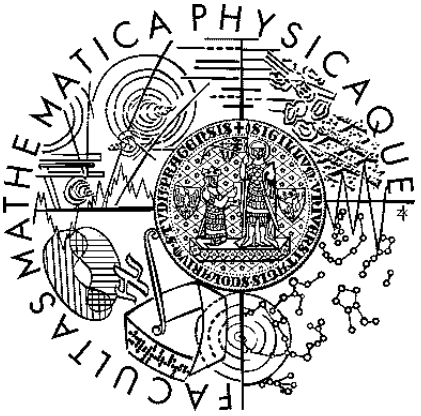
Univerzita Karlova v Praze
Matematicko-fyzikální fakulta

DIPLOMOVÁ PRÁCE



Maroš Vranec

XML Benchmarking

Katedra softwarového inženýrství

Vedoucí diplomové práce: RNDr. Irena Mlýnková, Ph.D.,

Studijni program: informatika

2008

First of all I wish to express my sincere gratitude and appreciation to my supervisor, Irena Mlýnková, for her planning the project of my work, her thoughtful guidance, her valuable suggestions, her precious comments during discussions, her kind supervision with me and her continuous encouragement to make my research work successful.

Prohlašuji, že jsem svou diplomovou práci napsal samostatně a výhradně s použitím citovaných pramenů. Souhlasím se zapůjčováním práce.

V Praze dne 8.8. Maroš Vranec

Contents

List of Figures

List of Tables

Název práce: XML Benchmarking
Autor: Maroš Vranec
Katedra (ústav): Katedra softwarového inženýrství
Vedoucí diplomové práce: RNDr. Irena Mlýnková, Ph.D.
e-mail vedoucího: Irena.Mlynkova@mff.cuni.cz

Abstrakt: V této práci studujeme možnosti a omezení benchmarkování XML databází. Práci začneme analýzou stávajících řešení, pokračovat budeme návrhem a implementací našeho generátoru XML benchmarků a zakončíme ji použitím benchmarků na vybranou skupinou databází. Navrhovaný benchmark je unikátní tím, že generuje testovací data včetně operací nad nimi (místo pevné sady dotazů). Text obsahuje i experimentální výsledky, které ukazují výhody a nevýhody tohoto řešení.

Klíčová slova: XML, XMLMS, generace, benchmarkování

Title: XML Benchmaring
Author: Maroš Vranec
Department: Department of Software Engineering
Supervisor: RNDr. Irena Mlýnková, Ph.D.
Supervisor's e-mail address: Irena.Mlynkova@mff.cuni.cz

Abstract: In the presented work we study possiblities and limitations of benchmaring XML management systems (XMLMSs). We start by an analysis of existing solutions, continue by proposal and implementation of our XMLMS benchmark generator and finish with an application of it on a selected group of XMLMSs. The proposed benchmark is unique because it generates both testing data sets and operations (instead of being just a single fixed set). The work includes experimental results that depict advantages and disadvateges of the solution.

Keywords: XML, XMLMS, generating, benchmarking

Chapter 1

Introduction

1.1 Background

Since the extensible markup language (XML) [Bray06] is a simple human-readable way to represent data, it is being extensively used in a great variety of application domains, not only in software companies. An exploitation of XML documents occurs in banks, medicine institutions, governments and markets of all kinds to name just a few. Due to its increasing popularity the XML became very wide-spread - now it occurs almost everywhere where the software is.

With the growing amount of generated and hand-written XML documents the need for their management rises as well. The first (or the second) idea which comes to mind is to use existing mature relational database management systems (RDBMSs) to store and manage XML data. Other ways are possible too - e.g. storing XML database in its original form, the XML files. The databases of XML documents are called XML management systems (XMLMSs [Bressan01] for short).

1.2 Motivation for Benchmarking XMLMSs

Managing XML data one way or another, it is apparent that XMLMSs have much in common with managing other kinds of data as in RDBMSs. It is useful to benchmark RDBMSs basically for three reasons [Bressan03]:

1. to find out which operations are supported by existing solutions;
2. to measure the actual state of the art, i.e. to determine which of current implementations is better for a particular purpose; and
3. to motivate a research in areas where existing solutions do not perform well.

Since XMLMSs have so much in common with RDBMSs, the benchmarking of XMLMSs should be of similar use.

In the industry of XML management the need for benchmark is even more evident. The reason is that XML data are ubiquitous and therefore very heterogeneous. An obvious example can be XML data which must

preserve an order of elements (e.g. an XHTML [XHTML00] page) and one that must not (an unordered catalog for example). Each XMLMS can be optimized for some class of XML data but might not be efficient in managing all of them. So there is a need for determining which XMLMS is efficient for which application.

1.3 Aim of this Work

The purpose of this work is a research on possibilities and limitations of XML benchmarking projects that enable to test performance of XMLMSs.

The method used to construct this thesis consists of three parts:

1. literature study,
2. proposal and implementation of an XML benchmark and
3. application of it.

First of all we analyze existing solutions and projects and discuss their advantages and disadvantages. The core of the work is a proposal and implementation of own benchmarking project that focuses on statistically frequent XML patterns. It involves **a benchmark generator** which generates the XML documents, their schemes and the workload (queries to an XMLMS), according to the existing analysis of XML databases. As far as we can tell this approach (generating everything) has never been used before. Finally we produce suitable experimental results that depict the advantages and disadvantages of the proposal.

1.4 Structure of this Document

Chapter 2 contains formal definitions used in this document. Terms like XML document, various parameters of XML files, XML benchmarking, XQuery and others are precisely defined, if possible, or, at least, explained using typical examples.

Chapter 3 is an analysis of existing solutions. It is a brief description and comparison of most-known XML benchmarks.

Chapter 4 is a proposal of a solution to the problem of benchmarking the XMLMS performance based on disadvantages and open issues identified in Chapter 3. It will be a generator of XMLMS benchmark (both data and queries will be generated).

Chapter 5 contains implementation details of our benchmark solution. It includes technical details how benchmark generator was written, what restrictions solution has and what decisions were made during the development.

Chapter 6 is an example of our benchmark in action. It includes a step by step tutorial for using a benchmark for real-world XMLMS. There are also illustrative results of benchmarking selected real-world XMLMSs against each other.

Chapter 7 is a recapitulation and discussion of advantages and disadvantages of the proposal. Suggestions for future work are also included.

Chapter 2

Technologies and Formal Definitions

Definitions in this chapter are taken mainly from [Mlynkova06]. Step by step we define all terms needed to understand the rest of this paper. We start by basic definitions (e.g. what is an XML document), continue by characteristics of XML documents (e.g. what is a depth of an XML document) and their schemes and finish by benchmark related definitions (e.g. what is an XMLMS). Definitions also contain examples when it is appropriate.

2.1 Basic Definitions

Example 2.1 shows us a fragment of an XML document [Bray06]. It has a *root element*, in this particular case named 'addresses'. Other *elements* are children of the root element forming a tree structure (which is called an *XML Tree*). Note that the 'person' element in the example also has an *attribute* with name 'idnum' and value '0123'. There are also examples of text content of the elements, e.g. '100 Main Street' is one of them. "'" is a processing instruction telling processor to translate it to the apostrophe character '.

Example 2.1 An example of an XML document.

```
<addresses>
    <!-- This is a comment about following person. -->
    <person idnum="0123">
        <lastName>Doe</lastName>
        <firstName>John</firstName>
        <phone location="mobile">(201) 345-6789</phone>
        <email>jdoe@foo.com</email>
        <address>
            <street>100 Main Street</street>
            <city>O&apos;Hara Township</city>
            <state>New Jersey</state>
            <zip>07670</zip>
        </address>
    </person>
</addresses>
```

An *XML document* is a finite ordered tree T = (A; N; E; r) where A is a finite alphabet, N is a set of nodes of the tree, E is a set of edges of the tree, and r $\in$ N denotes a root element of the tree. Each node n $\in$ N is associated with a type of the node which can be one the following: *element, attribute, text, processing instruction*, or *comment*. Nodes with element or attribute type are also associated with a node label l $\in$ A called an *element name* or an *attribute name* respectively. The tree T is called *A-tree*. Figure 2.1 is the example of an A-tree.

Note
For simplicity, we use the terms element and attribute for nodes of the respective type.

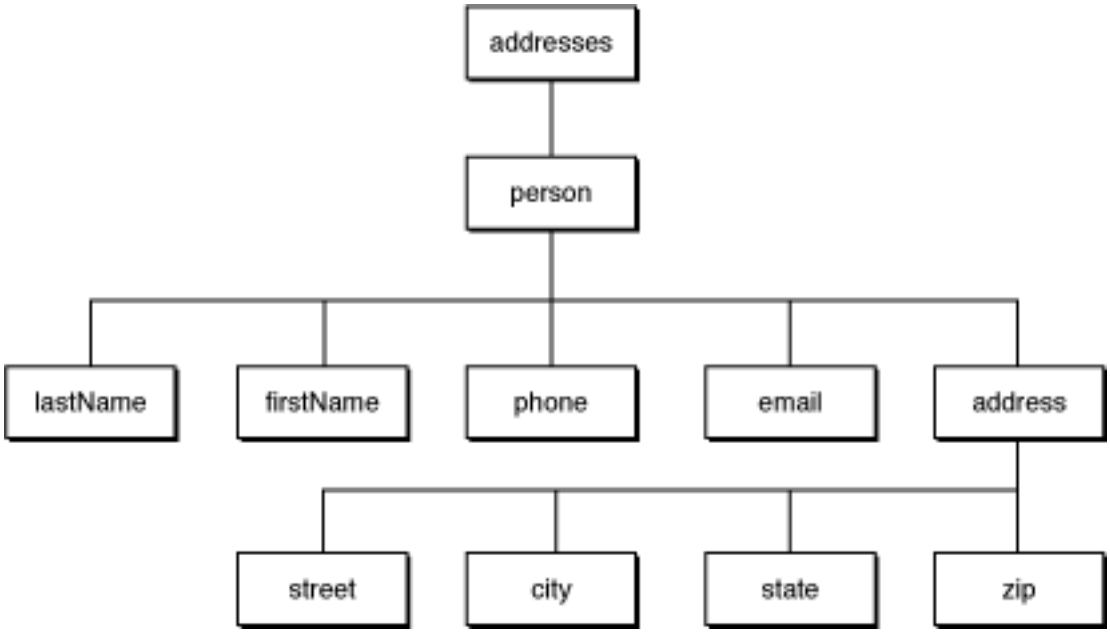


Figure 2.1: An example of an XML tree (just names of elements included).

A *DTD* is a collection of element declarations of the form e $\rightarrow \alpha$ where e $\in$ A is an element name and α is its content model, i.e. regular expression over A. The *content model* α is defined as $\alpha = \varepsilon$ | pcdata | f | (α_1, α_2, ..., α_n) | ($\alpha_1|\alpha_2|...|\alpha_n$) | β* | β+ | β?, where ε denotes the empty content model, pcdata denotes the text content, f denotes a single element name, "," and "|" stand for concatenation and union (of content models α_1, α_2, ..., α_n), and "*", "+", and "?" stand for zero or more, one or more, and optional occurrence(s) (of content model β) respectively. One of the element names s $\in$ A is called a start symbol.

Example 2.2 An example of a DTD.

```
<!ELEMENT address ( street, city, state, zip ) >
<!ELEMENT addresses ( person ) >
<!ELEMENT city ( #PCDATA ) >
<!ELEMENT email ( #PCDATA ) >
<!ELEMENT firstName ( #PCDATA ) >
<!ELEMENT lastName ( #PCDATA ) >
<!ELEMENT person ( lastName, firstName+, phone, email*, address ) >
<!ATTLIST person idnum NMTOKEN #REQUIRED >
<!ELEMENT phone ( #PCDATA ) >
<!ATTLIST phone location NMTOKEN #REQUIRED >
<!ELEMENT state ( #PCDATA ) >
<!ELEMENT street ( #PCDATA ) >
<!ELEMENT zip ( #PCDATA ) >
```

For instance, the element 'person' in Example 2.2 can have one or more 'firstName'-s as child elements.

An A-tree *satisfies the DTD* if its root is labeled by start symbol s and for every node n $\in$ N and its label e, the sequence e_1, e_2, ..., e_k of labels of its child nodes matches the regular expression α, where e $\rightarrow \alpha$.

Note
The XML document in Example 2.1 satisfies the DTD in Example 2.2.

2.2 Definitions of Characteristics of XML Documents

A *distance of elements* e_1 and e_2 is the number of edges in a respective A-tree separating their corresponding nodes. A *level of an element* is the distance of its node from the root node. The level of the root node is 0. A *depth of an XML document* is the largest level among all the elements.

Note
In Example 2.3, a level of the element middle_node is 2.

Example 2.3 An example of an XML document with depth of 2.

```
<root>
 <node1>
  <middle_node/>
 </node1>
 <node2>
  <late_node/>
 </node2>
</root>
```

An element name e' is *reachable from e*, denoted by e $\Rightarrow$ e', if either e $\rightarrow \alpha$ and e' occurs in α or $\exists$e" such that e $\Rightarrow$ e" and e" $\Rightarrow$ e'. An element e' is a *descendant of element* e, if l' is element name of e', l is element

name of e and l $\Rightarrow$ l'. A content model α is *derivable*, denoted by e $\Rightarrow \alpha$, if either e $\rightarrow \alpha$ or e $\Rightarrow \alpha$', e' $\rightarrow \alpha$", and $\alpha = \alpha$'[e'/α"], where α'[e'/α"] denotes the content model obtained by substituting α" for all occurrences of e' in α'. An element name e is *reachable* if s $\Rightarrow$ e, where s is the start symbol. Otherwise it is called *unreachable*.

Note
In Example 2.2 of DTD, the element 'lastName' is reachable from the element 'addresses' but not vice versa.

An element is *trivial* if it has an arbitrary amount of attributes and its content model $\alpha = \varepsilon$ | pcdata.

A content model α is *mixed*, if $\alpha = (\alpha_1|...|\alpha_n|\text{pcdata})^* \mid (\alpha_1|...|\alpha_n|\text{pcdata})+$ where n $\geq$ 1 and for $\forall$i such that $1 \leq i \leq n$ content model $\alpha_i \neq \varepsilon \wedge \alpha_i \neq$ pcdata. An element e is called *mixed-content element* (see Example 2.4) if its content model α is mixed.

Example 2.4 An example of a mixed-content element.

```
<root>
 <node1>
  This is a text within a mixed-content element.
  <middle_node/>
  Another text within node1.
 </node1>
</root>
```

An element e is *recursive* if there exists at least one element d in the same document such that d is a descendant of e and d has the same element name as e. The element-descendant association is called an *ed-pair*.

A *simple path* (in a non-recursive DTD) is a list of elements $e_1, e_2, ..., e_k$, where $e_i \rightarrow \alpha_i$ and e_{i+1} occurs in α_i for $1 \leq i < k$. The number of elements in the list is called a *length of a simple path*.

A *fan-in of an element* e is the cardinality of the set {f | f $\rightarrow \alpha$' and the element name e' of element e occurs in α'}. An element with large fan-in value is called a *hub*.

An *element fan-out of an element* e is the cardinality of the set {f | e' is the element name of element e, e' $\rightarrow \alpha$ and the element name f' of element f occurs in α}. An *attribute fan-out of an element* e is the number of its attributes.

Following two definitions are taken from [Bourret99].

Data-centric documents are documents that use XML as a data transport. They are designed for machine consumption and the fact that XML is used at all is usually superfluous. That is, it is not important to the application or the database that the data is, for some length of time, stored in an XML document. Examples of data-centric documents are sales orders, flight schedules, scientific data, and stock quotes.

Document-centric documents are (usually) documents that are designed for human consumption. Examples are books, email, advertisements, and almost any hand-written XHTML document. They are characterized by less regular or irregular structure, larger grained data (that is, the smallest independent unit of data might be at the level of an element with mixed content or the entire document itself), and lots of mixed content. The order in which sibling elements and PCDATA occurs is almost always significant.

2.3 Advanced Parameters of XML Documents

An element e is *linearly recursive* if it is recursive and for every α such that e $\Rightarrow \alpha$ e is the only recursive element that occurs in α and neither of its occurrences is enclosed by "*" or "+". An element is *non-linearly recursive* if it is recursive but it is not linearly recursive. See Example 2.5.

Example 2.5 Linearly recursive and non-linearly recursive elements.

