Benchmarking of Native XML Database Systems

by

Yu Yang

A Thesis Submitted in Fulfilment of the Requirements for the Award of

Master of Computer Science
(Honours)

from
University of Wollongong
School of Information Technology and Computer Science

February 2005
DECLARATION

I, Yu Yang, declare that this thesis, submitted in fulfilment of the requirements for the award of Master of Computer Science (Honours), in the School of Information Technology and Computer Science, University of Wollongong, is wholly my own work unless otherwise referenced or acknowledged. The document has not been submitted for qualifications at any other academic institution.

Yu Yang

February 2005
Abstract

In this thesis, XML database systems, standard database benchmark techniques, and XML database system benchmarks such as XBench, XMark, XMach-1, and XOO7 are reviewed. A new benchmark system called MyBench is provided for benchmarking of native XML database systems. First, graph grammars are utilised to define a group of productions with different tree structures, then the algorithm for generating XML documents is provided. Second, a group of benchmark query patterns with different functionalities is provided. We also depict how to automatically generate benchmark queries in terms of these patterns and the parameters input by the user. Third, we utilise XSLT to implement MyBench XML document generator, and we also utilise relational DBMS to implement benchmark queries. Finally, MyBench is compared with other XML database benchmark techniques. We also summarise the contributions of this thesis and formulate some open problems.
ACKNOWLEDGEMENTS

I would like to extend my sincere thanks to my supervisors Dr. Janusz Getta, Dr. Peter Eklund, Dr. John Fulcher without whose invaluable assistance this thesis would not have been possible.

My thanks also go to the technical staff in the School of Information Technology and Computer Science for the help they gave me.

I am also grateful to my parents and sister for their support throughout this work.
# Table of Contents

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Declaration</td>
<td>1</td>
</tr>
<tr>
<td>Abstract</td>
<td>2</td>
</tr>
<tr>
<td>Acknowledgments</td>
<td>3</td>
</tr>
<tr>
<td>Table of Contents</td>
<td>4</td>
</tr>
<tr>
<td>List of Figures</td>
<td>8</td>
</tr>
<tr>
<td>List of Tables</td>
<td>10</td>
</tr>
<tr>
<td><strong>1. Introduction</strong></td>
<td>11</td>
</tr>
<tr>
<td><strong>2. XML</strong></td>
<td>14</td>
</tr>
<tr>
<td>2.1. Definition</td>
<td>14</td>
</tr>
<tr>
<td>2.2. Well-formed XML</td>
<td>14</td>
</tr>
<tr>
<td>2.2.1. Elements</td>
<td>15</td>
</tr>
<tr>
<td>2.2.2. Entity references</td>
<td>15</td>
</tr>
<tr>
<td>2.2.3. Comments</td>
<td>15</td>
</tr>
<tr>
<td>2.2.4. Processing instructions</td>
<td>16</td>
</tr>
<tr>
<td>2.2.5. CDATA sections</td>
<td>16</td>
</tr>
<tr>
<td>2.2.6. Document type declarations</td>
<td>16</td>
</tr>
<tr>
<td>2.3. XSLT</td>
<td>17</td>
</tr>
<tr>
<td>2.4. DTD</td>
<td>17</td>
</tr>
<tr>
<td>2.5. XML Schemas</td>
<td>18</td>
</tr>
<tr>
<td>2.6. Graphical interpretation</td>
<td>19</td>
</tr>
<tr>
<td><strong>3. XML database systems</strong></td>
<td>21</td>
</tr>
<tr>
<td>3.1. Native XML database systems</td>
<td>21</td>
</tr>
<tr>
<td>3.2. XML-enabled database systems</td>
<td>21</td>
</tr>
<tr>
<td>3.3. Comparison between native XML and XML-enabled database systems</td>
<td>22</td>
</tr>
</tbody>
</table>
3.4. Current native XML database products

3.4.1. Lore
   3.4.1.1. Lorel query language
   3.4.1.2. DataGuides
   3.4.1.3. Managing external data
   3.4.1.4. Proximity search

3.4.2. eXist
   3.4.2.1. Document management in collections
   3.4.2.2. Index-based query processing
   3.4.2.3. Interfaces

3.4.3. 4Suite, 4Suite Server

3.4.4. Xindice

3.4.5. DBDOM

3.4.6. DOM-Safe

3.4.7. GoXML DB

3.4.8. XIS

3.4.9. Tamino

3.4.10. TEXTML server

3.4.11. Berkeley DB XML

3.4.12. Summary of native XML database products

4. Standard Database Benchmarking Techniques

4.1. Relational database system benchmark
   4.1.1. TP1 benchmark
   4.1.2. Wisconsin benchmark

4.2. Object relational database system benchmark

4.3. Object oriented database system benchmark

4.4. TPC benchmarks
   4.4.1. TPC-C
   4.4.2. TPC-H
   4.4.3. TPC-R
   4.4.4. TPC-W
   4.4.5. Difference between these TPC benchmarks
5. XML Database System Benchmarks

