separately.

Performance evaluation of XQuery engines

We have analyzed existing benchmarks for performance evaluation of XQuery benchmarks and arrived at the conclusion that the XQuery community will benefit from new benchmarks—both application benchmarks and micro-benchmarks—that have a good coverage of XQuery features. Indeed, at the time of writing this thesis, another XQuery application benchmark has been proposed, TPox [Nicola *et al.*, 2007], while we developed and presented micro-benchmarks and related methodology. A new question arises: do these new benchmarks and benchmarking methodology fulfill the need for benchmarking XQuery processing? In either case, our analysis showed that a serious investment should be made for maintaining the benchmarks at the same pace as the development of the XQuery engines (and language) themselves, otherwise the benchmarks quickly become obsolete.

Addressing the need for precise and comprehensive benchmarks, we developed a micro-benchmarking repository and related methodology, MemBeR. We have also developed a MemBeR micro-benchmark targeting the processing of value-based joins expressed in XQuery. Nevertheless, there is still a long way before the repository will contain micro-benchmarks covering many language features of XQuery and other XML query languages. We hope that more contributions will

be made to MemBeR and that the MemBeR micro-benchmarks will be used by the community.

With respect to the value-based join micro-benchmark that we developed, we have two questions left. One is whether the two benchmark parameters that did not show impact on the performance of the tested engines have impact on other engines and what impact. Another question is whether there are other parameters that might influence the impact of performance of this language features and, if so, can the micro-benchmark be extended to include them.

Benchmarking tools are only one aspect of achieving good performance evaluation. We have addressed the problem of ensuring the repeatability of experimental studies in the database community. Though, we made a first step and showed that repeatability may be indeed achieved and measured, there are still problems that need to be addressed, if ensuring repeatability is to become a common practice. One of the problems is the amount of effort that the reviewing process takes. Tools that automate the process of conducting and reporting on experiments, such as XCheck (Chapter 5), might be useful to reduce this effort.

Another aspect of repeatability that we did not discuss in this thesis is that of proper archiving mechanisms for ensuring the accessibility of experimental data and results. Long-term preservation and even curation of experimental results is another key factor of scientific proliferation. This question is actively being addressed in other database related fields, such as Information Retrieval [Agosti *et al.*, 2007].

The realization of the need of serious empirical evaluation is gaining ground in the database community, as is witnessed by the experimental studies repeatability reviewing efforts at SIGMOD 2008 and 2009, by the establishment of "The Experiments and Analyses" track at VLDB 2008 and 2009, the organization of workshops like ExpDB 2006 and 2007, etc.. We hope that this thesis makes a contribution to this development. Nevertheless, as indicated by the above list of open questions, the biggest hurdles lie ahead.

Recursion in XQuery

We have proposed optimization techniques for recursion in XQuery by introducing a declarative recursive operator to XQuery. In spite of the fact that the inflationary fixed point operator covers a large class of recursive query needs in XQuery, some natural recursive operations cannot be expressed with it or it is cumbersome, e.g., *recursive XML construction* (XML transformations) and *recursive aggregates*. It remains an open question what set of declarative recursive operators would be most natural to implement in the XQuery settings. This set should: (*i*) cover the most useful, commonly used, recursive query needs, and (*ii*) be easily implementable and susceptible to automatic optimizations.

On a more theoretical level, we made a connection between Core XPath extended with the IFP operator (CXP+IFP) and Modal Logic extended with

the IFP operator (ML+IFP). We exploited this connection and established that CXP+IFP is highly undecidable. Several natural follow-up question have arisen.

One follow-up question is whether CXP+IFP node expressions are strictly more expressive than ML+IFP formulas.

Another natural follow-up question is whether the undecidability result can be strengthened to smaller fragments of CXP+IFP. Recall that in CXP+IFP, the variables can be used both as atomic path expressions and as atomic node expressions. The former is the most natural, but the translation we gave from ML+IFP to CXP+IFP crucially uses the latter. Our conjecture is that the fragment of CXP+IFP in which variables are only allowed as atomic path expressions is already undecidable.

It is also natural to consider CXP+IFP expressions where the fixed point variables occur only under an even number of negations, so that the WITH-operator computes the least fixed point of a monotone operation. Note that this fragment is decidable, since it is contained in monadic second-order logic. Thus, further questions like the complexity of the static analysis problems and the expressive power of this language are open to investigation.

To wrap up, we believe that optimizing recursion in XQuery by exploring declarative recursive operators is worthwhile investigating, it might lead to further significant performance improvements and interesting theoretical questions.