In this example, the `lin_rec` element is linearly recursive while, `non_lin` and `non_lin2` are examples of non-linearly recursive elements.

```
<!ELEMENT root ( lin_rec, non_lin, non_lin2 ) >
<!ELEMENT lin_rec ( lin_rec, non_rec* ) >
<!ELEMENT non_rec ( #PCDATA ) >

<!ELEMENT non_lin ( non_lin*, non_rec* ) >
<!ELEMENT non_lin2 ( non_lin2, other_rec* ) >
<!ELEMENT other_rec ( other_rec ) >
```

An element e is *trivially recursive* if it is recursive and for every α such that e $\Rightarrow \alpha$ e is the only element that occurs in α and neither of its occurrences is enclosed by "*" or "+".

An element e is *linearly recursive* if it is recursive and for every α such that e $\Rightarrow \alpha$ e is the only recursive element that occurs in α and neither of its occurrences is enclosed by "*" or "+".

An element e is *purely recursive* if it is recursive and for every α such that e $\Rightarrow \alpha$ e is the only recursive element that occurs in α.

An element that is recursive but not purely recursive is called a *generally recursive* element.

2.4 Patterns in XML Documents

There are some interesting patterns of XML documents defined in [Mlynkova06]. The authors found out that some patterns of elements in XML documents are reoccurring in many examples. So it is reasonable to consider these patterns as XML properties as well.

DNA pattern was mentioned first. A nonrecursive element e is called a *DNA pattern* if it is not mixed and its content model α consists of a nonzero amount of trivial elements and one nontrivial and nonrecursive element whose occurrence is not enclosed by "*" or "+". The nontrivial subelement is called a *degenerated branch*. A *depth of a DNA pattern* e is the maximum depth of its degenerated branch.

A nonrecursive element e is called a *relational pattern* if it has an arbitrary amount of attributes, it is not mixed, and its content model $\alpha = (e_1, e_2, ..., e_n)^* \mid (e_1, e_2, ..., e_n)+ \mid (e_1 \mid e_2 \mid ... \mid e_n)^* \mid (e_1 \mid e_2 \mid ... \mid e_n)+$, where $e_1, e_2, ..., e_n$ are trivial elements. A nonrecursive element e is called a *shallow relational pattern* if it has an arbitrary amount of attributes, it is not mixed, and its content model $\alpha = f^* \mid f+$, where f is a trivial element.

2.5 Benchmark Related Terms

System under test (shortened as *SUT*) refers to a system that is being tested for correct operation.

Note
The term is used mostly in software testing.

A *database management system (DBMS)* is a system or a software designed to manage a database, and run operations on the data requested by numerous clients. A *relational database management system (RDBMS)* is a DBMS that is based on the relational model as introduced by Edgar F. Codd [Codd70]. Relational databases are the most common kind of databases in use today (assuming one does not count a file system as a database). Examples of an RDBMS are: Oracle Database [Oracle08], Microsoft SQL Server [Vithanala05], MySQL [MySQL05] and PostgreSQL [Leidecker07].

XML management system (shortened as *XMLMS*) is a system that allows queries and manipulation of XML data (similarly to RDBMS querying and manipulating generic data). Current XMLMSs can be divided into two categories: XML-enabled databases and Native XML databases. *XML-enabled databases*, typically relational databases, provide extensions for transferring data between XML documents and relations. Such systems are generally designed to store and retrieve data-centric XML documents. On the other hand, *Native XML databases* store an XML document in its native form (i.e. text) and has it as a fundamental unit of logical storage. Examples of XML-enabled databases are: Microsoft SQL Server [Vithanala05], Oracle XML DB [Oracle07]. Examples of native XML databases are: eXist [Chaudhri03], MonetDB/XQuery [Boncz06], Galax [Fernandez04], X-Hive [Webster06].

In computer science, a *benchmark* is the result of running a computer program, a set of programs, or other operations, in order to assess the relative performance of an object, by running a number of standard tests and trials against it. The term benchmark is also commonly used for specially designed benchmarking programs. These software benchmarks are, for example, run against compilers or database management systems.

Benchmarks are designed to mimic a particular type of workload on a component or system. Synthetic benchmarks do this by specially-created programs that impose the workload on the component. They are also called *micro benchmarks*. *Application benchmarks*, instead, run actual real-world programs on the system. Whilst application benchmarks usually give a much better measure of real-world performance on a given system, synthetic benchmarks still have their use for testing out individual components.

One can understand 'an XML Benchmark' in many different ways. It can test XML parsers, XMLMSs, XML document generators, etc. In this work an XML Benchmark is a benchmark which assesses XML management system(s). A typical XML benchmark consists of a data (XML documents and corresponding schema) and a workload. A workload is the amount of work assigned to performed query, execution mode, steady state, and performance metric requirements. An XML-related workload will typically be a set of queries to the database.

There are four properties of a generic benchmark that are used to compare benchmarks, which are defined in [Gray93]. They consist of relevance, portability, scalability and simplicity. Similarly to the four properties of generic benchmarks there are four properties of XML-related workload. These are suggested in [Bressan03] and could be used to compare existing benchmarks. They are:

1. If there is a restriction on XML document structure.
2. If there is a database size and load volume scalability.
3. If there is a query type variability.
4. If there is an ad hoc and open interface for schema input and operation input.

2.6 Introduction to XQuery

There are several possibilities when deciding what technology to use when making an XML-related workload (i.e. the database queries). Two most common technologies are the XQuery [Boag07] and the XPath [Clark99]. XPath is just one part of the XQuery, and therefore we will use XQuery. *XQuery* is a query language (with some programming language features) that is designed to query collections of XML data. It is semantically similar to SQL. "The mission of the XML Query project is to provide flexible query facilities to extract data from real and virtual documents on the World Wide Web, therefore finally providing the needed interaction between the Web world and the database world. Ultimately, collections of XML files will be accessed like databases."

Most of following examples of XQuery are taken from XQuery Tutorial of [W3Schools].

Example 2.6 shows an XML document `books.xml` that we will query.

Example 2.6 To-be-queried XML document (books.xml).

```
<?xml version="1.0" encoding="ISO-8859-1"?>

<bookstore>

<book category="COOKING">
  <title lang="en">Everyday Italian</title>
  <author>Giada De Laurentiis</author>
  <year>2005</year>
  <price>30.00</price>
</book>
<book category="CHILDREN">
  <title lang="en">Harry Potter</title>
  <author>J K. Rowling</author>
  <year>2005</year>
  <price>29.99</price>
</book>

<book category="WEB">
  <title lang="en">XQuery Kick Start</title>
  <author>James McGovern</author>
  <author>Per Bothner</author>
  <author>Kurt Cagle</author>
  <author>James Linn</author>
  <author>Vaidyanathan Nagarajan</author>
  <year>2003</year>
  <price>49.99</price>
</book>

<book category="WEB">
  <title lang="en">Learning XML</title>
  <author>Erik T. Ray</author>
  <year>2003</year>
  <price>39.95</price>
</book>

</bookstore>
```

The doc() function is used to query XML documents (see Example 2.7).

Example 2.7 How to query an XML document in XQuery.

```
doc("books.xml")
```

XQuery use paths to navigate through XML tree (similar to XPath) - see Example 2.8.

Example 2.8 Query using path expression.

This query selects all titles of all books in `books.xml`.

```
doc("books.xml")/bookstore/book/title
```

And the result will be:

```
<title lang="en">Everyday Italian</title>
<title lang="en">Harry Potter</title>
<title lang="en">XQuery Kick Start</title>
<title lang="en">Learning XML</title>
```

The main engine of XQuery is the FLWOR expression (which stands for For-Let-Where-Order-Return) which is kind of a generalized SELECT-FROM-HAVING-WHERE from SQL. Following Example 2.9 shows all of them. For clause is fairly intuitive: it iterates over an input sequence and calculates some value for each item in that sequence. The XQuery let clause simply declares a variable and gives it a value. Where clause specifies a condition to filter the items we are interested in. If you want the query results in sorted order, it can be achieved using the order by clause. And finally the return clause defines the items that are included in the result.

Example 2.9 Example of FLWOR expression of XQuery.

```
for $book in doc("books.xml")/bookstore/book
let $title := $book/title
where $book/price>30
order by $title
return $title
```

This query returns:

```
<title lang="en">Learning XML</title>
<title lang="en">XQuery Kick Start</title>
```

Please note that XQuery is much more robust querying language. It has a robust typing system, includes many other functions than doc() and the mechanism to declare user-defined functions as well. But the information provided here will be just enough for us to understand it and be able to use it in the following text. More information can be found in [Boag07].

2.7 Statistical Distributions

This work correlates tightly with the statistics available for XML documents (mainly from [Mlynkova06]). Since the term of statistical distribution will occur at several places in the following text, we repeat its definition for better understanding. Some examples of a statistical distribution will be provided as well.

Given a random variable $X:O \rightarrow Y$ between a probability space (O, F, P), the sample space, and a measurable space (Y,Σ), called the state space, a *statistical distribution* on (Y, Σ) is a probability measure $X * P:\Sigma \rightarrow [0,1]$ where $X * P$ is the push forward measure of P.

In other words, a statistical distribution describes the values and probabilities associated with a random event. The values must cover all of the possible outcomes of the event, while the total probabilities must sum to exactly 1, or 100%. For example, a single coin flip can take values Heads or Tails with a probability of exactly 1/2 for each; these two values and two probabilities make up the probability distribution of the single coin flipping event. This distribution is called a discrete distribution because there are a countable number of discrete outcomes with positive probabilities.

What follows are the examples of discrete statistical distributions that will be extensively used in the rest of this work.

2.7.1 Binomial Distribution

Binomial distribution is one of the best known statistical distributions. It is the distribution of the number of successes in a sequence of n independent yes/no experiments, each of which yields success with probability p. As we can see, it has two parameters - integral n and real probability p between 0 and 1.

Its probability mass function is (also see Figure 2.2):

$$\binom{n}{k} p^k (1-p)^{n-k}$$

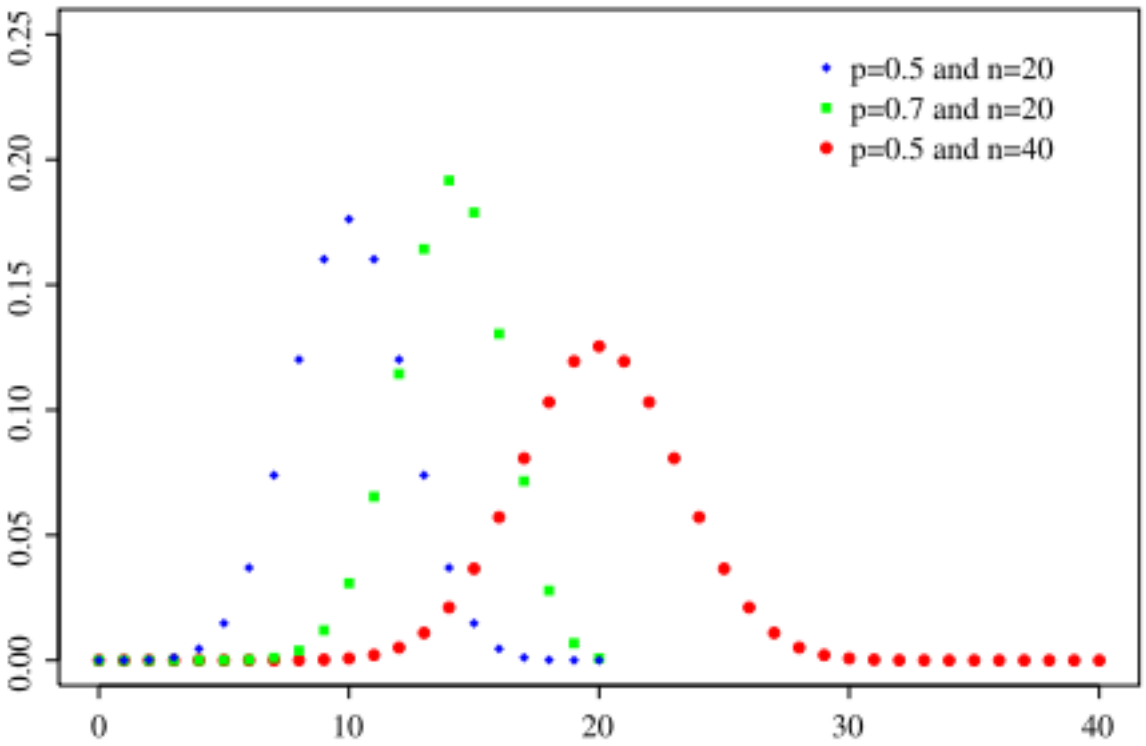


Figure 2.2: Example probability mass function of binomial distribution.

Its mean value is np and its variance is np(1-p).

Example 2.10 An elementary example of binomial distribution.

Roll a standard die ten times and count the number of sixes. The distribution of this random number is a binomial distribution with n = 10 and p = 1/6.

2.7.2 Uniform Distribution

This is another well known and very simple distribution. It is a discrete probability distribution that can be characterized by saying that all values of a finite set of possible values are equally probable. It has two parameters - lower bound a and higher bound b - which represent bounds for possible values with the uniform distribution.

Its probability mass function is (also see Figure 2.3):

$$\begin{matrix} \frac{1}{n} & \text{for } a \leq k \leq b \\ 0 & \text{otherwise} \end{matrix}$$

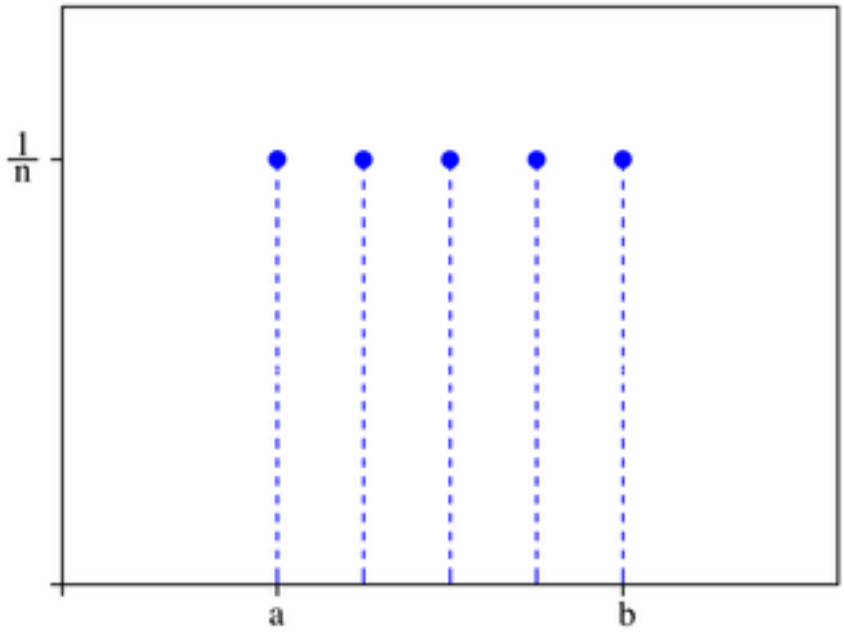


Figure 2.3: Example probability mass function of uniform distribution.

Its mean value is (a+b)/2 and its variance is $((b-a+1)^2-1)/12$.

Chapter 3

Related Works

In this chapter we analyse existing solutions in the field of XMLMS benchmarking. We briefly introduce several most-known benchmarks and compare them to better understand their disadvantages. Overcoming some of their drawbacks can be then understood as a main goal of this work.

3.1 XMark

XMark [Schmidt01] is a benchmark developed as a project on CWI which is the National Research Institute for Mathematics and Computer Science in Amsterdam. This institution is a part of the Stichting Mathematisch Centrum (SMC), a Dutch foundation for promotion of mathematics and computer science and their applications. Seven people contributed to this project.

XMark is a single-user application level benchmark. Testing was done on a prototype XMLMS called Monet XML [Schmidt01].

Its database model is based on an Internet auction site (see Figure 3.1). It contains a single XML document having a size up to 10 GB containing both textual and binary data. It is also the only benchmark which contains just one XML file as its data set. Main business entities are `item`, `person`, `open auction`, `close auction`, and `category`. The text data used are the 17000 most frequently occurring words of Shakespeare' s plays. The standard data size is 100MB and users can change the data size by ten times from an initial data. XMark own database generator is used to create the data set. There is no XML schema in the data set which would possibly provide some optimization opportunities for query optimizer [Yao02].