5.1. XBench
   5.1.1. ToXgene
   5.1.2. XBench queries

5.2. XMark
   5.2.1. Xmlgen
   5.2.2. XMark queries

5.3. XMach-1
   5.3.1. XMach-1 document generator
   5.3.2. XMach-1 queries

5.4. XOO7
   5.4.1. XOO7 document generator
   5.4.2. XOO7 queries

6. Graph Grammars

6.1. Definitions
6.2. Basic concepts
6.3. Specific definition
6.4. DTD, XML Schemas and graph grammars

7. MyBench XML Document Generator

7.1. Description of graphs
7.2. BNF notation for document generator
7.3. Algorithm of document generator

8. Benchmark Queries

8.1. Basic concepts
8.2. Classification of queries
8.3. Patterns and instances of benchmark queries
8.4. Algorithm of benchmark query generator

10. Implementation of Benchmark Query Generator
   10.1. Generation of information table
   10.2. Generation of paths
   10.3. Insertion of specified attributes
   10.4. Output of benchmark queries

11. Related XML Benchmark Systems
   11.1. Comparison of XML document structure control
   11.2. Comparison of XML document size control
   11.3. Comparison of benchmark query generation

12. Contributions, Conclusions and Open Problems
   12.1. Contributions
   12.2. Conclusions
   12.3. Open problems

Bibliography

Appendix A
<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.1</td>
<td>An example of a directed graph</td>
<td>20</td>
</tr>
<tr>
<td>6.1</td>
<td>Single level production</td>
<td>41</td>
</tr>
<tr>
<td>6.2</td>
<td>Examples of single level production</td>
<td>42</td>
</tr>
<tr>
<td>6.3</td>
<td>Multiple level production</td>
<td>43</td>
</tr>
<tr>
<td>6.4</td>
<td>Multiple level production with unbalanced structure</td>
<td>43</td>
</tr>
<tr>
<td>6.5</td>
<td>Multiple level production with balanced structure</td>
<td>44</td>
</tr>
<tr>
<td>6.6</td>
<td>Start graph and a set of productions</td>
<td>46</td>
</tr>
<tr>
<td>6.7</td>
<td>Intermediate graph one</td>
<td>47</td>
</tr>
<tr>
<td>6.8</td>
<td>Intermediate graph two</td>
<td>47</td>
</tr>
<tr>
<td>6.9</td>
<td>Terminal graph</td>
<td>47</td>
</tr>
<tr>
<td>6.10</td>
<td>Control of the depth and width</td>
<td>48</td>
</tr>
<tr>
<td>6.11</td>
<td>Reusability of productions</td>
<td>49</td>
</tr>
<tr>
<td>7.1</td>
<td>An example of a final production</td>
<td>50</td>
</tr>
<tr>
<td>7.2</td>
<td>A group of productions</td>
<td>51</td>
</tr>
<tr>
<td>7.3</td>
<td>Generation of multiple level production</td>
<td>51</td>
</tr>
<tr>
<td>7.4</td>
<td>Generation of a final production</td>
<td>52</td>
</tr>
<tr>
<td>8.1</td>
<td>Path models</td>
<td>62</td>
</tr>
<tr>
<td>8.2</td>
<td>Path ([P_1 X_1, P_3 X_1])</td>
<td>63</td>
</tr>
<tr>
<td>8.3</td>
<td>Composite path ([P_1 X_1, Path, P_6 X_1])</td>
<td>64</td>
</tr>
<tr>
<td>8.4</td>
<td>Path ([P_1 X_2, 3])</td>
<td>64</td>
</tr>
<tr>
<td>8.5</td>
<td>Composite path (Path, 2)</td>
<td>65</td>
</tr>
<tr>
<td>9.1</td>
<td>Production * m</td>
<td>72</td>
</tr>
<tr>
<td>9.2</td>
<td>An example of self-copy</td>
<td>73</td>
</tr>
<tr>
<td>9.3</td>
<td>Combine function</td>
<td>74</td>
</tr>
<tr>
<td>9.4</td>
<td>([P_1 * m_1, P_2 * m_2, \ldots, P_n * m_m])</td>
<td>75</td>
</tr>
<tr>
<td>9.5</td>
<td>An example of combination</td>
<td>76</td>
</tr>
<tr>
<td>9.6</td>
<td>((P_1 * m_1, P_2 * m_2, \ldots, P_n * m_m) * k)</td>
<td>77</td>
</tr>
<tr>
<td>10.1</td>
<td>Production one</td>
<td>79</td>
</tr>
</tbody>
</table>
Figure 10.2: Production two  
Figure 10.3: Production three
List of Tables