Appendix A

LiXQuery: a Quick Syntax Reference

[1] ⟨*Query*⟩ ::= (⟨*FunDef*⟩ “;”)*⟨*Expr*⟩
[2] ⟨*FunDef*⟩ ::= “declare” “function” ⟨*Name*⟩ “(”(⟨*Var*⟩(“,”⟨*Var*⟩)*)?“)”
::= “{”⟨*Expr*⟩“}”
[3] ⟨*Expr*⟩ ::= ⟨*Var*⟩ | ⟨*BuiltIn*⟩ | ⟨*IfExpr*⟩ | ⟨*ForExpr*⟩ | ⟨*LetExpr*⟩ | ⟨*Concat*⟩ |
::= ⟨*AndOr*⟩ | ⟨*ValCmp*⟩ | ⟨*NodeCmp*⟩ | ⟨*AddExpr*⟩ | ⟨*MultExpr*⟩ |
::= ⟨*Union*⟩ | ⟨*Step*⟩ | ⟨*Filter*⟩ | ⟨*Path*⟩ | ⟨*Literal*⟩ | ⟨*EmpSeq*⟩ |
::= ⟨*Constr*⟩ | ⟨*TypeSw*⟩ | ⟨*FunCall*⟩
[4] ⟨*Var*⟩ ::= “$”⟨*Name*⟩
[5] ⟨*BuiltIn*⟩ ::= “doc(”⟨*Expr*⟩“)” | “name(”⟨*Expr*⟩“)” | “string(”⟨*Expr*⟩“)” |
::= “xs:integer(”⟨*Expr*⟩“)” | “root(”⟨*Expr*⟩“)” |
::= “concat(”⟨*Expr*⟩, ⟨*Expr*⟩“)” | “true()” | “false()” |
::= “not(”⟨*Expr*⟩“)” | “count(”⟨*Expr*⟩“)” | “position()” | “last()”
[6] ⟨*IfExpr*⟩ ::= “if ”“(”⟨*Expr*⟩“)” “then”⟨*Expr*⟩ “else”⟨*Expr*⟩
[7] ⟨*ForExpr*⟩ ::= “for”⟨*Var*⟩(“at”⟨*Var*⟩)? “in”⟨*Expr*⟩ “return”⟨*Expr*⟩
[8] ⟨*LetExpr*⟩ ::= “let”⟨*Var*⟩“:=”⟨*Expr*⟩ “return”⟨*Expr*⟩
[9] ⟨*Concat*⟩ ::= ⟨*Expr*⟩“,”⟨*Expr*⟩
[10] ⟨*AndOr*⟩ ::= ⟨*Expr*⟩(“and” | “or”)⟨*Expr*⟩
[11] ⟨*ValCmp*⟩ ::= ⟨*Expr*⟩(“=” | “<”)⟨*Expr*⟩
[12] ⟨*NodeCmp*⟩ ::= ⟨*Expr*⟩(“is” | “<<”)⟨*Expr*⟩
[13] ⟨*AddExpr*⟩ ::= ⟨*Expr*⟩ (“+” | “-”) ⟨*Expr*⟩
[14] ⟨*MultExpr*⟩ ::= ⟨*Expr*⟩ (“*” | “idiv”) ⟨*Expr*⟩
[15] ⟨*Union*⟩ ::= ⟨*Expr*⟩“|”⟨*Expr*⟩
[16] ⟨*Step*⟩ ::= “.” | “..” | ⟨*Name*⟩ | “@”⟨*Name*⟩ | “*” | “@*” | “text()”
[17] ⟨*Filter*⟩ ::= ⟨*Expr*⟩“[”⟨*Expr*⟩“]”
[18] ⟨*Path*⟩ ::= ⟨*Expr*⟩(“/” | “//”)⟨*Expr*⟩
[19] ⟨*Literal*⟩ ::= ⟨*String*⟩ | ⟨*Integer*⟩
[20] ⟨*EmpSeq*⟩ ::= “()”
[21] ⟨*Constr*⟩ ::= “element” “{”⟨*Expr*⟩“}” “{”⟨*Expr*⟩“}” |
::= “attribute” “{”⟨*Expr*⟩“}” “{”⟨*Expr*⟩“}” |
::= “text” “{”⟨*Expr*⟩“}” | “document” “{”⟨*Expr*⟩“}”
[22] ⟨*TypeSw*⟩ ::= “typeswitch ”“(”⟨*Expr*⟩“)” (“case” ⟨*Type*⟩ “return”⟨*Expr*⟩)+
::= “default” “return”⟨*Expr*⟩
[23] ⟨*Type*⟩ ::= “xs:boolean” | “xs:integer” | “xs:string” | “element()” |
::= “attribute()” | “text()” | “document-node()”
[24] ⟨*FunCall*⟩ ::= ⟨*Name*⟩“(”(⟨*Expr*⟩(“,”⟨*Expr*⟩)*)?“)”

Figure A.1: The syntax of LiXQuery [Hidders *et al.*, 2004] presented in the Extended Backus-Naur Form (EBNF) notation.

Bibliography

Abiteboul, S., Hull, R., and Vianu, V. (1995). *Foundations of Databases*. Addison Wesley.

ACM (2008). SIGMOD/PODS Conference. `http://www.sigmod08.org/`.