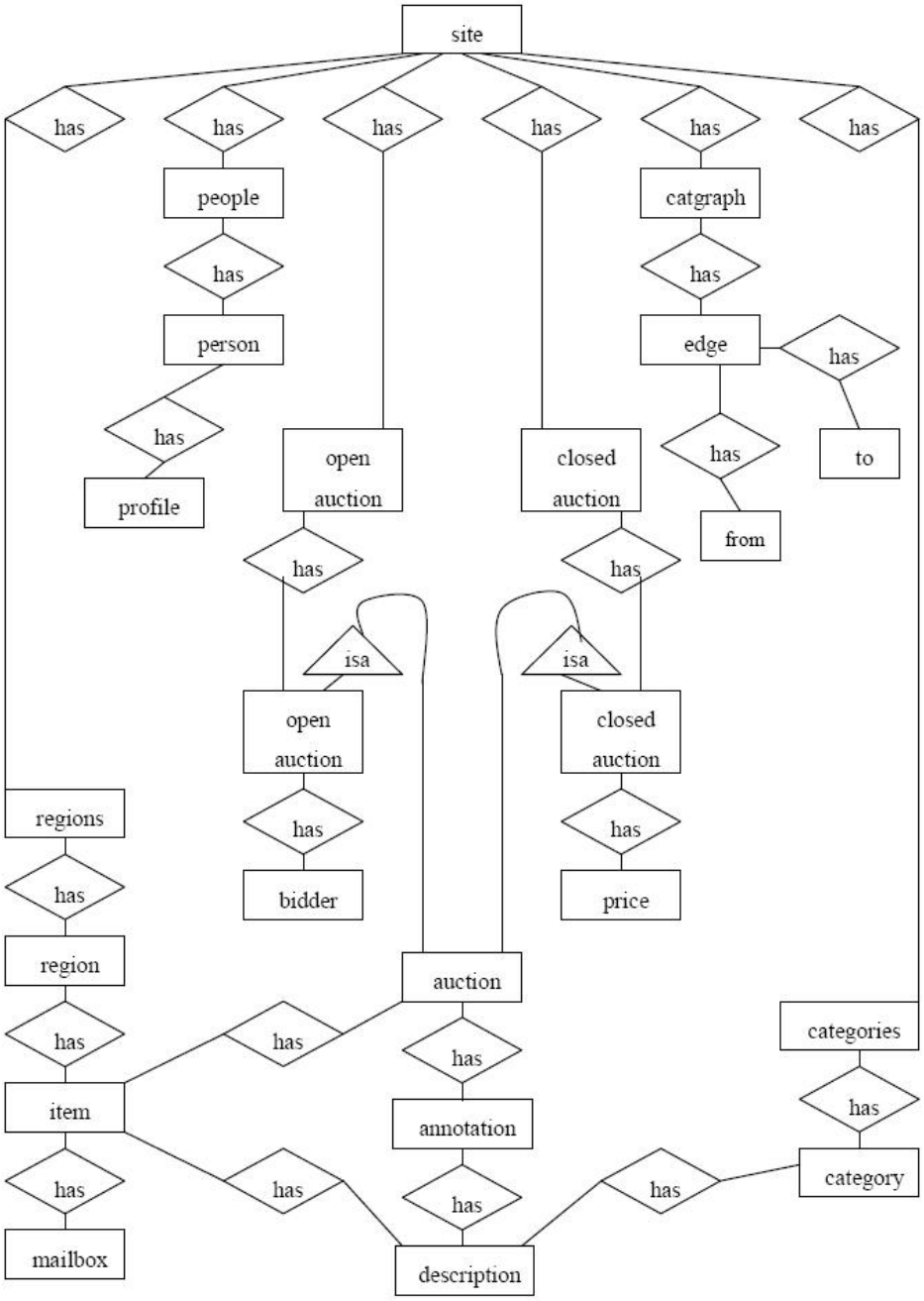


Figure 3.1: E-R schema of XMark's auction site [Bressan01].

XMark offers 20 XQuery queries covering basics of XML query processing. Provided queries can be divided into five groups according to functionality covered. The first group tests path query processing. The second one contains queries where XMLMS has to preserve the order of elements. The third one tests the performance in presence of IDREFS. The fourth one involves value based joins and the fifth involves aggregate functions, sorting, and merging of parts of documents [Bressan01]. See Table 3.1 and Table 3.2.

It is also remarkable that XMark appears to be the most exploited benchmark (i.e. the most cited one in the papers) among all standard ones (XMark, XBench, X007, MBench and XMach-1) [Afanasiev08].

Group	ID	Description	Comment
I	Q1	Return the name of the person with ID 'person0' registered in North America.	Checking ability to handle strings with a fully specified path.
	Q5	How many sold items cost more than 40.	Checks how well a DBMS performs since the XML model is document oriented. Checks for typing in XML.
	Q14	Return the names of all items whose description contains the word 'gold'.	Text search but narrowed by combining the query on content and structure.
II	Q2	Return the initial increases of all open auctions.	Evaluates cost of array lookups. Query is about order of data which relational systems lack.
	Q3	Return IDs of all open auctions whose current increase is at least twice as high as initial.	More complex evaluation of an array lookup.
	Q4	List reserves of those open auctions where a certain person issued bid before another person.	Querying tag values capturing document orientation of XML.
	Q11	For each person, list the number of items currently on sale whose price does not exceed 0.02% of person' s income.	Value based joins. Authors feel this query is a candidate for optimizations.
	Q12	For each richer-than-average person, list the number of items currently on sale whose price does not exceed 0.02% of the person' s income.	As above.
	Q17	Which persons do not have a homepage?	Determine processing quality in presence of optional parameters.

Table 3.1: Queries of XMark [Bressan01].

Group	ID	Description	Comment
III	Q6	How many items are listed on all continents?	Tests efficiency in handling path expressions.
	Q7	How many pieces of prose are in our database?	Query is answerable using cardinality of relations.
	Q8	List the names of persons and the number of items they bought.	Checks efficiency in processing IDREFs.
	Q9	List the names of persons and the names of items they bought in Europe.	As above.
	Q10	List all persons according to their interest use French markup in the result.	Grouping, restructuring and rewriting. Storage efficiency checked.
	Q13	List names of items registered in Australia along with their descriptions.	Tests ability of database to reconstruct portions of XML document.
	Q15	Print the keywords in emphasis in annotations of closed auctions.	Attempts to quantify completely specified paths. Query checks for existence of path.
	Q16	Return the IDs of those auctions that have one or more keywords in emphasis.	As above.
IV	Q18	Convert the currency of the reserve of all open auctions to another currency.	User defined functions checked.
	Q19	Give an alphabetically ordered list of all items along with their location.	Query uses ORDER BY.
	Q20	Group customers by their income and output the cardinality of each group.	The processor will have to identify that all the subparts differ only in values given to attribute and predicates used.

Table 3.2: Next queries of XMark [Bressan01].

3.2 XOO7

XOO7 Benchmark [Bressan03] is an XML version of database OO7 Benchmark [Carey94]. It has been used in four XMLMSs: XENA [Bressan03], LORE [McHugh97], Kweelt [Sahuguet01] and DOM-XPath [Bressan03]. It is another example of an application benchmark. Seven people participated in this project, mainly from the National University in Singapore.

Group	ID	Description
I	Q1	Exact match lookup. Generate 5 random numbers for AtomicPart's MyID. Return the AtomicParts according to the 5 numbers.
	Q4	Path lookup. Generate 5 random titles for Document. Return the Documents according to the 5 titles.
II	Q2	Select 1% of AtomicParts (with a buildDate after 1990).
	Q3	Select 10% of AtomicParts (with a buildDate after 1900).
	Q4	Select all AtomicParts.
III	Q5	Single-level "make". Find the CompositePart if it is more recent than the BaseAssembly it uses.
	Q6	Multi-level "make". Find the CompositePart (recursively) if it is more recent than the BaseAssembly or the ComplexAssembly it uses.
	Q8	Ad hoc join. Join AtomicPart and Document on the docId of AtomicPart and the MyID of Document.

Table 3.3: Queries of XOO7 [Bressan01].

Data set of XOO7 was taken from OO7 Benchmark and modified for XML purpose. The authors firstly took OO7's conceptual schema (an E-R diagram depicted in Figure 3.2) and transformed it into DTD. Then XOO7's own data generator was used to generate XML documents valid against the DTD. The user has three size options to choose (4MB - 1GB), so the data set is scalable. Generated XML files are complex and comprehensive in XML structures used (e.g. recursive elements are used).

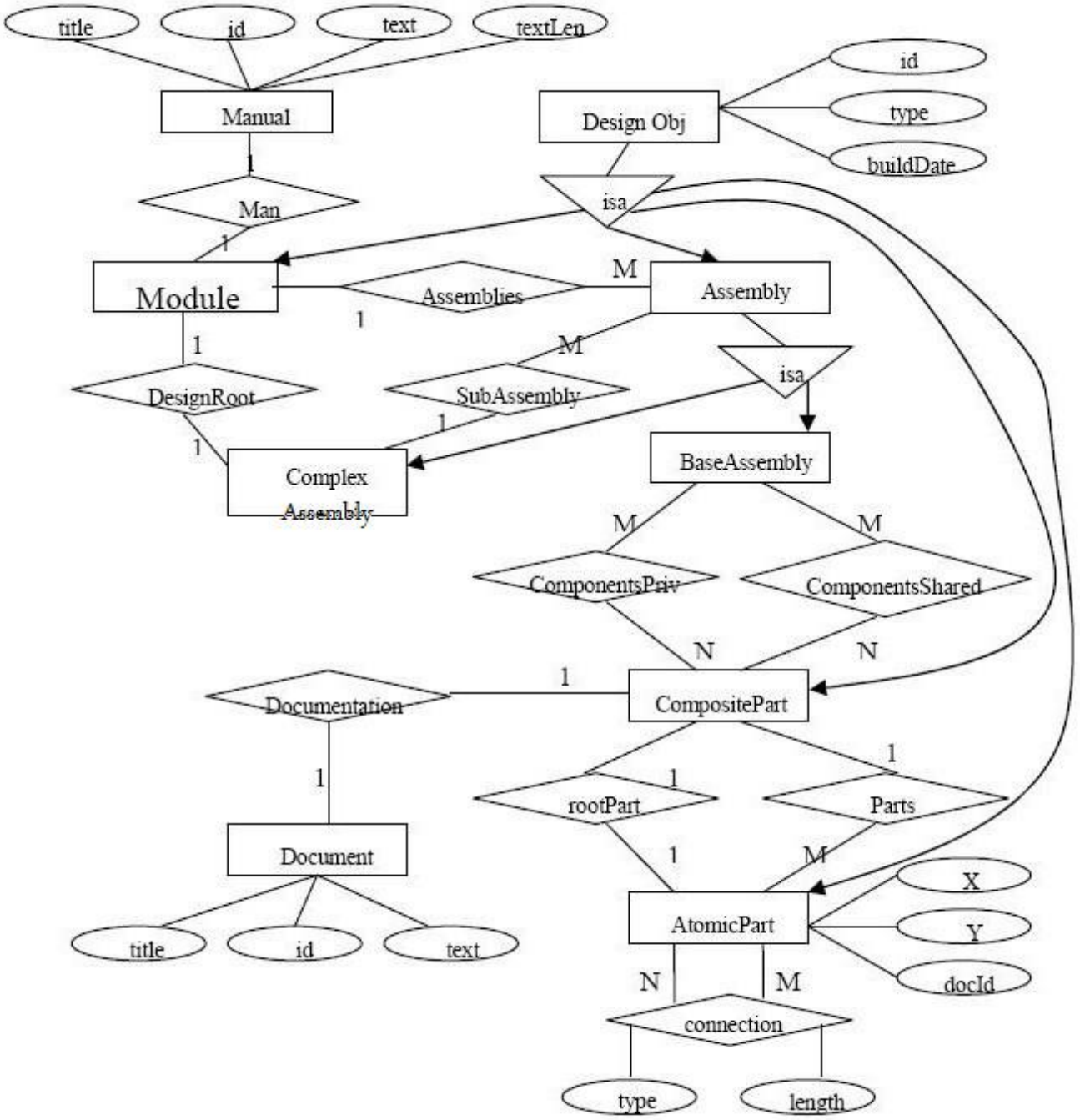


Figure 3.2: E-R schema of XOO7 [Bressan01].

Provided 22 XML queries can be split into three groups: standard database queries, navigational queries, and document queries. Queries (see Table 3.3) were once again converted from OO7 Benchmark's SQL queries to XQuery queries with some XML specific queries added. Original queries are outdated (syntactically incorrect) at the moment, but there is no problem rewriting them into the new syntax [Afanasiev08].

3.3 XMach-1

XMach-1 [Rahm02] is a bit special representative of application benchmarks because it is the only multiuser benchmark. It consists of a single database and multiple application servers and users. It was created by Database Group Leipzig of the Department of Computer Science of Universität Leipzig.

The benchmark is modeled for a web application that heavily uses XML data. It uses its own generator to create the data and eight parameters (number of XML documents per DTD, amounts for each type of element, etc.) of generated data can be specified. Its schema is depicted in Figure 3.3.

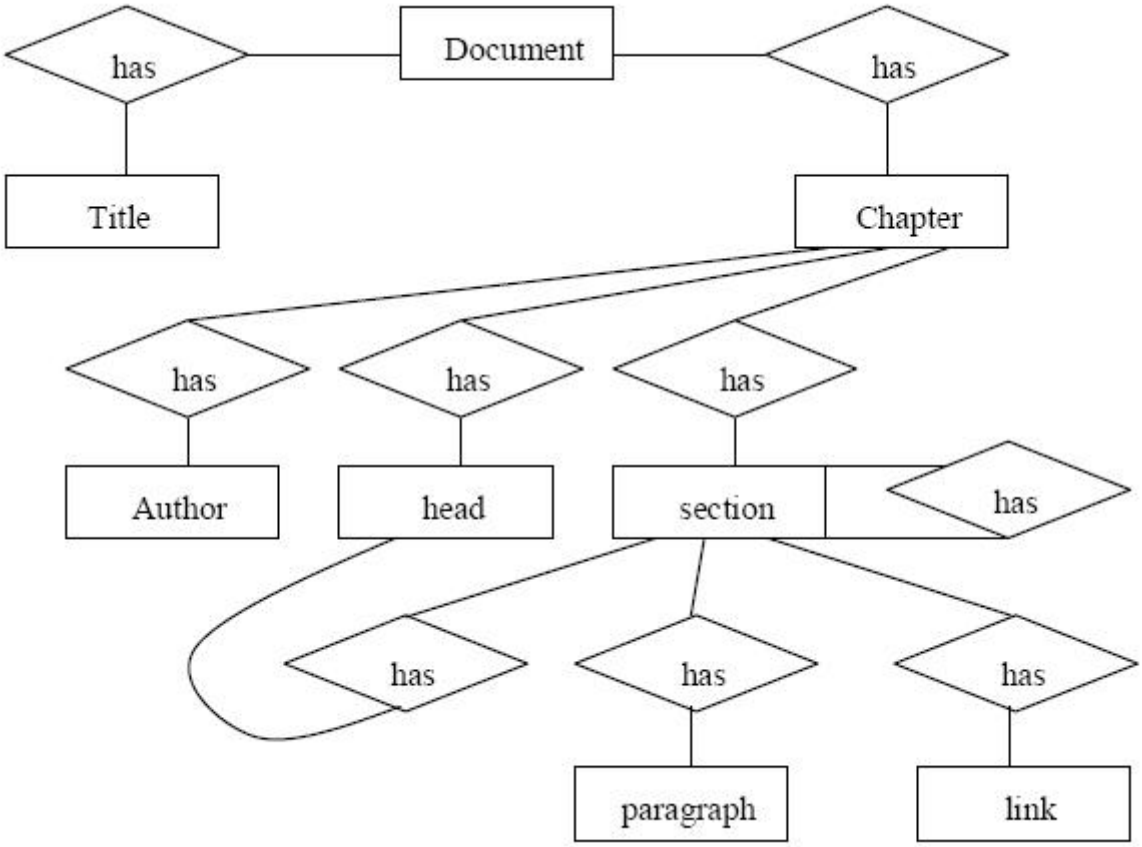


Figure 3.3: ER schema of XMach-1 [Bressan01].

Eight queries provided can be divided into four groups. The first group tests the management system using simple relational queries including basic comparisons. The second group tests document-centric data which must preserve the element order. The third group contains aggregate functions and metadata handling. Non-standard inserts and updates [Bressan01] are in the last group. See Table 3.4.

3.4 XBench

XBench [Yao02] is the newest of application benchmarks. It is a single-user benchmark applied on a single computer. XBench was created by two members of the Database Research Group in the University of Waterloo.