Table 3.1: Summary of native XML database products 28
Table 10.1: Structure of relational table 79
Table 10.2: XMLtoTable_P 80
Table 10.3: XMLtoTable_P 80
Table 10.4: The updated XMLtoTable_P 82
Table 10.5: Index of Level 83
Table 10.6: Index of Tagname 83
XML is self-describing. It can provide flexible information identification, and can be extensively used in many application domains such as chemistry, biology, and e-business, etc. With the development of web applications and the large amounts of XML documents that are being generated, it is therefore necessary to work out how to manage them efficiently. Databases are the prime storage engines for many different types of data. Traditional DBMSs are designed for regular data. However, XML data often includes some irregular data such as pictures, audio and video files etc, which means that the storage of XML data is a challenge to traditional DBMSs.

Current research increasingly focuses on how to effectively manage large amounts of XML data. Now, more and more XML database products are available, and there are some different features and functionalities in these products. Therefore, we need an effective method to analyse and evaluate their performance.

Database benchmarking is a test that uses a set of programs and data to evaluate the performance of databases in a given configuration. Benchmarking of native XML database systems not only means evaluating the performance of current products, but also is regarded as an incentive for further development of XML database systems.

The aim of this thesis is to provide a benchmark called MyBench, which can evaluate the characteristics of current native XML database systems. There are three problems to be solved: the generation of XML documents, the generation of benchmark queries, the implementation of the XML document generator, and the implementation of the benchmark query generator.
First, we utilise graph grammars to define a group of productions with different tree structures. Then, in terms of the requirements of XML document structure and size, an algorithm for generating XML documents is provided. Second, while generating XML documents, we also automatically generate benchmark queries. We firstly provide a group of benchmark query patterns with different functionalities. A group of benchmark queries is generated according to queries patterns and the parameters input by the user. Third, an XSLT stylesheet is utilised to implement two templates for generating XML documents. We also utilise a relational DBMS to automatically generate benchmark queries. Finally, MyBench is compared with other XML benchmark systems on the XML document generator and the benchmark query generator.

Currently we mainly focus on solving the above three problems, and we provide two models for the generation of XML documents and benchmark queries. It should be acknowledged that there is no real evaluation phase in this thesis. In the future, MyBench system could be run on some comparable products of native XML database systems.

The thesis is structured as follows:

Chapter 1 provides a brief description of the main problems and a strategy for solutions thereof.

Chapter 2 defines XML, and describes XSLT, DTD, XML Schemas, as well as XML document categories.

Chapter 3 depicts native XML and XML-enabled database systems and the main difference between a native XML database system and an XML-enabled database system. It also surveys some current native XML database products.

Chapter 4 introduces benchmark techniques of relational database systems, object relational database systems, and object oriented database systems; it also introduces the TPC series benchmark.
Chapter 5 reviews current XML database system benchmark techniques. They include XBench, XMark, XMach-1, and XOO7.

Chapter 6 defines some basic concepts and graph grammars.

Chapter 7 describes how to use graph grammar to generate XML documents with different structures and sizes. It uses BNF notations to describe basic concepts and provides an algorithm for generating an XML document.

Chapter 8 examines how to generate benchmark queries with different query functions. It provides a group of patterns and instances of benchmark queries, and includes an algorithm for generating benchmark queries.

Chapter 9 describes how to use an XSLT stylesheet to implement the MyBench XML document generator.

Chapter 10 describes how to use a relational DBMS to implement a benchmark query generator.

Chapter 11 describes the main differences between the MyBench system and other XML database benchmark systems.

Chapter 12 summarises the contributions of this thesis and formulates some open problems.
Chapter 2

XML

2.1. Definition

The abbreviation, XML [1], stands for Extensible Markup Language. XML describes the concepts and rules for the creation of specific markup languages. A markup language describes how the text is structured according to the tags in a document. Tags are used to mark words to indicate actions or identifications.

XML is a markup language for documents that contain structured information [2]. Documents that have structured information usually include both the content and what the content stands for. Content can be words, pictures, audio and video files, etc. Different positions of content in a document can lead to different meaning. For example, content that is located in a block of text is different from content in a table, even if they are completely similar.