Afanasiev, L. and Marx, M. J. (2006). An analysis of the current XQuery benchmarks. In *Proceedings of the 1st International Workshop on Performance and Evaluation of Data Management Systems (ExpDB 2006)*, Chicago, Illinois, USA. ACM Press.

Afanasiev, L. and Marx, M. J. (2008). An analysis of XQuery benchmarks. *Information Systems, Special Issue "Performance Evaluation of Data Management Systems"*, **33**(2), 155–181.

Afanasiev, L. and ten Cate, B. (2009). On Core XPath with Inflationary Fixed Points. In *Proceedings of the 6th Workshop on Fixed Points in Computer Science (FICS 2009)*, Coimbra, Portugal.

Afanasiev, L., Manolescu, I., and Michiels, P. (2005a). MemBeR: a micro-benchmark repository for XQuery. In *Proceedings of the 3rd International XML Database Symposium (XSym 2005)*, number 3671 in LNCS, pages 144–161. Springer.

Afanasiev, L., Blackburn, P., Dimitriou, I., Gaiffe, B., Goris, E., Marx, M., and de Rijke, M. (2005b). PDL for Ordered Trees. *Journal of Applied Non-Classical Logic*, **15**(2), 115–135.

Afanasiev, L., Franceschet, M., Marx, M. J., and Zimuel, E. (2006). XCheck: a Platform for Benchmarking XQuery Engines. In *Proceedings of the 32nd International Conference on Very Large Data Bases (VLDB 2006)*, pages 1247–1250, Seoul, Korea. ACM Press.

Afanasiev, L., ten Cate, B., and Marx, M. J. (2007). Lekker bomen. *Nieuwsbrief van de NVTI*, **11**, 38–52.

Afanasiev, L., Grust, T., Marx, M. J., Rittinger, J., and Teubner, J. (2008). An Inflationary Fixed Point Operator in XQuery. In *Proceedings of the 24th International Conference on Data Engineering (ICDE 2008)*, pages 1504–1506, Cancun, Mexico.

Afanasiev, L., Grust, T., Marx, M. J., Rittinger, J., and Teubner, J. (2009). Recursion in XQuery: Put Your distributivity Safety Belt On. In *Proceedings of the 12th International Conference on Extending Database Technology (EDBT 2009)*, pages 345–356, Saint Petersburg, Russia. ACM.

Agosti, M., Nunzio, G. M. D., and Ferro, N. (2007). Scientific Data of an Evaluation Campaign: Do We Properly Deal with Them? In *Evaluation of Multilingual and Multi-modal Information Retrieval*, volume Volume 4730/2007 of *Lecture Notes in Computer Science*, pages 11–20. Springer Berlin / Heidelberg.

Al-Khalifa, S., Jagadish, H. V., Patel, J. M., Wu, Y., Koudas, N., and Srivastava, D. (2002). Structural Joins: A Primitive for Efficient XML Query Pattern Matching. In *Proceedings of the 18th International Conference on Data Engineering (ICDE 2002)*, San Jose, CA, USA.

Altinel, M. and Franklin, M. J. (2000). Efficient Filtering of XML Documents for Selective Dissemination of Information. In *Proceedings of the 26th International Conference on Very Large Data Bases (VLDB 2000)*, pages 53–64, Cairo, Egypt.

Axyana Software (2006). Qizx/Open version 1.0 : An open-source Java implementation of XQuery. `http://www.axyana.com/qizxopen`.

Axyana Software (2009). Qizx/Open version 3.0: An open-source Java implementation of XQuery. `http://www.xmlmind.com/qizx/`.

Bancilhon, F. and Ramakrishnan, R. (1986). An Amateur's Introduction to Recursive Query Processing Strategies. In *Proceedings of the ACM SIGMOD International Conference on Management of Data (SIGMOD 1986)*, Washington, D.C., USA.

Barbosa, D., Mendelzon, A. O., Keenleyside, J., and Lyons, K. A. (2002). ToXgene: An extensible template-based data generator for XML. In *Proceedings of the 5th International Workshop on the Web and Databases (WebDB 2002)*, pages 49–54, Madison, Wisconsin, USA.

Berkeley XML DB (2009). `http://www.oracle.com/database/berkeley-db/xml/index.html`.

Beyer, K., Cochrane, R. J., Josifovski, V., Kleewein, J., Lapis, G., Lohman, G., Lyle, B., Özcan, F., Pirahesh, H., Seemann, N., Truong, T., Van der Linden, B., Vickery, B., and Zhang, C. (2005). System RX: One Part Relational, One Part XML. In *Proceedings of the ACM SIGMOD International Conference on Management of Data (SIGMOD 2005)*, pages 347–358, New York, NY, USA. ACM.

Blackburn, P., de Rijke, M., and Venema, Y. (2002). *Modal Logic*. Cambridge University Press.