Group	ID	Description	Comment
I	Q1	Get document with given URL.	Returns a complete document (complex hierarchy with original ordering preserved).
	Q2	Get doc_id from documents containing a given phrase.	Text retrieval query. The phrase is chosen from the phrase list.
	Q5	Get doc_id and id of parent element of author element with given content.	Finds chapters of a given author. Query across all DTDs/text documents.
II	Q3	Return leaf in tree structure of a document given by doc_id following first child in each node starting with document root.	Simulates exploring a document with unknown structure (path traversal).
	Q4	Get document name (last path element in directory structure) from all documents that are below a given URL fragment.	Browses directory structure. Operation on structured unordered data.
	Q6	Get doc_id and insert date from documents having a given author (document attribute).	Join operation.
III	Q7	Get doc_id from documents that are referenced by at least four other documents.	Gets important documents. Needs some kind of group by and count operation.
	Q8	Get doc_id from the last 100 inserted documents having an author attribute.	Needs count, sort and join operations and accesses metadata.
IV	M1	Insert document with given URL.	The loader generates a document and URL and sends them to the HTTP server.
	M2	Delete a document with given doc_id.	A robot requests deletion, e.g. because the corresponding original document does no longer exist in the web.

Table 3.4: Queries of XMach-1 [Bressan01].

XBench is called the family of XML benchmarks, because it has more than one data set (unlike the other mentioned XML benchmarks). The group around XBench analyzed XML data and came to a conclusion that there are four rough types of XML applications and therefore four data sets are needed (see Table 3.5 for details). They test individually each of these types of databases: data-centric / single-document (i.e. containing one XML file only), data-centric / multi-document, document-centric / single-document, and document-centric / multi-document. Document-centric means that it must preserve the order of elements in each XML document (e.g. XHTML pages). Single-document means that the XMLMS contains a single XML document only. The authors use ToXgene [Barbosa02] - a generator of synthetic XML data - to generate specific type of data set.

	Document-Like (DL)	*Data-Centric (DC)*
Single-Document (SD)	GCIDE Dictionary, Oxford English Dictionary (OED)	TPC-W [Smith00] Catalog Data
Multi-Document (MD)	Reuters News Corpus, Springer-Valag Digital Library	TPC-W Transactional Data

Table 3.5: Data sets of XBench [Yao02].

Queries	**Functionality**	**DL or DC**
Q1	Top level exact match.	Both
Q2	Deep level exact match.	Both
Q3	Function application.	Both
Q4	Relative ordered access.	Both
Q5	Absolute ordered access.	Both
Q6	Existential quantifier.	Both
Q7	Universal quantifier.	Both
Q8	Regular path expressions (unknown element name).	Both
Q9	Regular path expressions (unknown subpaths).	Both
Q10	Sorting by string types.	Both
Q11	Sorting by non string types.	Both
Q12	Document structure preserving.	Both
Q13	Document structure transforming.	Both
Q14	Missing elements.	Both
Q15	Empty (null) values.	Both
Q16	Retrieve individual docs.	Both
Q17	Uni-gram search.	DL
Q18	N-gram search.	DL
Q19	References and joins.	Both
Q20	Datatype cast.	Both

Table 3.6: Queries of XBench. Borrowed from [Yao02].

Provided queries cover the whole XQuery Core [Boag07] similarly to XMark or X007 benchmarks. The

benchmark contains separate queries for each of the four types of data sets and the complete set has 67 queries (see Table 3.6).

3.5 Michigan Benchmark

Michigan benchmark (or MBench for short) is the only micro-benchmark among these (most known) benchmarks. It was created in the University of Michigan [Runapongsa02]. While the others test XMLMSs for performance in an application-like domain, this benchmark tests it for performance of doing basic tasks like simple selecting. It is meant to be used by developers of XMLMS to test different techniques of processing.

Its data set is similar to those in XMark and XOO7 in the sense that it has only one data set trying to comprehend all possibilities XML can offer. The data set is scalable in the similar meaning as in other benchmarks - size is adjustable. In Michigan Benchmark the depth of generated XML tree is constant and only the fan-out is adjusted according to the specified size.

Queries of Michigan benchmark are expressed in XPath because the XQuery was unstable in its beginnings. 57 XPath queries are provided, including bulk loads, inserts, deletes, and updates.

3.6 Comparison of Related Work

This chapter introduces a brief comparison of related benchmarks and discusses possible improvements. The reason to do this is to gain more insight into disadvantages of each one of the related benchmarks and then try to make up new ideas for the new benchmark. All next chapters will discuss this new benchmark.

The first obvious difference between Michigan Benchmark and other four ones is that MBench is a micro benchmark. That is why it's data generator is much more tunable than the other four.

A brief comparison of other four benchmarks is depicted in this Table 3.7:

The use of first two benchmarks is limited to the E-Commerce applications (because of their input data) while XOO7 is limited to library applications. An advantage of XMach-1 is that it is multi-user and networked what makes it more realistic. XBench seems to be the most advanced of them, however it lacks some features of XMach-1 (e.g. multiuser support, distributed environment). Its XML documents are the most diverse since it is the only one using four different types of data sources. It also is the only one using XSDs and multiple DTDs. XMach-1 is the only one of these benchmarks that applies updates on XMLMS as well.

According to the [Afanasiev08] 38% of standard benchmarks' queries are syntactically incorrect. This could have a major impact on how much a benchmark is used. XMark, the only benchmark from standard ones with all queries correct, is the one used most. According to [Afanasiev08], most of the benchmarks are not suitable for a thorough investigation of an XQuery engine. Many benchmark queries are similar to each other, using the same XQuery syntactical constructs. Moreover, only 10% of all queries use XQuery properties not present already in XPath 1.0 or XPath 2.0.

3.7 Disadvantages of Analyzed Benchmarks

We have observed these major disadvantages when working with related benchmarks (of XMLMSs):

- Limited application domains.
- Limited size and/or number of the documents.
- Fixed set of XMLMS queries.
- Queries do not correspond to the latest XQuery specification.

Next we will improve these disadvantages proposing a brand new benchmark.

	XMach-1	**XMark**	**XOO7**	**XBench**
Application domain	E-Commerce	E-Commerce	Library	Various
Users	Multi-user	Single-user	Single-user	Single-user
Document environment	Multiple documents	Single document	Multiple documents	Both
Scalability in no. of documents	From 10,000 to 10,000,000 documents	1	Unlimited	Various depending on domains
Scalability in size of documents	10 KB	From 10 MB to 10 GB	Unknown	Various depending on domains
DB physical distribution	Distributed	Centralized	Centralized	Centralized
XML schema	No	No	No	Yes
DTD	Multiple but same structure	One	One	Multiple
Query operations	Cover most of them	Cover most of them	Most of them	Complete XQuery
Update operations	Few	None	None	None
Multi-function queries	Yes	No Applicable	No	Yes
Syntactically incorrect XQuery queries	100%	0%	100%	34%
Number of papers	1	22	Unknown	5
XQuery specific queries (impossible in XPath)	25%	5%	5%	13%

Table 3.7: Comparison of application benchmarks [Yao02] , [Afanasiev08].

Chapter 4

Benchmark Specification

This chapter deals with functional and other requirements on XML benchmark from a user's perspective.

Firstly, we have observed a lack of freedom in generation of data in present benchmarks. So the data generator will be the key part of this work. Our aim is to support as much interesting XML data parameters as possible.

Another part can be schema generator. Today many XML documents have some sort of XML schema attached to them. Examples of such an XML schema are DTD [Bray06] or XSD [Fallside04]. This kind of generator should be obviously a part of data generator, that generates XML documents. The reason is that they correlate very tightly together. It is also possible to use an external tool that will e.g. take the generated XML documents and outputs XML schemes for them. It is reasonable to use such tools not only because they exist, but they are also verified and functional.

Thinking big, another main part can be generator of workload (i.e. XPath or XQuery queries). This generator will parse existing XML documents and produce queries for them.

From the bird's eye view what we are trying to do is a generator of XML benchmark data (with corresponding schemes) and the workload (queries).

It is interesting that the generation of benchmark can start with any of the three generators (data, schema or query generator). An overview of possible strategies can be seen in Table 4.1. The most of the existing works took an approach when schema is fixed, data is generated according to the schema and the queries are fixed (i.e. similarly to the second strategy from Table 4.1).

However, this work takes another approach - the XML documents are generated first, then the schemes and the queries on top of them (the first strategy in Table 4.1). It is a notable fact that we built our benchmark on the statistical data about the *real-world* XML databases from [Mlynkova06]. The most of these statistics cover properties of XML documents, not their schemes. We suppose that the reason is that not all XML documents even have their schema. Hence, when we were going to decide whether to generate XML documents on top of their schemes or vice versa, these statistics convinced us of generating XML documents first. This way we can ensure much more realistic generated data than the other way.

Next we will go through all kinds of parameters of the generators a user can specify and decide if we are going to implement them or not. Note that there can be some conflicts between the characteristics of XML data. There are many dependencies between them and a user may specify contradicting properties. Discussion on these conflicts between XML data characteristics (as generator parameters) will be part of this chapter too.

data → schema & queries	XML documents are generated first. Schema and queries are then generated on top of them. It is also possible to generate the queries on top of the schema (instead of the XML documents). Schema can be generated by an external tool as many capable ones already exist.
schema → data → queries	Another approach is to generate the schema first. XML documents are generated next (possibly by an external tool). Queries are then generated on top of the XML documents or the schema.
queries -> schema & data	The most eccentric strategy is to generate queries first (or let the user to input them) and then generate XML documents and their schemes on top of them.

Table 4.1: Three possible strategies when generating a benchmark.

One thing that these parameters have in common is that the generator never respects them totally accurately. For example: there is a chance that output files will have a little percentage of bytes higher (or lower) than user originally specified. However these deviations will have a minimal impact on quality of generated benchmark.

It is a notable fact as well that pre-defined sets of data should be prepared to simulate frequent kinds of XML data (e.g. document-centric XML documents, data-centric documents, etc.). There should be no mandatory parameter. This way the user will not be bothered from knowing exactly what every kind of parameter does.

4.1 Parameter Types

The data generator will support basic parameter types (e.g. strings, integers, floating point numbers, ...) as well as the advanced parameters - discrete statistical distributions. This kind of parameters is actually used in most cases. It is so because of a great amount of XML documents that can be generated by the data generator. Consider generating 1000 XML documents. If a typical passed parameter would be just one constant number, every generated XML document would have it and they would all have almost the same structure.

How a statistical distribution solves this problem? Imagine a number generator that takes any statistical distribution and produce its numbers as needed by other parts of a program. This number generator exists inside the benchmark generator and provides a collection of numbers that satisfy the needed distribution (passed as a program argument). This way the user will specify required distribution for 1000 files and let the program do the math. To better understand this feature, Example 4.1 should help.

Example 4.1 Passing distribution as a parameter.

The user specifies that she wants the depths of XML documents to have a binomial distribution with n=10 and p=0.5. Note how better this is than a constant depth for each of the 1000 documents.

```
java -jar generator.jar NumberOfGeneratedFiles=1000 DepthGen=binomial ↩
   (10,0.5)
```

Now the number generator accepts the distribution the user wants and will generate the XML documents with depths like 2, 3, 7, 3, 5, 5, 4, 5, 3, 5 and so on.

4.2 Basic Parameters

Let us start simple by saying that the generator will support no parameters from those mentioned in Chapter 2. This may reveal some unmentioned attributes that are actually needed. The first question is where to create XML data. There is no need for parameter here, the generator will just create them in the actual directory where it was started. However, a parameter for this would be more user-friendly.

More interesting question is what should be generated. Let us say the generator will only know how to generate a single file with a specified name. So the generator must have one implicit parameter - the name of the output file. Generating more files is then just cycling through list of file names. But there is a problem - how to generate thousands of XML files. According to the real data, most of XML databases consist of either one large XML document or many smaller XML documents. Hence it would be inhuman to require user input of, e.g. 1000 file names. So another solution is needed - the output files will have a generated name (partially specified by the user). For example: gen001.xml, gen002.xml, gen003.xml and so on. Hence, there is no need to specify file names, just the number of files.

4.3 Adding Structure to XML Document

Now, let us consider adding parameters to the XML data generator that will affect the overall shape of the XML tree.

All of these parameters describe the XML tree structure, so there is no surprise they are all related and influence each other. And hence we will not be able to fulfill all of these specified characteristics in the output XML data. For example if a distribution of levels of elements is given, it is hard to satisfy also user-specific depth of XML documents. Some conflicts, like this one, are obvious, some are less, and this chapter discusses most of them.

4.3.1 Size (or Number of Elements)

The key parameter of XML documents is their size in bytes. It is a very basic parameter telling us a very little about the real structure of an XML document, but it can be good enough for telling us roughly how complex the document is. It is even used as the only parameter in the most successful benchmark XMark.

The size can be influenced by:

- the amount of elements and attributes in the document,

- the length of element names,
- the length of attribute names,
- text contents, and
- attribute values.

An average length of element and attribute names and their values can be computed just by looking into the text generator input file (see Section 4.4.1). Percentage of text contents is a parameter which is also related to mixed-content part of the specification (see Section 4.4.2). These four parameters can be considered to be *sealed* by other parts of the benchmark generator. The only parameter left is the total amount of elements in the document which can be considered equivalent to the size in bytes of the XML document. So the user should be enabled to decide which of them will she specify - size in bytes or size in the amount of elements.

The size parameter also affects the fan-out parameter. When considering fan-out parameter, the generator must watch out for number of elements it generates. Also it will have to stop at certain level. So the overall depth is another parameter affected.

Specification of size for a single XML document is straightforward. If the user wants to generate thousands of them, a single size is not enough. So the final solution for specifying the size of XML documents is a statistical distribution. The user should be able to say, e.g. 'I want 1000 XML documents with sizes that have a uniform distribution from 500 to 10,000 bytes'.

4.3.2 Levels of Elements

Another characteristic of the output data is the distribution of levels. The user would want something like: 'a normal distribution of levels with a mean of 4 and variance of 3'. But this parameter is very restrictive. If one specifies the distribution of levels of elements, there is no place for any other parameter for structure of the XML tree. Everything is sealed because the whole XML tree is already specified. So specifying distribution of levels of elements would prevent us from fulfilling parameters like fan-out of elements and patterns like the DNA pattern.

4.3.3 Fan-out

Fan-out also determines the overall structure of the XML document. Users will probably want to define a statistical distribution of the fan-out for the elements again.

There are two conflicts when specifying this parameter. The depth and size are affected. Since the size is the most important parameter, the user will have to choose between specifying depth and fan-out. And since the distribution of fan-out is more precise than specifying the overall depth of XML document, specifying the depth will be transformed as a special case of specifying the distribution of fan-out.

Talking about specification of fan-out, it is not very convenient to have just one fan-out distribution for thousands of output XML documents. That way the mean value of fan-out will always stay the same and the output documents would look too similar. So it should be possible to specify a statistical distribution of the fan-out *average* value (at least). These two distributions will be combined by the data generator to get the desired output. See Example 4.2 for some more explanation.

Example 4.2 Example of specifying fan-out.

User says:

- I want the average value of fan-out to have a uniform distribution between 3 and 8.
- I want the fan-out to be similar to the binomial distribution with p=0.5 and n=10 in every generated document.

And for every element in every XML tree the data generator will:

- Generate next number of specified binomial distribution.
- Adds to it the difference between the mean value of the binomial distribution and the desired mean value (generated from the uniform distribution generator for every XML file).

4.3.4 Depth

When the user specifies the depth of an XML document, the generator can compute mean value of desired fan-out and continue to generate a balanced XML tree according to this fan-out. So specifying depth is not so precise specification of fan-out. These two parameters depend on each other, so it is hard and not very useful for a user to provide them both.

As with other parameters, specifying depth as a single number is not acceptable. When the user wants to generate more than one XML document, a single-valued depth may not be sufficient. So in reality the user will specify a statistical distribution of the depths she wants. From these depths the generator will compute the distribution of average fan-out. These two are obviously in the conflict but the user is still free to specify fan-out kind of distribution (see Section 4.3.3 for details). Just the average value of the fan-out is sealed by specifying the depth.

The only question left is how exactly will be the fan-out affected by specifying the depth. As we already know (from Section 2.2), the depth is a property of an XML document as a whole. This leads us not to the fan-out distribution inside the XML document, but just the distribution of average fan-out of all XML documents. Let D be generated depth for an XML document and N be generated amount of elements in it. We can compute desired average fan-out as the $\log_D N$. Example 4.3 shows how specifying depth is an alternative way to specify distribution of average fan-out.

Example 4.3 Example of specifying distribution of depth among XML documents.

User says:

- I want the distribution of depths in XML documents to be binomial distribution with p=0.2 and n=30 (average depth is 6).
- I want the uniform distribution of sizes of XML documents between 500 and 10 000 bytes (average size is 5250 bytes).

What the generator will actually understand is this:

- Given that average element has size of 20 bytes (computed from other generator parameters), the distribution of size of XML documents is a uniform distribution between 25 and 500 elements (so the average number of elements is 262.5).
- So the average value of average fan-out should be $\log_6 262.5 = 3.1$. That means that the required distribution of fan-out will be computed from distribution of depths and the distribution of size of XML documents. It will be $\log_{\mathrm{bin}(0.5,\ 30)}\mathrm{uni}(25, 500)$.

4.4 Mixed Content

The next parameters of XML data are the mixed-content related. Mixed content is a very frequent pattern occurring in almost every document-centric XML document. Including it in the generator is therefore almost mandatory.

It has a property of almost never being dependent on the shape of XML tree. The only exception are the simple elements, which will be implemented in another place of the data generator - as a pattern (see Section 4.6.3).

4.4.1 Generated Text

Generating mixed content is all about generating text. The markup part of mixed content will be generated by other parts of the data generator (see Section 4.3) so the only part remaining is the text.

For the purposes of our benchmark generator, the generated text does not have to be totally natural. Looking at the generated text from the point of view of a XMLMS (which is our target audience for parsing the text content) it is the same if sentences make sense or not. The data generator will try to make text as natural as possible. Text is being taken from external source at random places. This way it is possible that sentences will not start from their start (see Example 4.4), but generated text will be more random.

Example 4.4 Text generated by our data generator.

```
store when the events began? Probably Mr. Hug alone. although the robbers ↩
   might have been waiting for him, but if so, this would have probably ↩
   been stated. What did the porter say to the robbers? Nothing, because ↩
   the
```

The source words used as a text content are taken from an external file. It can even be user's own file with language specific characters (including diacritics) and so localized XML files can be expected as a result.

4.4.2 Percentage of Text in XML Document

As we can see from an analysis of XML data, markup has significant percentage in XML documents. The rate is approximately 50:50 to the text. That means that the portion of the text in XML documents is approximately as important as its structure.

This parameter has conflicts only with other mixed-content parameters such as percentage of simple and trivial elements. So there is no real problem when generating XML data according to it.

4.4.3 Percentage of Trivial Elements

Trivial elements are commonly reoccurring patterns in XML documents. But they are also components of simple elements. So they will be a part of a simple elements generation (see Section 4.4.4).

4.4.4 Percentage of Simple Elements

Simple elements can be also generated, however there are some difficulties in doing that.

Firstly, there is obviously a dependence on the amount of mixed-content elements. But the percentage of simple elements and the percentage total mixed-content elements are not conflicting parameters (rather complementary), so they can be implemented both.

More serious conflict is with trivial elements and the overall structure of an XML document. Trivial elements will not be implemented separately, so this is not the problem. But simple elements are very frequent, so we must be aware of them when implementing e.g. the fan-out parameter - they should not influence the desired average fan-out for an XML document. The final solution is implementing simple elements as patterns (see Section 4.6.3).

4.4.5 Average Depth of Mixed-content Elements

Another parameter taken into account can be the average depth of mixed-content elements. Elements with level near the average depth would have greater probability of being mixed-content. This has no influence on structure of the XML tree because it only adds a text to selected nodes.

The parameter affected by this (and every other mixed-content) parameter is the size of the whole XML document. The size of XML document will consist of markup content and mixed content. The generator will have to balance these two sizes so that the desired rate (e.g. 50:50) is preserved.

4.5 Recursive Parameters

Recursion is another main aspect of XML documents not so often taken into account. We will try to consider every possible type of it.

There is a minor or none conflict with both structure of XML tree and the mixed-content parameters, because recursion is all about the names of the elements.

4.5.1 Trivial Recursion

Trivial recursion is easily implemented in such branches of tree where every node has exactly one child. They will have to be implemented artificially as a pattern (similarly to DNA or relational patterns in Section 4.6).

4.5.2 Linear Recursion

Linear recursion is even more easily implemented than the trivial recursion since it has no constraints on structure of the XML tree. It is a subset of pure recursion. There are two things we have to care about: no more than one element name can occur in the recursion and it can occur just once in every level of the recursion.

4.5.3 Pure Recursion

Pure recursion is easily implementable too partially because it has also no constraints on the shape of an XML tree. The only thing to care about when implementing pure recursion is that just one element name can be used as a name of recursive element.

Pure recursion will have two parameters: chance to start pure recursion and probability that element inside pure recursion will be the recursive element (usually higher than first parameter).

4.5.4 General Recursion

Every other type of recursion is general. We can choose more than one recursive element and add them to recursion.

4.6 Patterns

Three attributes of XML documents have a form of a pattern. It is the DNA pattern, the relational pattern and the shallow relational pattern.

4.6.1 Generating DNA Pattern

According to its definition, the DNA pattern is an element that has one non-trivial non-recursive sub-element and multiple trivial ones.

DNA pattern has some quantitative parameters. For example number of trivial elements at each level of DNA is important. Number of DNA levels is important as well. These two parameters can be passed by a user (as a distribution).

Having its parameters given, trivial nodes can be created. After that, a non-trivial node is created and the algorithm continues with normal generation of XML subtree.

4.6.2 Generating Relational Patterns

According to the definition, a relational pattern is simpler than a DNA pattern. It is an element whose children are all trivial. It has an arbitrary number of attributes and no mixed content.

Generating such pattern requires 2 parameters - number of attributes and number of trivial children. Given that, the generation is simple - the program will generate trivial children.

Generating a shallow relational pattern is even simpler. It is just a constrained relational pattern, so there should be no problem at all. All trivial children elements will have the same name.

4.6.3 Generating Simple Elements

A simple element has also a form of a reoccurring pattern. It is an element with trivial child elements only. Its purpose in our version of the benchmark generator is to provide a specific amount of a simple elements' text content. The main reason to implement them as a pattern (instead of normal elements of tree structure) is because they contain one of the biggest part of the text content in document-centric XML documents.

4.7 Schema Generators

According to statistics [Mlynkova06], schemes for XML documents are used quite often. As long as some XMLMSs can use these schemes as an information how to create the internal representation of XML documents (e.g. tables), it is interesting to create schemes at least for some of the documents.

It is a notable fact that there are two possible strategies on how to generate data. XML documents can be generated first and schemes will be generated on top of them, or vice versa. Most of existing benchmarks generate XML documents from one sealed XML scheme. However, this benchmark takes a different approach and generates schemes from XML documents. The reason for this is because we have much more statistics about the XML documents than about the schemes. If the documents are generated realistically, the schemes should be realistic as well.

4.7.1 DTD Generator

The most useful parameter a user can specify for a DTD generator is a percentage of XML documents to create a DTD for. There are massive differences in each of the categories of XML documents when looking at this percentage. For instance 93.7% of document-centric XML documents have their DTD, but none of the semantic web documents have one [Mlynkova06].

Other parameters of DTDs (e.g. fan-in) are hard to implement when XML documents are already given. As long as contents of XML documents is generated realistically and a DTD is generated according to their tree structure, there should not be any problem in authenticity of the DTD itself. So DTD generator will be implemented on top of generated data and will reflect them (not vice versa).

Final solution for our XML benchmark will be an external DTD generator, which infers schemes from the generated XML documents.

4.7.2 XSD Generator

XSD generator is principally the same as DTD generator and there will be no implementation of its advanced parameters. It will reflect already generated XML documents.

Note that an external tool can be used to generate XSD (or any other schema) similarly to the generation of DTD schemes.

4.8 Query Generator

Query generator is the next important component of this XML benchmark (after XML data generator). It is responsible for generating queries according to the pre-generated XML data.

There is a useful table in [Afanasiev08] dividing other benchmarks' queries into categories - see Table 4.2. There are many similar tables in white papers about each of the benchmarks but this one should be the most *unbiased* as Mr. Afanasiev just compared the benchmarks and did not propagate his own.

Category	Michigan	XMark	XOO7	XMach	XBench
Core XPath	12	3	1	0	1
XPath 1.0	4	3	8	3	12
Navigational XPath 2.0	22	5	6	1	22
XPath 2.0	5	8	6	2	23
Sorting	1	1	1	1	9
Recursive functions	2	0	0	1	0
Intermediate results	0	0	0	0	0

Table 4.2: Categories of queries of other benchmarks.

Next we will go through all the categories from Table 4.2, briefly define them and discuss generating some queries in each one of them. Note that extending the query generator by other examples of queries in each category was taken into account when designing the generator. Generator of queries uses XQuery templates with empty parts filled by XML document name and important element names. Therefore it is very easy to add new query types to the generator. See Example 4.5 for an illustration how templates work.

Example 4.5 Using templates for query generation.

Following code is a template for sorting XQuery generic query:

```
for $a in %s//%s order by $a return <result>{$a}</result>
```

First "%s" (which denotes a substitution of a string) will be replaced by the document that is queries and second "%s" will be substituted by the name of the interesting element.

4.8.1 Tightly coupled Data Generator and Query Generator

Our implementation of benchmark generator tightly couples data and query generators. The reason for doing this is because the query generator should reflect user's intentions (which are represented mainly in data generator parameters) also in the generated queries. For example if the user tends to generate many mixed-content elements, the query generator can guess that she would want to generate queries concerning it as well. For this reason, the data generator outputs most usable element names for queries in many categories (most common element, mostly used element in recursion, mostly used mixed-content element and mostly used trivial element). The user can express her intention by choosing from the four categories and the query generator will generate the queries according to the user's choice (see Example 4.6).

Example 4.6 Specifying user's interest.

A user can choose from 4 categories. In this case she has chosen mixed content as her main interest:

```
java -jar generator.jar InterestingElement=mixed-content ...
```

And the generator will generate all queries (except exact match queries) about an element that occurs mostly as a mixed-content one.

4.8.2 Core XPath

These queries test XMLMS performance when finding an XML element(s) specified by the navigational part of the XPath. Excluded are the use of position information, all functions and the general comparisons.

The XBench distinguishes between two special cases (see Section 3.4) - the queried element on the first level and a nested element (see Example 4.7). From the point of view of the query generator it is good to let the user decide how deep the queried element should be. The user will determine the number of the queries and statistical distribution of levels of their queried elements.

Example 4.7 An example of an exact match query with a deeper element.

Following XQuery query returns all found elements with specified path.

```
for $th in doc("input.xml")/held/inches/able/held
return
    $th
```

Another parameter (besides the level of the queried element) that a user can specify is of course the number of generated queries. This can be specified also by a probability that an element (at required level) is queried.

4.8.3 Text Queries

Preserving the order of a text is one of the most important attributes of XMLMSs. The text queries test an XMLDB for the performance when searching for a text. They also belong to the category of Core XPath. A user of the query generator can specify the number of these queries.

Example 4.8 An example of a text query.

This example queries a document for elements containing the word 'key' somewhere in their contents.

```
for $a in doc("input.xml")//going
where contains ($a, "key")
return
 <result> {$a} </result>
```

4.8.4 XPath 1.0

When storing a document-centric XML document, an XMLMS must store the elements in their original order (otherwise the meaning of the text would be lost). That is why there exist ways to query an XMLMS for the ordered access.

There are two possibilities when querying for order:

- Absolute order - these queries retrieve elements according to their absolute position in the XML tree. Examples are 'give me the first element with property X and Y'.
- Relative order - queries that return elements according to their neighboring elements in the XML tree. See Example 4.9.

Example 4.9 An example of a relative access query.

Following query returns next element to every element named 'until'.

```
for $a in doc("input.xml")//until,
    $p in doc("input.xml")//until[. >> $a][1]
return
    <result>$p</result>
```

A user of the query generator can specify the number of an absolute and relative order access queries. Besides these obvious parameters, more customizable absolute ordered queries could be useful. User can specify what index should be used in absolute ordered queries.

4.8.5 Navigational XPath 2.0

From XPath 2.0 are excluded the use of position information and all aggregation and arithmetic functions. We chose to include queries with 'some' and 'every' clause that are also part of the XQuery. They are used to construct a quantified expressions. An example of generated query of such kind can be seen in Example 4.10. The user is able to specify the amount of this kind of queries. 'Some' and 'every' versions of a query are generated randomly in rate 50:50.

Example 4.10 An example of generated quantified expression.

Following query returns true if every 'appeased' element contains the word 'the' in its text content.

```
every $appeased in doc("input.xml")//appeased
satisfies contains ($appeased, "the")
```

4.8.6 XPath 2.0

An instance of this category can be aggregation function queries, which apply an aggregation function on the data. Examples of such functions are a sum of some set of numbers, an average value, etc. The aggregation function queries suppose that there exist many similar elements, which can be used as a source for these aggregate functions. The practical usage of an aggregate function in XQuery can be seen in Example 4.11.

Example 4.11 An example of a query with an aggregate function.
This query computes the count of the elements with name 'beat' in the 'input.xml' XML document.

```
count(doc("input.xml")//beat)
```

The user of query generator should be able to specify the number of particular aggregate function queries.

4.8.7 Sorting

These queries use sorting on some of the generated attributes to get an ordered list of elements. A user of query generator can specify the number of generated queries of this kind.

Example 4.12 An example of a query using sorting.
This query uses a sorting on elements content to sort the selected result alphabetically.

```
for $a in doc("input.xml")//elem
order by $a
return
    <result>{$a}</result>
```

4.8.8 Recursive Functions

XQuery enables a user to define her own functions, possibly recursive. We have chosen a simple recursive function so generic that it can be used on any XML document. It computes a depth of an XML document. Example of generated query can be seen in Example 4.13. The user can specify the percentage of generated XML documents that will be queried.

Example 4.13 Generated usage of a recursive function.
Following recursive function returns a depth of an XML document.

```
declare function local:depth ( $root as node()? ) as xs:integer?
{
  if ($root/*)
  then max($root/*/local:depth(.)) + 1
  else 1
};

local:depth(doc("input.xml"))
```

4.8.9 Intermediate Results

Intermediate result queries contain 'let' clause of XQuery (e.g. see Example 4.14). User can specify percentage of queries XML documents.

Example 4.14 An example of intermediate result query.

Following example returns the first 'mr' element from `input.xml` and also count of its child nodes.

```
let $x := (doc("input.xml")//mr)[1]
return
  <result>
    {$x}
    <count>{count($x)}</count>
  </result>
```

4.8.10 More complex Queries with Joins

One might argue, why more complex queries like ones with joins are missing in the benchmark generator. The reason for this is that generated data is too artificial - there is no real connection between XML files as it is between database 'tables'. It was not our intention to provide such an aspect of relational data represented in XML and we rather concentrated on existing analysis of XML databases and its statistics. We simply chose the first strategy in Table 4.1 and for joins in queries the second strategy is much more suitable.

4.9 Summary

This section provides information about all the parameters of generators considered in Table 4.4.

A comparison of our generated queries with queries of other benchmarks can be seen in Table 4.3. Queries are divided into categories according to the set of XQuery that they use.

Category	Other 5 benchmarks combined	Benchmark Generator
Core XPath	17	arbitrary
XPath 1.0	30	arbitrary
Navigational XPath 2.0	56	arbitrary
XPath 2.0	44	arbitrary
Sorting	13	arbitrary
Recursive functions	3	arbitrary
Intermediate results	0	arbitrary

Table 4.3: Comparison of queries with other benchmarks.

Parameter name	Conflicts with	Will be implemented	Type
output directory	none	yes	constant
number of documents	none	yes	constant
size (in bytes)	number of elements, fan-out, depth, text content, attributes	yes	statistical distribution
number of attributes	size, size of text content	yes	statistical distribution
levels of elements	size, number of elements, fan-out, patterns	no	
fan-out	depth, size	yes	statistical distribution
depth	fan-out, size	yes (supplementary to the fan-out)	statistical distribution
percentage of text	size	yes	constant
simple mixed-content elements	percentage of text, patterns	yes	constant
average depth of mixed-content	none	yes	statistical distribution
percentage of DTDs	none	yes	constant
percentage of XSDs	none	yes	constant
probability of relational pattern	none	yes	constant
probability of pure recursion	other recursions	yes	constant
probability of purely recursive element	none	yes	constant
probability of trivial recursion	other recursions	no	constant
probability of linear recursion	other recursions	yes	constant
probability of general recursion	other recursions	no	constant
exact match query frequency	none	yes	constant
exact match levels	none	yes	statistical distribution
aggregate queries	none	yes	constant
sorting queries	none	yes	constant
text queries	none	yes	constant
absolute order queries	none	yes	constant
relative order queries	none	yes	constant
quantification queries	none	yes	constant
recursive function query per file	none	yes	constant
intermediate result query per file	none	yes	constant
interesting element for queries	none	yes	constant

Table 4.4: Parameters of the benchmark generator and their relations.