Böhme, T. and Rahm, E. (2001). XMach-1: A benchmark for XML data management. In *Proceedings of Datenbanksysteme in Büro, Technik und Wissenschaft (BTW 2001)*, Oldenburg, Germany.

Boncz, P., Grust, T., van Keulen, M., Manegold, S., Rittinger, J., and Teuber, J. (2006a). MonetDB/XQuery: a fast XQuery processor powered by a relational engine. In *Proceedings of the 25th ACM SIGMOD International Conference on Management of Data (SIGMOD 2006)*, pages 479–490, Chicago, Illinois, USA.

Boncz, P., Grust, T., van Keulen, M., Manegold, S., Rittinger, J., and Teuber, J. (2006b). MonetDB/XQuery. An XQuery Implementation. `http://monetdb.cwi.nl/XQuery`.

Börger, E., Grädel, E., and Gurevich, Y. (1997). *The Classical Decision Problem*. Springer, Berlin.

Bosak, J. (1999). Shakespeare. `http://www.ibiblio.org/xml/examples/shakespeare/`.

Bressan, S., Lee, M. L., Li, Y. G., Wadhwa, B., Lacroix, Z., Nambiar, U., and Dobbie, G. (2001a). The X007 Benchmark. `http://www.comp.nus.edu.sg/~ebh/XOO7.html`.

Bressan, S., Dobbie, G., Lacroix, Z., Lee, M., Li, Y., Nambiar, U., and Wadhwa, B. (2001b). X007: Applying 007 benchmark to XML query processing tool. In *Proceedings of the 10th International Conference on Information and Knowledge Management (CIKM 2001)*, pages 167–174, Atlanta, Georgia, USA.

Brundage, M. (2004). *XQuery: The XML Query Language*. Addison-Wesley Professional.

BumbleBee (2006). BumbleBee: An XQuery Test Harnest. `http://www.xquery.com/bumblebee/`.

Carey, M. J., DeWitt, D. J., and Naughton, J. F. (1993). The 007 benchmark. In *Proceedings of the ACM SIGMOD International Conference on Management of Data (SIGMOD 1993)*, pages 12–21, Washington, D.C., USA. ACM.

Chamberlin, D., Robie, J., and Florescu, D. (2000). Quilt: An XML Query Language for Heterogeneous Data Sources. In *Proceedings of the 3rd International Workshop on the Web and Databases (WebDB 2000)*, LNCS, pages 53–62, Dallas, Texas, USA. Springer-Verlag.

Cohen, P. R. (1995). *Empirical Methods for Artificial Intelligence*. Bradford Books.

Dawar, A., Grädel, E., and Kreutzer, S. (2004). Inflationary Fixed Points in Modal Logic. *ACM Transations on Computational Logic*, **5**, 282–315.

Diaz, A. L. and Lovell, D. (1999). IBM Alpha Works XML Generator. `http://www.alphaworks.ibm.com/tech/xmlgeneratorhp`.

EXSLT (2006). EXSLT: Extensions to XSLT. `http://www.exslt.org/`.

Fan, W., Qeerts, F., Jia, X., and Kementsietsidis, A. (2006). SMOQE: A System for Providing Secure Access to XML. In *Proceedings of the 28th International Conference on Very Large Dada Bases (VLDB 2006)*, pages 1227–1230, Seoul, Korea.

Fan, W., Geerts, F., Jia, X., and Kementsietsidis, A. (2007). Rewriting Regular XPath Queries on XML Views. In *Proceedings of the 23rd International Conference on Data Engineering (ICDE 2007)*, pages 666–675, Istanbul, Turkey.

Fernández, M., Siméon, J., Chen, C., Choi, B., Gapeyev, V., Marian, A., Michiels, P., Onose, N., Petkanics, D., Ré, C., Stark, M., Sur, G., Vyas, A., and Wadler, P. (2006). Galax: An implementation of XQuery. `http://www.galaxquery.org`.

Fiebig, T., Helmer, S., Kanne, C.-C., Moerkotte, G., Neumann, J., Schiele, R., and Westmann, T. (2002). Anatomy of a Native XML Base Management System. *The VLDB Journal*, **11**(4), 292–314.

Galindo-Legaria, C. and Joshi, M. (2001). Orthogonal Optimization of Subqueries and Aggregation. In *Proceedings of the ACM SIGMOD International Conference on Management of Data (SIGMOD 2001)*, pages 571–581, New York, NY, USA. ACM.

Georgetown Protein Information Resource (2001). Protein Sequence Database. `http://www.cs.washington.edu/research/xmldatasets/`.

Gottlob, G. and Koch, C. (2002). Monadic Queries over Tree-Structured Data. In *Proceedings of the 17th IEEE Symposium on Logic in Computer Science (LICS 2002)*, pages 189–202.

Gottlob, G., Koch, C., and Pichler, R. (2005). Efficient Algorithms for Processing XPath Queries. *ACM Transactions on Database Systems (TODS)*, **30**(2), 444–491.