Chapter 5

Implementation Details

5.1 Architecture Overview

There are couple of models whose purpose is to describe the architecture of a software system. One of the oldest is 4+1 views model [Kruchten95]. It contains of 4+1 views on developing system, what serves as an analogy to real-world architecture blueprints. For our purposes the Logical view and the Process view should be enough to see the whole architecture.

In addiction a programming language is very important decision to be made. This benchmark will be programmed in Java for portability purposes.

5.1.1 Logical View

Logical view is user's or functional view on the system. It presents domain of the system (names of entities used in this domain area). As domain of this XML benchmark is very simple, no complicated figure is needed. Domain of XML benchmark involves XML, DTD, XSD and XQuery files, which are the output of XML benchmark generators. Already defined parameters are also part of it. Figure 5.1 shows how files and parameters fit together in our XML benchmark generator (some of the defined parameters are displayed as attributes of generators):

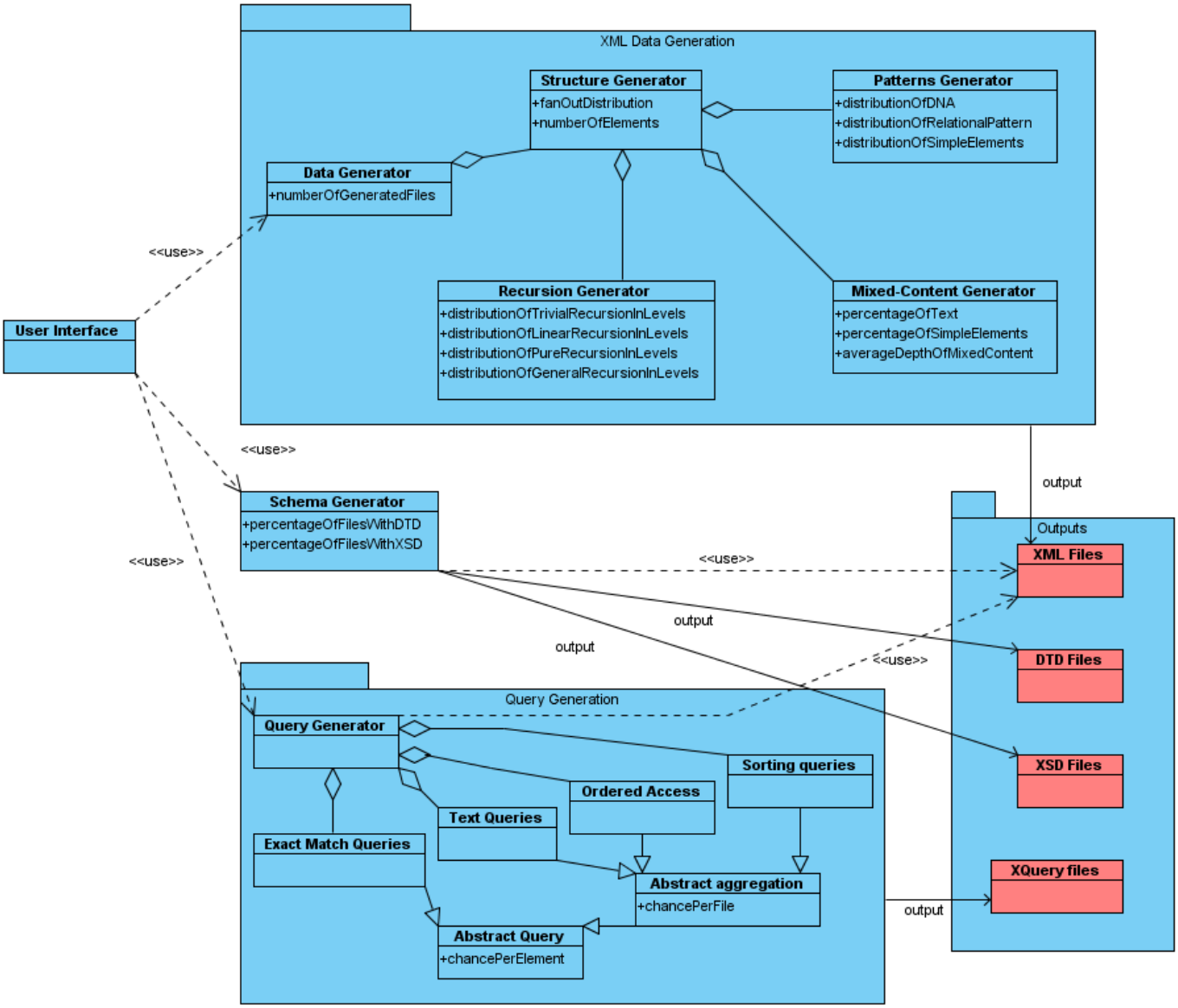


Figure 5.1: Simplified domain diagram.

5.1.2 Process View

This view's purpose is to show the process of generating whole benchmark data.

The process could be made relatively straightforward. Firstly, the generator generates XML documents (according to the given parameters). Then, according to XML documents, XML schemes are generated. And, finally, the queries for XMLMSs are generated as well.

Better would be if the generation of each XML file and its schema would be parallel (for performance reasons) as shown in Figure 5.2. This way the generators are not forced to wait for the slower hardware (a hard disk) to process given instructions and other computations could take the remaining processor time. We suppose generation of thousands of XML files (and corresponding schemes), so the performance gain should be significant.

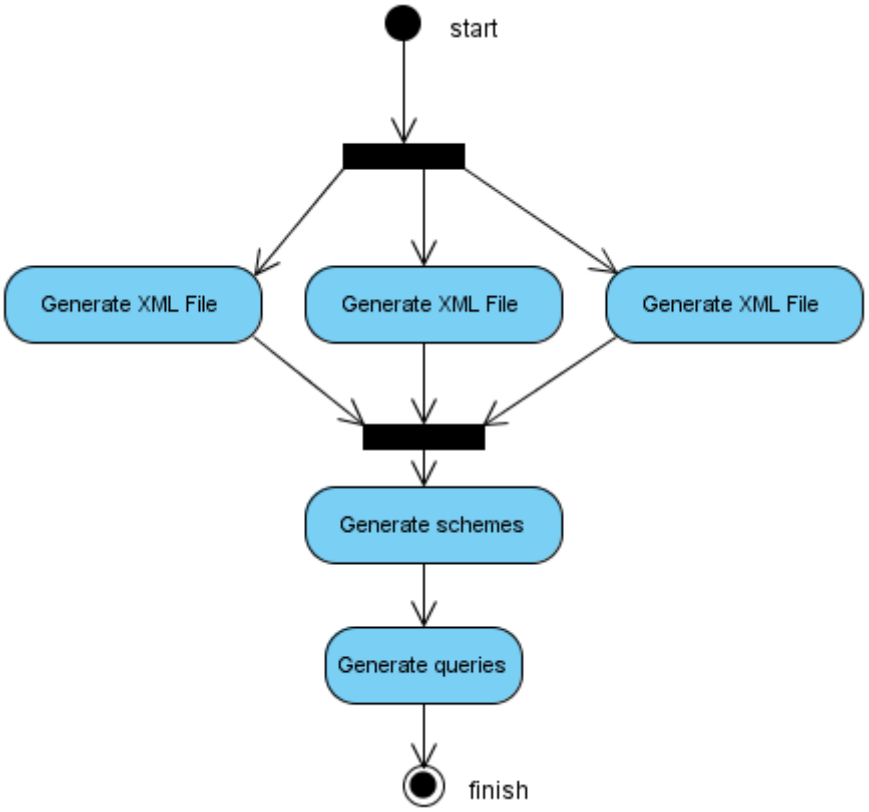


Figure 5.2: Parallelized process of generation XML data.

Another concern of process view are non-functional requirements, which are shown in Table 5.1.

5.2 Algorithm of Data Generation

Process of generating XML documents is crucial for benchmark generator. The core of the process is a recursive function. Its purpose is to simulate a path in an XML tree and create the nodes. Following Example 5.1 shows the function.

Requirement	How system will stand
Performance	DOM should not be used at all for memory concerns.
Reusability	Both data generator and query generator can be used in other areas as well. Query generator could be slightly modified to be independent on the data generator.
Extensibility	Data generator is extensible by adding additional parameters - statistics about generated data. Query generator can be easily extended by adding new templates of XQuery queries to it. Templates for automatic benchmark testers (i.e. XCheck [Franceschet06]) can be added as well.
Testability	It is relatively easy to check if passed parameters were met in generated XML documents.
Scalability	Generated benchmark is not designed to be used in a distributed network. However, the generator can be run on multiple computers simultaneously (each one running instance producing independent data).
User interface	Program will probably support command-line only interface, but a GUI could be easily added.

Table 5.1: Non-functional requirements on benchmark generator.

Example 5.1 Example of code generating the XML tree structure.

```
private void generate(final int level) {
  // Return-from-recursion check.
  if (level == depth) {
   xmlGenerator.writeLeafElement();
   return;
  }

  // Pattern instead of regular element.
  if (patternGenerator.generatePattern(level, depth, fanoutGen)) {
   return;
  }

  // Element's tag.
  final boolean isMixedContent =
   mixedContentGenerator.isElementMixedContent(level, false);
  xmlGenerator.writeOpeningElement(isMixedContent);

  // Children (with text content or not).
  final int fanout = fanoutGen.nextInt();
  for (int i = 0; i < fanout; ++i) {
   writeTextContent(isMixedContent);
   generate(level + 1);
  }
  writeTextContent(isMixedContent);

  // Ending tag.
  xmlGenerator.writeClosingElement();
 }
```

Generation of patterns obviously takes place before everything else and patterns generator is careful not to impact a desired shape of an XML tree. After that the elements are created in a usual way: opening of element, recursively child elements, text content (if any), and closing the element.

Schema is generated by a slightly modified third party generator for each file. It is also possible to use other third party schema generators which can use groups of XML documents to generate a schema.

Query generator uses statistics provided by the data generator to determine which element of each XML document it will query. An exception is a call to exact match query generator which is done between the generation of each element because there can be more reasonable queries on one file in this case.

5.3 Pre-defined Settings of Parameters

This section deals with default settings built in the generator. Default settings for an XML generator are important, because there is a finite number of XML applications. These applications can be divided to a small number of groups. It is more user-friendly when a generator has some parameters pre-set, so the user does not have to set all of them.

As stated in [Mlynkova06], XML data can be divided into these six classes:

- data-centric documents, i.e. documents designed for database processing (e.g. database exports, lists of employees, lists of IMDb movies and actors etc.),
- document-centric documents, i.e. documents which were designed for human reading (e.g. Shakespeare's plays, XHTML documents, novels in XML, DocBook documents etc.),
- documents for data exchange, e.g. medical information on patients and illnesses etc.,
- reports, i.e. overviews or summaries of data (usually of database type),
- research documents, i.e. documents which contain special (scientific or technical) structures (e.g. protein sequences, DNA/RNA structures etc.), and
- semantic web documents, i.e. RDF documents.

Table 5.2 extracts all important statistics from [Mlynkova06] for each one of the six categories.

A user can use these pre-defined categories of benchmarks through a command line (see Section 5.4). For instance, specifying:

```
java -jar generator.jar -data-exchange
```

...is equivalent to specifying all the parameters like:

```
java -jar generator.jar NumberOfGeneratedFiles=9 PercentageOfTextGen= ↩
   uniform(31,40) PercentageOfFilesWithDTD=100 ...
```

5.4 User interface

A user interface to such thing as a benchmark generator consists of interface to pass the parameters of a generated benchmark.

All three parts of the benchmark can be made as command-line utilities. Passing a parameter to a command-line program consists of passing it as a program argument. This is not considered very user-friendly, but user-friendly-ness is not a part of this work. Example of command-line approach can be seen in Example 5.2.

Example 5.2 Passing parameters through command-line arguments.

```
java -jar generator.jar NumberOfGeneratedFiles=10 PercentageOfTextGen= ↩
   uniform(50,50) NumberOfElementsGen=uniform(50,200) ...
```

However two other user-friendly approaches that can be implemented in the future.

The user might want to specify the arguments in the form of a configuration XML file which she is used to work with. Instead of running three command-line utilities on three configuration XML files, all the information can be stored in just one XML file and processed by another simple utility that will translate it to three command-line calls. This configuration through an XML file could be even more friendly if it would support more calls to the generators. The user can then specify e.g. groups of XML data or queries which

Category	Data-centric documents	Document-centric documents	Documents for data exchange	Reports	Research documents	Semantic web documents
Number of XML documents	89	305	9	1491	153	26
Average size of document (kB)	672	182	1744	3903	709	5116
Sample variation of size (MB)	510.66	0.54	154.40	61.84	352.32	5534.50
Documents with DTD (%)	99.7	93.7	100	0	99.8	0
Documents with XSD (%)	0	57.8	0	100	99.6	0
Average depth of document	5	7	5	5	5	5
Attribute exploitation (%)	31.7	96.2	92.2	100.0	99.9	99.9
Mixed elements (%)	0.2	76.5	8.7	0.0	10.1	2.4
Simple mixed content (%)	55.9	79.4	99.6	0	2	3
Percentage of text	43.8	81.6	36.3	6.2	33.1	54.9
Linear recursion (%)	0.06	19.92	32.57	0	0.65	2.52
Pure recursion (%)	0.03	18.76	22.48	0	0	1.46
DNA pattern elements (%)	1.19	10.66	8.08	0.00	8.64	0.61
Relational pattern elements (%)	29.23	6.23	29.53	94.29	22.66	41.56

Table 5.2: Pre-defined settings for application categories.

have different properties. An example of this approach (similar to MemBeR project of [Afanasiev05]) can be seen in Example 5.3.

Example 5.3 Passing parameters through a configuration XML file.

This example shows how a user can specify two calls to data generator (data source can be divided into two distinct categories) and then schema and query generator calls.

```
<configuration>
  <dataGenerator preSetRule="data-centric" numberOfGeneratedFiles="10" />
  <dataGenerator preSetRule="document-centric" NumberOfGeneratedFiles ↩
     ="20"/>
  <schemaGenerator />
  <queryGenerator />
</configuration>
```

On top of the proposed XML configuration file and command-line utilities a GUI application can be built. It would have many property pages and one 'generate' button. However, there will probably be no graphical user interface when operating with XML generators. It is not because GUI is hard to implement, but it is out of the scope of this work and there is little added value in it.

Chapter 6

Application of the Benchmark

This chapter is a practical counterpart to the rest of this work. It includes a brief information about the current status of XMLMSs, provides a how-to tutorial on applying our benchmark generator and displays some interesting outcomes of the benchmark.

6.1 Overview of Current XMLMSs

This section contains an overview about the present situation in XML DB world - what XMLMSs exists and which of them will be benchmarked by this benchmark. Table 6.1 shows representative XMLMSs.

Because of our limited technical equipment, only a few of these XMLMSs will be benchmarked by the output of our benchmark generator - Qexo, Qizx, Saxon and partially eXist. The main reason is that the most-known XMLMSs have very similar performance, so we have chosen less known and less robust ones to better show their differences.

6.2 Tutorial for our Benchmark Generator

This section contains a simple tutorial how to use our benchmark generator to benchmark an arbitrary XMLMS.

The overall process of generating and applying a benchmark can be divided into these simple steps:

1. Install to-be-tested XMLMS.
2. Generate the benchmark (data, schema and queries).
3. Run installed XMLMS.
4. Load the input data into the database.
5. One by one, test the queries in the XMLMS.
6. Write down the running times for each query.

Name of XMLMS	Description
eXist [Chaudhri03]	An open-source native XML database built on XML technology. It is currently the most popular XMLMS.
MonetDB/XQuery [Boncz06]	An open-source high-performance native XMLMS
Galax [Fernandez04]	An open-source native XMLMS. It implements most of the XQuery specification.
Berkeley DB XML [Chaudhri03]	Oracle Berkeley DB XML is an open source, embeddable XML database, that adds a document parser, XML indexer and XQuery engine on top of Oracle Berkeley DB.
X-Hive [Webster06]	X-Hive/DB is a high-performance, scalable, native XML database that supports all of the major XML standards; offers a Java programming interface; and provides methods for storing, querying, and publishing XML data.
Microsoft SQL Server 2005 [Vithanala05]	XML-Enabled database with limited support for XQuery (through 'xml' column type).
Oracle [Oracle08]	Similarly to Microsoft SQL Server, contains XMLType which can store XML documents. It is another representative of XML-Enabled databases.
DB2 [Saracco06]	Another XML-Enabled relational database from IBM.

Table 6.1: Overview of current XMLMSs.

We start by installing the tested XMLMS, which in this case will be a native XMLMS, eXist [Chaudhri03]. It is multi-platform (written in Java), so no special requirements on operating system are made. Installation process is fairly simple, just follow the steps in the installer.

We continue by generating a benchmark using the benchmark generator made in this work. We used our predefined reports parameters command-line arguments (the actual values are not important in this tutorial):

```
java -jar generator.jar -reports
```

This will generate XML documents and XML schemes in the directory `./generated` and XQuery queries in `./generated/queries`.

The next step is to run installed XMLMS. In this particular case (we use Windows Vista and eXist XMLMS), running 'eXist Database Startup' from the Start menu should be enough. If everything went OK, we should see something like Figure 6.1.

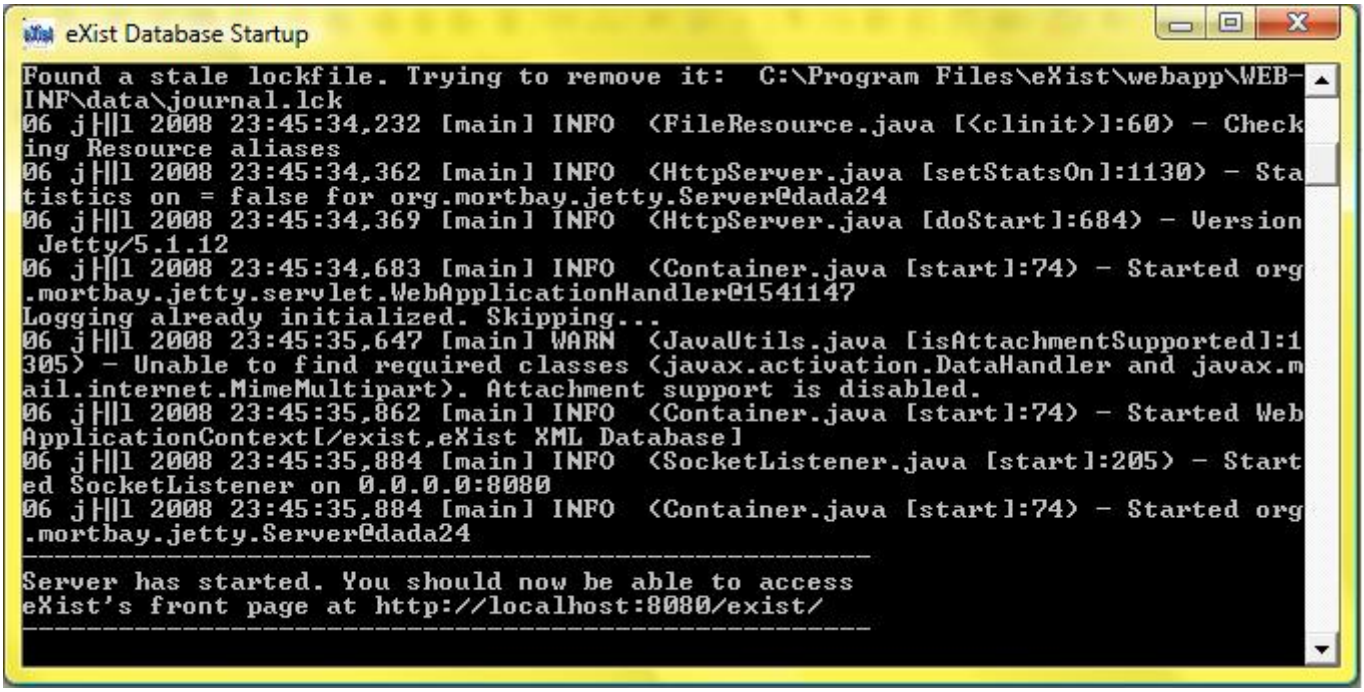


Figure 6.1: eXist startup.

As suggested from the 'eXist Database Startup' window, we should now be able to access the XMLMS from its start page, `http://localhost:8080/exist/`. As we can see, this particular XMLMS has web user interface, so no special client application is needed. We continue by looking into the `./generated/queries` directory and copying the content of each XQuery file into 'XQuery Sandbox' of eXist, which can be found on the address `http://localhost:8080/exist/sandbox/sandbox.xql`. Then by clicking on the 'Send' button, we not only get the result, but eXist also provides us with the time of execution which is also visible (see Figure 6.2).

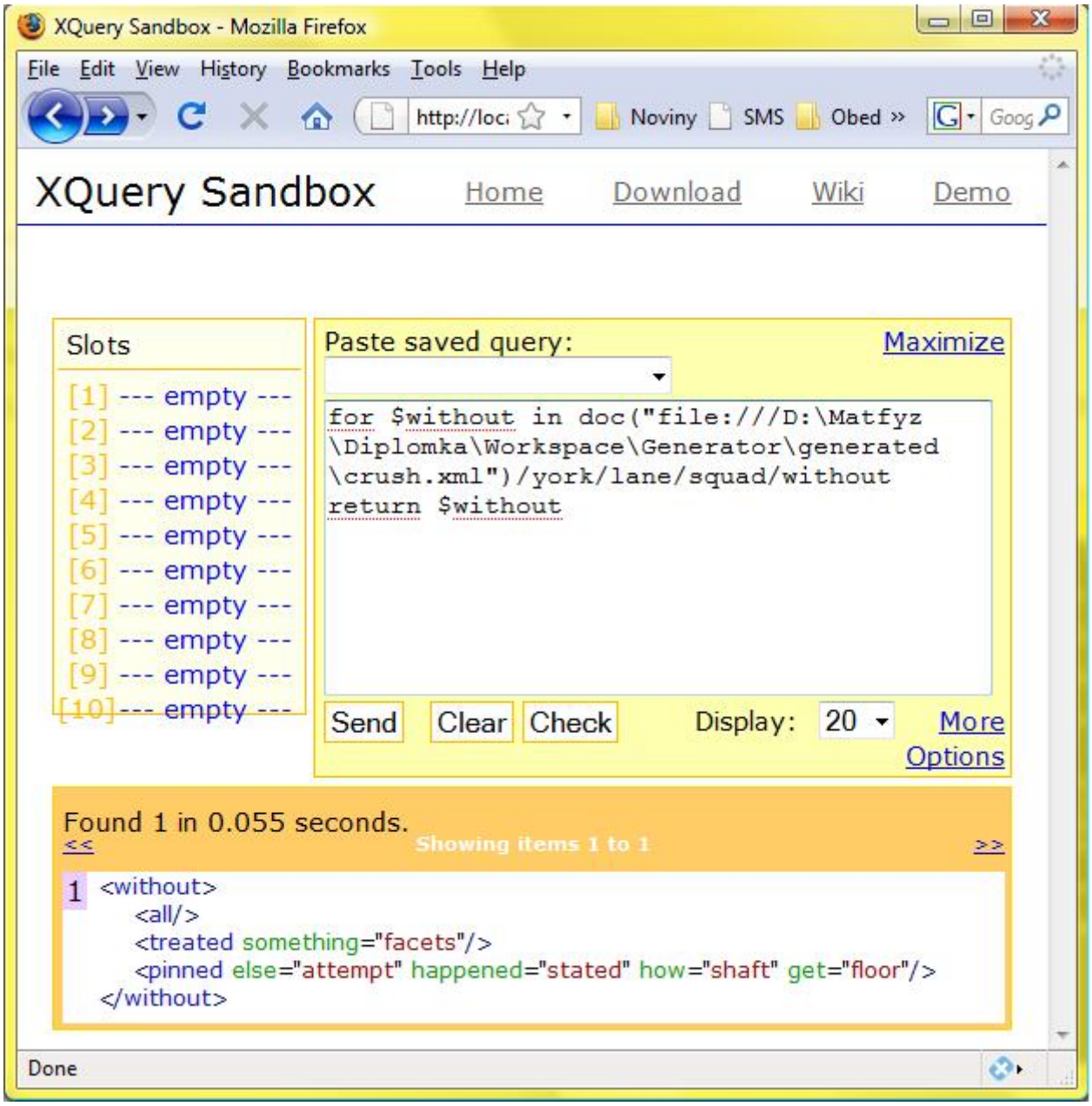


Figure 6.2: eXist querying.

The last thing left is to collect the times of execution of each of the generated queries and compare them to each other and/or to other XMLMSs results. As we have seen in this tutorial, this collecting can be done in the manual way. But there are some ready-to-use automation tools, so-called test-harnesses, that can help us doing so. Currently the most used is the XCheck perl script [Franceschet06]. However, it is supposed to run under GPL/Linux platform only, so in our preliminary tests we used a Bumblebee project [Clarkware03] instead, because it is available also for Windows. Using it, it is fairly easy to determine execution times in reasonable time and effort. We just left out the details to keep this tutorial simple (and also independent on such tools).

6.3 Outcomes of the Benchmark

This section shows some interesting results that were found when testing our benchmark generator. Please note that it is impossible to present full results of all available benchmarks, because their amount is large. We have chosen a few typical use-cases of benchmark generator and shown how to take advantage of its unique abilities (i.e. generation of benchmarks instead of one sealed set of queries).

6.3.1 Comparing XMLMSs

We start by a typical approach to a benchmarking - to compare two or more XMLMSs. We chose six predefined sets of parameters (see Section 5.3 for details) as our six examples of data. Six different benchmarks were generated: data-centric, document-centric, data exchange, reports, research and semantic webs. Total execution times of all benchmark queries can be seen in Table 6.2.

XMLMS	Qizx	Qexo	Saxon
Data-centric documents and queries	3.268s	4.942s	11.423s
Document-centric documents and queries	10.564s	19.919s but failed on text queries	61.767s
Documents for data exchange and queries	8.116s	15.438s	18.29s
Reports and queries	55.324s	failed	failed
Research documents and queries	3.945s	5.05s	7.139s
Semantic web documents and queries	7.43s	9.874s	27.902s

Table 6.2: Comparing different XMLMS using our generated benchmarks.

As we can see from Table 6.2, Qizx and Qexo XMLMSs performed almost equally in data-centric, data exchange, research and semantic web scenarios while Qizx sustained superior performance. Remaining two benchmarks output more interesting results. Both Qexo and Saxon failed in case of reports - the cause of failure was low heap space available (according to the log files). We will use the scalability of benchmark generator in Section 6.3.2 to determine maximum XML document size which Qexo can work with. Overall Saxon has the worst presentation.

The most interesting are the document-centric benchmark results. It seems that Qizx is much faster than Qexo when querying document-centric documents. Moreover, Qexo does not support text queries because of function `contains` which it does not know. We generated 55 absolute ordered queries, 70 aggregate function queries, 137 exact match queries, 12 intermediate result queries, 9 quantification queries, 14 recursive function queries, 31 relative order queries, 9 sorting queries and 72 text queries in the document-centric benchmark. As we can see from Table 6.3 Qexo needs twice time of Qizx and fails in some of the categories. None of the categories shows extra deviation from this pattern, so we can conclude that Qexo has overall problems when working with the document-centric documents. See Figure 6.3 for more details. Saxon has problems with basic queries (especially in exact match queries), but outperforms Qexo in more advanced queries. eXist database was added here to show how hard recursive function queries are.

Query category	**# of generated queries**	**Saxon average time for query (milliseconds)**	**Qizx average time for query (milliseconds)**	**Qexo average time (milliseconds)**	**eXist average time (milliseconds)**
Core XPath (exact match) queries	137	328	25	67	405
XPath 1.0 (absolute and relative order) queries	86	157	26	52	250
Navigational XPath 2.0 (quantification) queries	9	109	64	failed	289
XPath 2.0 (aggregate function) queries	70	55	38	75	301
Sorting queries	9	873	437	failed	274
Recursive function queries	14	failed	failed	failed	1549
Intermediate result queries	12	170	162	234	413

Table 6.3: Comparing XMLMSs in query categories of document-centric benchmark.

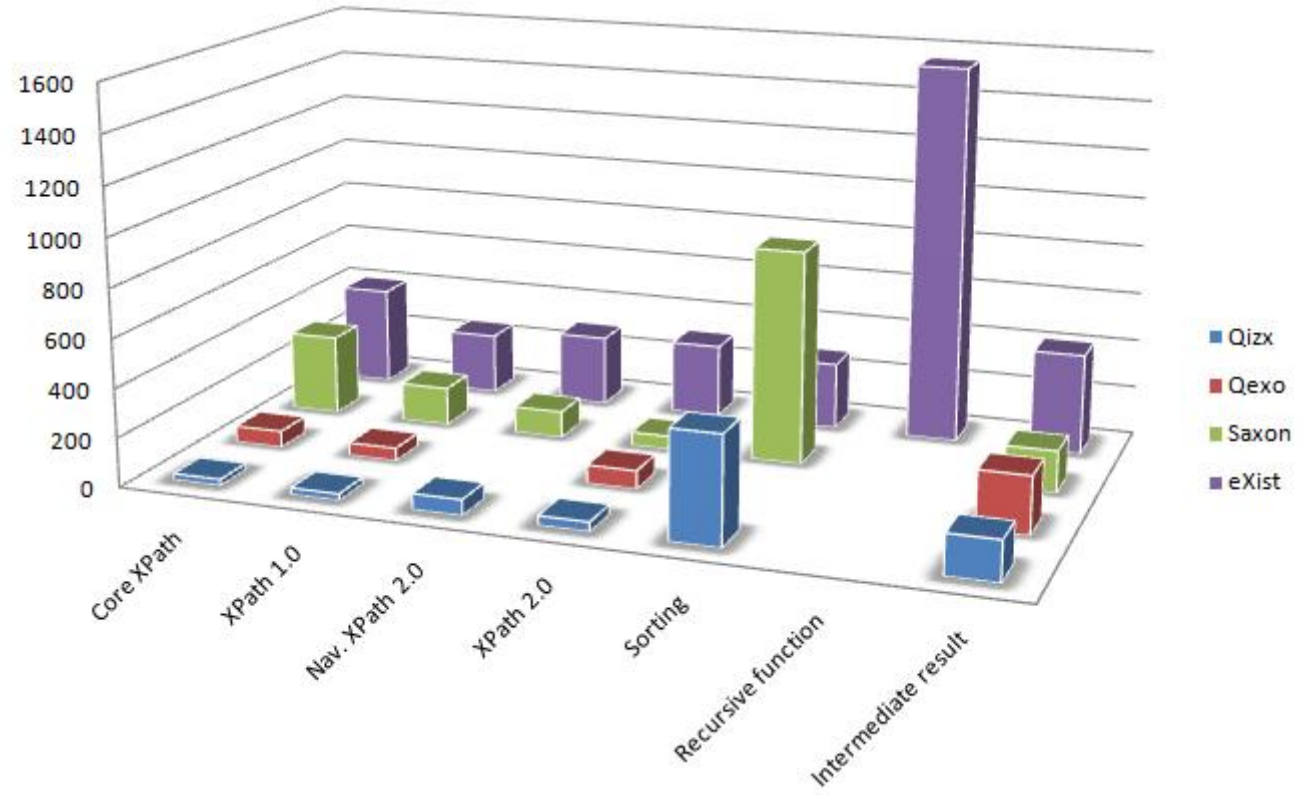


Figure 6.3: XMLMS and queries of different categories in document-centric benchmark.

6.3.2 Scalability of Benchmark Generator

We can also use the benchmark generator capabilities to detect the boundaries of XMLMS. As we have seen in Table 6.2 Qexo has some problems with big files and its default allocated heap space. We will leave its default setting of heap space and consider the Qexo as a black box unconfigurable XMLMS. We will try to determine approximate size of the XML file that Qexo can safely work with. We will use XML documents with various sizes to determine the size boundary of Qexo:

```
java -jar generator.jar NumberOfGeneratedFiles=260 NumberOfElementsGen= ↩
   uniform(12000,18000) ...
```

This produced XML documents with various size ranging from 1MB to 13MB. As we have seen from a log file, the Qexo XMLMS has problems with files bigger than 7MB. This may be a useful information when working with this engine.

6.3.3 Modifying Parameters

It is obvious that possibilities of a benchmark generator are endless. We can use any of the parameters (or a combination) and modify them to see an impact on the database engines. For an illustration we choose one parameter and study its influence on tested XMLMSs. Table 6.4 shows how correlates the percentage of recursion with corresponding total times for text queries. These numbers were quite surprising at first

(see Figure 6.4). But a further investigation led us to conclusion that the percentage of pure recursion has obviously an impact on the number of elements in the results (because we are selecting the interesting - i.e. recursive - element) and logically, the size of results influences the execution time of queries.