Gou, G. and Chirkova, R. (2007). Efficiently Querying Large XML Data Repositories: A Survey. *IEEE Transactions on Knowledge and Data Engineering*, **19**(10), 1381–1403.

Grinev, M. and Lizorkin, D. (2004). XQuery Function Inlining for Optimizing XQuery Queries. In *Proceedings of the 8th East-European Conference on Advances in Databases and Information Systems (ADBIS 2004)*.

Grust, T., van Keulen, M., and Teubner, J. (2003). Staircase Join: Teach a Relational DBMS to Watch its (Axis) Steps. In *Proceedings of the 29th International Conference on Very Large Data Bases (VLDB 2003)*, Berlin, Germany.

Grust, T., Sakr, S., and Teubner, J. (2004). XQuery on SQL Hosts. In *Proceedings of the 30th International Conference on Very Large Data Bases (VLDB 2004)*, pages 252–263.

Gulutzan, P. and Pelzer, T. (1999). *SQL-99 Complete, Really*. CMP Books.

Güntzer, U., Kieüling, W., and Bayer, R. (1987). On the Evaluation of Recursion in (Deductive) Database Systems by Efficient Differential Fixpoint Iteration. In *Proceedings of the 3rd International Conference on Data Engineering (ICDE 1987)*, pages 120–129.

Han, J., Qadah, G. Z., and Chaou, C. (1988). The Processing and Evaluation of Transitive Closure Queries. In *Proceedings of the 1st International Conference on Extending Database Technology (EDBT 1988)*, pages 49–75.

Hidders, J., Paredaens, J., Vercammen, R., and Demeyer, S. (2004). A Light but Formal Introduction to XQuery. In *Proceedings of the 2nd International XML Database Symposium (XSym 2004)*, pages 5–20.

Ioannidis, Y. E. (1986). On the Computation of the Transitive Closure of Relational Operators. In *Proceedings of the 12th International Conference on Very Large Data Bases (VLDB 1986)*, pages 403–411.

Jagadish, H. V., Lakshmanan, L. V. S., Srivastava, D., and Thompson, K. (2001). TAX: A Tree Algebra for XML. In *Proceedings 8th International Workshop on Database Programming Languages (DBPL 2001)*, pages 149–164.

Jagadish, H. V., Al-Khalifa, S., Chapman, A., Lakshmanan, L. V. S., Nierman, A., Paparizos, S., Patel, J. M., Srivastava, D., Wiwatwattana, N., Wu, Y., and Yu, C. (2002). TIMBER: A native XML database. *The VLDB Journal*, **11**(4), 274–291.

Jain, R. K. (1991). *The Art of Computer Systems Performance Analysis: Techniques for Experimental Design, Measurement, Simulation, and Modeling*. Wiley Publishing.

Janin, D. and Walukiewicz, I. (1996). On the Expressive Completeness of the Propositional mu-Calculus with Respect to Monadic Second Order Logic. In *Proceedings of the 7th International Conference on Concurrency Theory (CONCUR 1996)*, pages 263–277.

Kay, M. H. (2009). SaxonB. An XSLT and XQuery processor. `http://saxon.sourceforge.net`.

Keppel, G. (1973). *Design and Analysis*. Prentice-Hall, Inc., Englewood Cliffs, NJ.

Kepser, S. (2004). A simple proof of the Turing-completeness of XSLT and XQuery. In *Proceedings of the Extreme Markup Languages Conference*, Montréal, Canada.

Koch, C. (2004). XML TaskForce XPath. `http://www.xmltaskforce.com`.

Koch, C. (2006). Arb: A highly scalable query engine for expressive node-selecting queries on (XML) trees. `http://www.infosys.uni-sb.de/~koch/projects/arb/`.

Krishnamurthy, R., Kaushik, R., and Naughton, J. F. (2003). XML-SQL Query Translation Literature: The State of the Art and Open Problems. In *Proceedings of the 1st International XML Database Symposium (XSym 2003)*, pages 1–18.

Lei, H. and Ross, K. A. (1998). Faster Joins, Self-Joins, and Multi-Way Joins Using Join Indices. *Data and Knowledge Engineering*, **28**(3), 277–298.

Manegold, S. (2008). An Empirical Evaluation of XQuery Processors. *Information Systems*, **33**(2), 203–220.

Manolescu, I. and Manegold, S. (2007). Performance Evaluation and Experimental Assessment—Conscience or Curse of Database Research? In *Proceedings of the 33rd International Conference on Very Large Data Bases (VLDB 2007)*, pages 1441–1442, Vienna, Austria.

Manolescu, I. and Manegold, S. (2008). Performance Evaluation in Database Research: Principles and Experience. In *Proceedings of the IEEE International Conference on Data Engineering (ICDE 2008)*, Cancun, Mexico.