Chance of pure recursion (%)	Total time for text queries for Saxon	Total time for text queries for Qizx
1	3.39s	2.265s
25	8.267s	6.692s
50	17.856s	15.98s

Table 6.4: Effect of recursion on text queries.

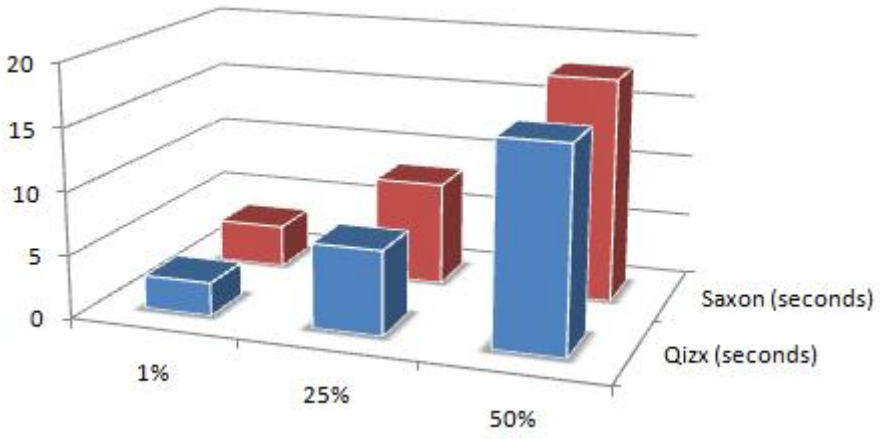


Figure 6.4: Recursion (pure) and text queries.

6.3.4 Consistency of Benchmark Results

One might argue why we should believe the results from such a benchmarks that are completely generated. How big is the chance that the results will be entirely different with the same parameters passed to the benchmark generator? For the sake of result consistency, multiple tries were made when benchmarking an XMLMS. Each time a whole benchmark was re-generated and applied to Qizx. The results can be seen in Table 6.5.

As we can see, the average total execution time for generated document-centric benchmark is 27.413s and the standard deviation is 1.333s. In case of generated semantic web benchmark, the average total time is 30.509s and standard deviation is 5.19s. Considering total randomness of generated data and queries (even the amount of queries is more-or-less randomized) the results are convincing.

XMLMS	Qizx
Total execution time of generated document-centric benchmark #1	26.592s
Total execution time of generated document-centric benchmark #2	29.871s
Total execution time of generated document-centric benchmark #3	25.948s
Total execution time of generated document-centric benchmark #4	27.358s
Total execution time of generated document-centric benchmark #5	27.297s
Total execution time of generated semantic web benchmark #1	30.117s
Total execution time of generated semantic web benchmark #2	27.550s
Total execution time of generated semantic web benchmark #3	28.579s
Total execution time of generated semantic web benchmark #4	33.579s
Total execution time of generated semantic web benchmark #5	32.718s

Table 6.5: Stability of results of a randomly generated benchmark.

Chapter 7

Conclusion

This chapter contains a summary and discussion of the proposed benchmark, as well as its future possible improvements.

7.1 Main Achievements

There are four major achievements of this work: the XML data generator, the XQuery query generator, the experimental results and last but not least, the research on XML benchmarking.

The XML data generator is the utility responsible for generating XML documents and their schemes. It was the most complex part to propose and implement and arguably the most important one. It supports many interesting parameters of XML documents, in fact so many that no third-party solution was suitable to be exploited and utilized. It is a notable fact that its parameters were taken from available statistics about *real-world* XML databases.

The query generator generates queries for XML management systems. It was an interesting idea to generate the whole XML benchmark instead of just XML documents. Every other XML benchmark has its queries pre-defined and fixed and just the data get generated. The main advantage of our approach contrary to a fixed set of queries is that we are much more flexible when generating data. We are not bound to any pre-defined queries.

Experimental results showed us how easy is to determine interesting insights about tested XML databases. Thanks to our different kinds of data and various queries we were able to show different behaviors of tested XMLMSs. This would not be possible with a fixed set of XQuery queries. Moreover, we have created pre-defined sets of benchmark parameters corresponding to the real use-cases of XML data.

Last but not least, this work examined possibilities of XML benchmarking. We went through many hypothetical scenarios how to benchmark an XMLMS and discussed their advantages and disadvantages. There occurred also the problems to be solved. We came across many inevitable limitations when generating realistic data and queries for a benchmark. For example it was hard to satisfy every user-specified parameter about the XML data, because they often conflicted. Our solution in this particular case was to support as many non-conflicting parameters as possible while some conflicting pairs of parameters were made supplementary - a user can choose which one of them she will specify.

7.2 Accomplished Goals

What we did put in three words is an **XML benchmark generator**. Let us see how it meets the objectives that were set up earlier in this thesis.

According to Section 1.2, there are three major reasons to benchmark a database system (which apply also to the XML management systems):

1. to find out which operations are supported by existing solutions;
2. to measure the actual state of the art, i.e. to determine which of current implementations is better for a particular purpose; and
3. to motivate a research in areas where existing solutions do not perform well.

Our benchmark generator can create benchmarks for diverse scenarios and as we have seen in Section 6.3, some of the tested XMLMSs shown limitations in their support of XQuery. In other practical tests, we surveyed the differences between two or more XMLMSs - each one was optimized for different type of data. The third point is harder to answer. We have shown that a parameterized benchmark enables users to benchmark XMLMSs more in detail. It also lets them find new bottlenecks of XMLMSs, which are out of reach of a fixed-query benchmark if it does not already cover them.

It has been said (in Section 6.3 and [Gray93]) that every benchmark should have these four basic properties:

- relevance - a benchmark of XMLMS is a benchmark of XQuery engine and as we have seen in Table 4.3, we generate queries that cover most of the XQuery language
- portability - our benchmark generator is written in a portable Java and outputs an XML format which is also portable
- scalability - generated benchmark is scalable in many possible ways through lots of parameters of the benchmark generator
- simplicity - no parameter of the benchmark generator is mandatory, so the benchmark is as simple as any other XML benchmark; for more information about the usage, see tutorial in Section 6.2

Another way to evaluate quality of an XML-related workload (XQuery queries) was suggested in [Bressan03]:

1. Is there a restriction on XML document structure?
2. Is there a database size and load volume scalability?
3. Is there a query type variability?
4. Is there an ad hoc and open interface for schema input and operation input?

Our answers:

1. No, there are no restrictions. XML documents and their schemes are generated and user is allowed to specify numerous parameters of created files.

2. Yes, there is. Moreover, every other parameter of generated XML documents is scalable as well (not only the size parameter).
3. Yes, there is. As shown in Table 4.3, our benchmarks support more categories of queries than the rest of benchmarks.
4. If we consider our generator parameters as a form of a 'schema', then yes, our benchmark generator is easily extensible by new parameters and new templates for generated queries.

The last evaluation of our benchmark generator will be our own set-up goals, which we had found when analysing related benchmarks (see Section 3.7). Our generated benchmarks have these four additional qualities compared to the other benchmarks:

- Unlimited application domains.
- Unlimited size and/or number of the documents.
- Unlimited amount of XMLMS queries.
- Queries do correspond to the latest XQuery specification.

7.3 Future Work

The work can be improved in various different directions. The most interesting ones are data generation, schema generation, query generation and automation of benchmarking.

The data generator is designed to be easily extensible by adding other parameters of XML documents. This should help to create XML documents even more realistic. Even then it is still important to maintain the user-friendliness of the generator and have the pre-defined sets of parameters.

More advanced third-party schema generator could possibly create some more sophisticated schemes for generated XML documents. It can be parameterized if it is appropriate as well.

Query generator can be easily enhanced by adding new types of generated queries.

Finally, a new tool can be created on top of benchmark generator that will repeatedly generate, test and report results of benchmarking to make it easier to use the benchmarks. Also a user interface of every implemented utility can be improved (see Section 5.4 for details).

In general, there were many interesting problems identified throughout the text, some of them never considered before. Several of them were solved in this work, the rest remains as interesting ideas for a future research.

Chapter 8

Bibliography

[Afanasiev05] Lordana Afanasiev, Ioana Manolescu, and Philippe Michiels, *MemBeR: A Micro-benchmark Repository for XQuery*, August 2005, http://ilps.science.uva.nl/Resources/MemBeR/publications/member-xsym2005.pdf.

[Afanasiev08] Loredana Afanasiev and Maarten Marx, *An analysis of XQuery benchmarks*, 2008, 0306-4379.

[Barbosa02] Denilson Barbosa, Alberto Mendelzon, John Keenleyside, and Kelly Lyons, *ToXgene: A template-based data generator for XML*, June 2002, http://www.cs.toronto.edu/tox/toxgene/docs/ToXgene.pdf.

[Boag07] Scott Boag, Don Chamberlin, Mary F. Fernández, Daniela Florescu, Jonathan Robie, and Jérôme Siméon, *XQuery 1.0: An XML Query Language*, 23 January 2007, http://www.w3.org/TR/xquery/.

[Boncz06] Peter Boncz, Torsten Grust, Maurice van Keulen, Stefan Manegold, Jan Rittinger, and Jens Teubner, *MonetDB/XQuery: A Fast XQuery Processor Powered by a Relational Engine*, June 2006, http://monetdb.cwi.nl/.

[Bourret99] Ronald Bourret, *XML and Databases*, September 1999, http://www.rpbourret.com/xml/XMLAndDatabases.htm.

[Bray06] Tim Bray, Jean Paoli, Michael Sperberg-McQueen, Eve Maler, and François Yergeau, *Extensible Markup Language (XML) 1.0 (Fourth Edition)*, 29 September 2006, Copyright © 2006 W3C, http://www.w3.org/TR/2006/REC-xml-20060816.

[Bressan01] Stéphane Bressan, Mong Li Lee, Ying Guang Li, Zoé Lacroix, and Ullas Nambiar, *The XOO7 XML Management System Benchmark*, November, 2001, http://www.comp.nus.edu.sg/~ebh/XOO7/download/XOO7_TechReport.pdf.

[Bressan03] Stéphane Bressan, Mong Li Lee, Ying Guang Li, Zoé Lacroix, Ioana Manolescu, and Ullas Nambiar, *The XOO7 Benchmark*, Springer Berlin / Heidelberg, January 2003, 1611-3349, Copyright © 2008 .

[Carey94] Michael J. Carey, David J. DeWitt, and Jeffrey F. Naughton, *The OO7 Benchmark*, January 21, 1994, http://pages.cs.wisc.edu/~dewitt/includes/benchmarking/oo7.pdf.

[Chaudhri03] Akmal B. Chaudhri, Awais Rashid, and Roberto Zicari, *XML Data Management: Native XML and XML-Enabled Database Systems*, December 2003, http://exist-db.org/webdb.pdf.

[Clark99] James Clark, Steve DeRose, Mary F. Fernández, Daniela Florescu, Jonathan Robie, and Jérôme Siméon, *XML Path Language (XPath) Version 1.0*, 16 November 1999, http://www.w3.org/TR/xpath.

[Clarkware03] *Getting Started With BumbleBee*, 2003, http://www.x-query.com/bumblebee/GettingStarted.pdf.

[Codd70] Edgar Frank "Ted" Codd, *A relational model of data for large shared data banks*, June 1970, 377-387, 0001-0782, ACM, http://portal.acm.org/citation.cfm?doid=362384.362685.

[Fallside04] David C. Fallside and Priscilla Walmsley, *XML Schema Part 0: Primer Second Edition*, 28 October 2004, http://www.w3.org/TR/xmlschema-0/.

[Fernandez04] Mary Fernández and Jérôme Siméon, *Implementing XQuery 1.0: The Story of Galax*, September 2004, http://www.galaxquery.org/slides/xsym2004.pdf.

[Franceschet06] Massimo Franceschet, Enrico Zimuel, Loredana Afanasiev, and Maarten Marx, *XCheck - A benchmark checker for XML query processors*, May 24, 2006, http://ilps.science.uva.nl/Resources/XCheck/manual/manual.pdf.

[Gray93] Jim Gray, *The Benchmark Handbook for Database and Transaction Systems (2nd Edition)*, 1993, 1-55860-292-5, Morgan Kaufmann.

[Kruchten95] Philippe Kruchten, *Architectural Blueprints—The "4+1" View Model of Software Architecture*, November 1995, http://www.cs.ubc.ca/~gregor/teaching/papers/4+1view-architecture.pdf, IEEE Computer Society Press, 0740-7459, 42-50.

[Leidecker07] Nico Leidecker, *Having Fun With PostgreSQL*, June 2007, http://www.portcullis.co.uk/uplds/whitepapers/Having_Fun_With_PostgreSQL.pdf.

[McHugh97] Jason McHugh, Serge Abiteboul, Roy Goldman, Dallan Quass, and Jennifer Widom, *Lore: A Database Management System for Semistructured Data*, ACM, September 1997, http://infolab.stanford.edu/lore/pubs/lore97.pdf.

[Mlynkova06] Irena Mlynkova, Kamil Toman, and Jaroslav Pokorny, *Statistical Analysis of Real XML Data Collections*, June 2006, http://kocour.ms.mff.cuni.cz/~mlynkova/dp.html.

[MySQL05] *A Practical Guide to Migrating to MySQL 5.0*, MySQL, October 2005, http://www.ibm.com/developerworks/db2/library/techarticle/dm-0604saracco/.

[Oracle06] *A Comparison of Oracle Berkeley DB and Relational Database Management Systems*, November 2006, http://www.oracle.com/database/docs/Berkeley-DB-v-Relational.pdf.

[Oracle07] *Oracle Database 11g XML DB Technical Overview*, July 2007, http://www.oracle.com/technology/tech/xml/xmldb/Current/xmldb_11g_twp.pdf.

[Oracle08] *Oracle Database 11g Product Family*, January 2008, http://www.oracle.com/technology/products/database/oracle11g/pdf/database-11g-product-family-technical-whitepaper.pdf.

[Rahm02] Erhard Rahm and Timo Böhme, *XMach-1: A Multi-User Benchmark for XML Data Management*, August 2002, http://dbs.uni-leipzig.de/files/projekte/XML/paper/xmach1-eextt2002.pdf.

[Runapongsa02] Kanda Runapongsa, Jignesh M. Patel, H. V. Jagadish, Yun Chen, and Shurug Al-Khalifa, *The Michigan Benchmark*, The University of Michigan, 2002, http://www.eecs.umich.edu/db/mbench/.

[Sahuguet01] Arnaud Sahuguet, *Kweelt: more than just "yet another framework to query XML!"*, 2001, 0163-5808, ACM.

[Saracco06] Cynthia M. Saracco and Don Chamberlin, *Query DB2 XML data with XQuery*, April 2006, http://www.ibm.com/developerworks/db2/library/techarticle/dm-0604saracco/.

[Schmidt01] Albrecht Schmidt, Florian Waas, Martin Kersten, Daniela Florescu, Ioana Manolescu, Michael J. Carey, and Ralph Busse, *The XML Benchmark Project*, 1386-3681, CWI, April 2001.

[Smith00] Wayne D. Smith, *TPC-W: Benchmarking An Ecommerce Solution*, 2000, http://www.tpc.org/tpcw/TPC-W_Wh.pdf.

[Vithanala05] Prasadarao K. Vithanala, *Introduction to XQuery in SQL Server 2005*, June 2005, http://technet.microsoft.com/en-us/library/ms345122.aspx.

[W3Schools] Hege Refsnes, StÍle Refsnes, and Jan Egil Refsnes, *W3Schools*, http://www.w3schools.com/.

[Webster06] Melissa Webster, *Northwest Airlines Streamlines Aircraft Mainetance with X-Hive/AMDS*, January 2006, http://content.hartman-communicatie.nl/assets/binaries/idc_jan06.pdf.

[XHTML00] W3C HTML Working Group, *XHTML™ 1.0 The Extensible HyperText Markup Language (Second Edition)*, 26 January 2000, http://www.w3.org/TR/xhtml1.

[Yao02] Benjamin Bin Yao, M. Tamer Özsu, and John Keenleyside, *XBench - A Family of Benchmarks for XML DBMSs*, Springer-Verlag, London, UK, December 2002, 3-540-00736-9.

Appendix A

Content of CD

Contents of the attached CD:

- `bin` - compiled sources
- `docs/api` - javadoc generated from comments in source files
- `docs/manual` - other documentation than javadoc
- `pages` - www interface of CD
- `src` - complete sources
- `index.html` - open it to run WWW user interface
- `readme.txt` - a brief help file