Manolescu, I., Afanasiev, L., Arion, A., Dittrich, J., Manegold, S., Polyzotis, N., Schnaitter, K., Senellart, P., Zoupanos, S., and Shasha, D. (2008a). The repeatability experiment of SIGMOD 2008. *SIGMOD Record*, **37**(1), 39–45.

Manolescu, I., Miachon, C., and Michiels, P. (2008b). Towards micro-benchmarking XQuery. *Information Systems*, **33**(2), 182–202.

Marx, M. (2004). XPath with Conditional Axis Relations. In *Proceedings of the 9th International Conference on Extending Database Technology (EDBT 2004)*, pages 477–494.

Meier, W. (2006). eXist. Open Source Native XML Database. `http://exist.sourceforge.net`.

Melton, J. and Simon, A. R. (2002). *SQL: 1999 - Understanding Relational Language Components*. Morgan Kaufmann.

Michael Ley (2006). DBLP XML Records. `http://dblp.uni-trier.de/xml/`.

Miklau, G. and Suciu, D. (2002). Containment and Equivalence for an XPath Fragment. In *Proceedings of the 21st ACM Symposium on Principles of Database Systems (PODS 2002)*, pages 65–76.

Mishra, P. and Eich, M. H. (1992). Join Processing in Relational Databases. *ACM Computing Surveys*, **24**(1), 63–113.

Murthy, R., Liu, Z. H., Krishnaprasad, M., Chandrasekar, S., Tran, A.-T., Sedlar, E., Florescu, D., Kotsovolos, S., Agarwal, N., Arora, V., and Krishnamurthy, V. (2005). Towards an Enterprise XML Architecture. In *Proceedings of the ACM SIGMOD International Conference on Management of Data (SIGMOD 2005)*, pages 953–957.

NASA (2001). NASA XML Project. `http://www.cs.washington.edu/research/xmldatasets/www/repository.html#nasa`.

Nentwich, C., Capra, L., Emmerich, W., and Finkelstein, A. (2002). xlinkit: A Consistency Checking and Smart Link Generation Service. *ACM Transactions on Internet Technology*, **2(2)**, 151–185.

Nicola, M. and van der Linden, B. (2005). Native XML Support in DB2 Universal Database. In *Proceedings of the 31st International Conference on Very Large Data Bases (VLDB 2005)*, pages 1164–1174.

Nicola, M., Kogan, I., and Schiefer, B. (2007). An XML transaction processing benchmark. In *Proceedings of the ACM SIGMOD International Conference on Management of Data (SIGMOD 2007)*, pages 937–948.

Pal, S., Cseri, I., Seeliger, O., Rys, M., Schaller, G., Yu, W., Tomic, D., Baras, A., Berg, B., Churin, D., and Kogan, E. (2005). XQuery Implementation in a Relational Database System. In *Proceedings of the 31st International Conference on Very Large Data Bases (VLDB 2005)*, pages 1175–1186.

Paparizos, S., Wu, Y., Lakshmanan, L., and Jagadish, H. (2004). Tree Logical Classes for Efficient Evaluation of XQuery. In *Proceedings of the ACM SIGMOD International Conference on Management of Data (SIGMOD 2004)*, Paris, France.

Park, C., Min, J., and Chung, C. (2002). Structural Function Inlining Techniques for Structurally Recursive XML Queries. In *Proceedings of the 28th International Conference on Very Large Data Bases (VLDB 2002)*, pages 83–94, Hong Kong, China.

Qexo (2006). Qexo - The GNU Kawa implementation of XQuery. `http://www.gnu.org/software/qexo/`.

Runapongsa, K., Patel, J., Jagadish, H., Chen, Y., and Al-Khalifa, S. (2002). The Michigan Benchmark: A Microbenchmark for XML Query Processing Systems. In *Proceedings of the 1st International Workshop on Efficiency and Effectiveness of XML Tools and Techniques(EEXTT 2002)*, pages 160–161.

Runapongsa, K., Patel, J., Jagadish, H., Chen, Y., and Al-Khalifa, S. (2003). The Michigan Benchmark: Towards XML Query Performance Diagnostics. Technical report, The University of Michigan.

Sahuguet, A., Dupont, L., and Nguyen, T.-L. (2000). Kweelt: a framework to query XML data. `http://kweelt.sourceforge.net/`.

Schmidt, A. R., Waas, F., Kersten, M. L., Florescu, D., Manolescu, I., Carey, M. J., and Busse, R. (2001). The XML Benchmark Project. Technical Report INS-R0103, CWI, Amsterdam, The Netherlands.

Schmidt, A. R., Waas, F., Kersten, M. L., Carey, M. J., Manolescu, I., and Busse, R. (2002). XMark: A Benchmark for XML Data Management. In *Proceedings of the 28th International Conference on Very Large Data Bases (VLDB 2002)*, pages 974–985, Hong Kong, China.

Swiss-Prot and TrEMBL (1998). SwissProt Protein Sequence Database. `http://www.cs.washington.edu/research/xmldatasets`.

ten Cate, B. (2006a). Regular XPath: Algebra, Logic and Automata. Unpublished note presented at AUTOMATHA Workshop on Algebraic Theory of Automata and Logic.

ten Cate, B. (2006b). The Expressivity of XPath with Transitive Closure. In *Proceedings of the 25th ACM Symposium on Principles of Database Systems (PODS 2006)*, pages 328–337.

ten Cate, B. and Marx, M. (2009). Axiomatizing the logical core of XPath 2.0. *Theory of Computing Systems*, **44**(4), 561–589.

Thomas, W. (1997). Languages, automata, and logic. In *Handbook of formal languages*, volume 3: Beyond words, pages 389–455. Springer-Verlag New York, Inc., New York, NY, USA.

TPC (2009). Transaction Processing Performance Council. `http://www.tpc.org/`.

Treebank (2002). Penn Treebank: A corpus of parsed sentences. `http://www.cs.washington.edu/research/xmldatasets/data/treebank`.

University of Antwerp (2006). Blixem: a LiXQuery engine. `http://adrem.ua.ac.be/~blixem/`.

Williams, T., Kelley, C., Lang, R., Kotz, D., Campbell, J., Elber, G., and Woo, A. (2008). Gnuplot: a function plotting utility. `http://www.gnuplot.info/`.

World Wide Web Consortium (1998). Document Object Model (DOM) Version 1.0. http://www.w3.org/DOM/.

World Wide Web Consortium (1999a). XML Path Language, Version 1.0 (XPath 1.0). http://www.w3.org/TR/xquery.

World Wide Web Consortium (1999b). XSL Transformations (XSLT) Version 1.0. http://www.w3.org/TR/xslt.

World Wide Web Consortium (2004a). XML Information Set (Second Edition). http://www.w3.org/TR/xml-infoset/.

World Wide Web Consortium (2004b). XML Schema, Version 1.0 (Second Edition). http://www.w3.org/TR/xmlschema-0/.

World Wide Web Consortium (2006a). XQuery Test Suite, Release Version 1.0.2. http://www.w3.org/XML/Query/test-suite/.

World Wide Web Consortium (2006b). XQuery Test Suite Result Summary. `http://www.w3.org/XML/Query/test-suite/XQTSReport.html`.

World Wide Web Consortium (2007). XML Path Language (XPath) 2.0. http://www.w3.org/TR/xpath20/.

World Wide Web Consortium (2007). XQuery 1.0: An XML Query Language. http://www.w3.org/TR/xquery.

World Wide Web Consortium (2007a). XQuery 1.0 and XPath 2.0 Data Model (XDM). http://www.w3.org/TR/xpath-datamodel/.

World Wide Web Consortium (2007b). XQuery 1.0 and XPath 2.0 Formal Semantics. http://www.w3.org/TR/xquery-semantics/.

World Wide Web Consortium (2007c). XSL Transformations (XSLT) Version 2.0. http://www.w3.org/TR/xslt20/.

World Wide Web Consortium (2008). Extensible Markup Language (XML) 1.0 (Fifth Edition). http://www.w3.org/TR/REC-xml.

World Wide Web Consortium (2009a). XQuery and XPath Full Text 1.0. `http://www.w3.org/TR/xpath-full-text-10/`.

World Wide Web Consortium (2009b). XQuery Update Facility 1.0. `http://www.w3.org/TR/xquery-update-10/`.

X-Hive/DB (2005). An XML Database System Management. `http://www.x-hive.com/products/db/index.html`.

Xalan (2002). An implementation of XPath 1.0. http://xalan.apache.org/.

XSLTMark (2006). XSLTMark: a benchmark for XSLT. `http://www.datapower.com/xmldev/xsltmark.html`.

Yao, B., Özsu, T., and Khandelwal, N. (2004). XBench Benchmark and Performance Testing of XML DBMSs. In *Proceedings of the 20th International Conference on Data Engineering (ICDE 2004)*, pages 621–633.

Yao, B. B., Özsu, M. T., and Keenleyside, J. (2002). XBench–A Family of Benchmarks for XML DBMSs. In *Proceedings of the 1st International Workshop on Efficiency and Effectiveness of XML Tools and Techniques(EEXTT 2002)*, pages 162–164.

Summary

In this thesis, we pursue two main research themes: performance evaluation and optimization of XML query processing. Our focus is on the XQuery query language. These themes are tightly connected, since performance evaluation is aimed at measuring the success of optimization techniques. More specifically, we pursue *benchmarking as a performance evaluation technique* on the one hand, and *optimization techniques for recursion* in XQuery, on the other hand.

In the first part of the thesis, we develop benchmarking methodology and tools for XQuery. We start by analyzing XQuery benchmarks published by 2006, namely XMach-1, XMark, X007, MBench, and XBench, and survey their usage. We analyze and compare their workloads (i.e., the data and query sets) and measures. We also execute the benchmarks on four XQuery engines and analyze the results. Next, we discuss how to achieve the repeatability of experimental evaluations of computer systems in the database domain. As part of setting up a methodology for repeatability, we perform a review of articles submitted to the research conference SIGMOD 2008 and measure the repeatability of the presented experimental evaluations. Further, we address the problems and challenges of automating the execution of performance evaluation benchmarks on many XML query engines and comparison of their performance. We present a software tool, XCheck, as a solution to these problems. As a result of our analysis of XQuery benchmarks, we identify a lack of suitable tools for precise and comprehensive performance evaluations. We address this problem by developing a methodology for micro-benchmarking XML query engines, which we refer to as MemBeR. MemBeR also comprises a framework for collecting and storing micro-benchmarks. Finally, we present a MemBeR-style micro-benchmark for testing performance of value-based joins expressed in XQuery. We evaluate the micro-benchmark by analyzing the performance of four XQuery engines.

In the second part of the thesis, we investigate declarative means of obtaining recursion in XQuery. Namely, we add an inflationary fixed point operator to XQuery. We propose an optimization technique for processing this opera-

tor. This optimization relies on a distributivity property of XQuery expressions. Further, we implement this technique on top of the XML database system, MonetDB/XQuery, and evaluate its performance using the tools developed in the first part of the thesis. Finally, we investigate the theoretical aspects of this inflationary fixed point operator in the context of Core XPath, the XML tree navigational core of XPath and XQuery. We prove that the satisfiability problem of Core XPath extended with the inflationary fixed point operator is undecidable.

Samenvatting

In dit proefschrift houden we ons bezig met twee belangrijke onderzoeksthema's: prestatie-analyse en optimalisatie van XML query verwerking. De nadruk ligt op de XQuery querytaal. Deze thema's zijn nauw met elkaar verbonden, aangezien prestatie-analyse is gericht op het meten van het succes van optimalisatietechnieken. Meer specifiek bestuderen we *benchmarking als een prestatie-analyse methode* aan de ene kant, en *optimalisatietechnieken voor recursie* in XQuery, aan de andere kant.

In het eerste deel van het proefschrift ontwikkelen we een benchmarking-methodologie en hulpmiddelen voor XQuery. We beginnen met het analyseren van XQuery benchmarks die in of voor 2006 gepubliceerd zijn, namelijk XMach-1, XMark, X007, MBench, en XBench, en brengen hun gebruik in kaart. We analyseren en vergelijken hun werklast (d.w.z. de data en de verzamelingen queries) en maten. We voeren de benchmarks ook uit op vier XQuery systemen en analyseren de resultaten. Vervolgens bespreken we hoe herhaalbaarheid van experimentele evaluaties van computersystemen kan worden bereikt in het databasedomein. In het kader van het opzetten van een methodologie voor herhaalbaarheid, voeren we een studie uit van ingezonden artikelen voor de onderzoeksconferentie SIGMOD 2008 en meten we de herhaalbaarheid van de gepresenteerde experimentele evaluaties. Verder gaan we in op de problemen en uitdagingen van het automatiseren van de uitvoering van prestatie-analyse benchmarks op veel XML query systemen en de vergelijking van hun prestaties. We presenteren een software tool, XCheck, als een oplossing voor deze problemen. Op basis van onze analyse van XQuery benchmarks identificeren we een gebrek aan geschikte hulpmiddelen voor nauwkeurige en uitgebreide evaluatie. We pakken dit probleem aan door het ontwikkelen van een methodiek voor het micro-benchmarken van XML query systemen, onder de naam MemBeR. MemBeR omvat ook een raamwerk voor het verzamelen en opslaan van micro-benchmarks. Ten slotte presenteren we een micro-benchmark in de geest van MemBeR voor het testen van de prestaties van value-based joins uitgedrukt in XQuery. We evalueren de micro-benchmark aan

de hand van een analyse van de prestaties van de vier XQuery systemen.

In het tweede deel van het proefschrift onderzoeken we declaratieve middelen voor het uitdrukken van recursie in XQuery. We voegen een inflationaire dekpuntsoperator toe aan XQuery. We stellen een optimalisatie techniek voor de verwerking van de operator voor, die is gebaseerd op een distributiviteitseigenschap van XQuery-expressies. Verder implementeren we deze techniek bovenop het XML-database systeem MonetDB/XQuery, en evalueren we de prestaties met behulp van de ontwikkelde instrumenten uit het eerste deel van het proefschrift. Tenslotte bestuderen we de theoretische aspecten van deze inflationaire dekpuntsoperator in de context van Core XPath, de kern van XPath en XQuery die geschikt is voor het beschrijven van navigatie in XML-bomen. We bewijzen dat het vervulbaarheidsprobleem van Core XPath uitgebreid met de IFP operator onbeslisbaar is.