A simple XML document example is as follows:

```xml
<?xml version="1.0"?>
<UOW> <title> Mr. </title> <name> <first>Yu </first> <last>Yang </last> </name> is a master student. </UOW>
```

2.2. Well-formed XML

Well-formed XML is XML that meets certain syntactical rules outlined in the XML specification [1]. The XML document is composed of markup and content. There are usually six kinds of markup in an XML document: elements, entity references, comments, processing instructions, CDATA sections, and document type declarations.
2.2.1. Elements

Elements are the basic components of an XML document, and represent pieces of information. Some nested elements perhaps might contain more specific information, such as attributes, character data, and entity references, etc. Elements are indicated or marked up by the tags in a document. Element content refers to the text between the start-tag and end-tag of an element. For example, `<first>Yu</first>` is an element, and “Yu” is the element content.

Attributes are information that is attached to the element. Attributes can add more details to the element and often define an instance of an element. For example, `<name nickname = “blue bird”>` is a name element with the attribute nickname that has the value “blue bird”.

2.2.2. Entity references

Entity references are markup that the parser can replace with character data. In XML the entities represent some special characters, and every entity has a unique name. Entity references provide an alternative way to insert some specific characters into the document.

For example, `<comparison>9 is > 7 & 5 is < 6</comparison>` could be made by doing the following: `<comparison>9 is &gt; 7 &amp; 5 &lt; 6</comparison>`.

2.2.3. Comments

In XML, comments begin with the character sequence `<!--` and end with the sequence `-->`. The parser can ignore the content that appears between them, and only verify that the comment is well formed. For example, `<first>Yu</first> <!-- Yu is Mr. Yang’s first name -->.`
2.2.4. Processing instructions

Processing instructions (PIs) are directives that are intended for an application. They are not regarded as a part of the character data in a document, but as the information that is passed through to the application. They can be defined in the following manner: <?Target Instructions for command ?>.

For example, <?nameprocessor SELECT * FROM names WHERE first name = Yu ?>. In this example, the PI target is nameprocessor, and the actual text of the PI is SELECT * FROM names WHERE first name = Yu.

2.2.5. CDATA sections

CDATA sections can be used to skip characters in an XML document during the time of processing. The sections are defined by beginning at the <![CDATA[ and ending at the ]]>.

For example, in order to skip <name> Yu Yang </name>, the following can be used: <![CDATA[<name> Yu Yang </name>]]>.

2.2.6. Document type declarations

XML uses the XML declaration to label documents as XML, and provides the parsers with information. For example, a typical XML declaration appears as follows: <?xml version='1.0' encoding='UTF-16' standalone='no' ?>.

The document type declaration is usually followed by the XML declaration. It is used to declare the root element and provide the location of the DTD. The document type declaration can refer to an external DTD, or only include part of the DTD.

For example, <!DOCTYPE Students SYSTEM “students.dtd” >. This example indicates that the DTD and XML document are in the same directory, and the root element is Students.
2.3. XSLT

XSLT [1] (Extensible Stylesheet Language Transformation) is a language that can transform XML documents into other text-based formats. These formats might include XML, HTML, WML, or XSL etc.

XSLT needs at least three text documents: a source document, an XSLT document, and a target document. The source document is a well-formed XML document that is to be transformed. The XSLT document is a text document in an XML format. This text document uses the stylesheet to convert the source document into the target document. The target document may be XML format files, or it might be in a completely different format such as text or PDF file, etc. The XSLT can generate the target document by applying the transformation rules to the source XML document. In order to perform an XSLT transformation, it also needs a XSLT engine. The engine is the software that will carry out the instructions in the stylesheet. At present, there are various engines available, such as Saxon, Xalan, and MSXML.

Now, programming by XSLT is different from programming by Java or C++. XSLT implements transformation by using a template, but does not use detailed programming. The template specifies the conditions where the process takes place and the output is to be generated.

2.4. DTD

Document Type Definition (DTD) [1][3] is a set of rules that define the hierarchical structure of any XML document. The XML parser utilises these rules to determine whether the XML document is valid.

DTDs can be broken down into certain basic parts: elements, tags, attributes, and entities. An element declaration specifies the name of the element and what content is valid in it. For example, `<!ELEMENT name_list (name*)>` is an element declaration. Tags like `<name>` and `</name>` are used to indicate elements. Attributes provide extra information about elements and can be used to describe the properties of an element. The following "image" element has additional information about a source file: `<image
Entities are the variables that can be reused within the document. When an XML parser parses a document, the entities can be expanded.

The following is a simple example of a DTD,

```xml
<?xml version="1.0"?>
<!DOCTYPE student [
  <!ELEMENT student (first name, last name, address)>  
  <!ELEMENT first name (#PCDATA)>  
  <!ELEMENT last name (#PCDATA)>  
  <!ELEMENT address (#PCDATA)>  
]>  
<student>  
  <first name> Yu </first name >  
  <last name> Yang </last name>  
  <address> Wollongong NSW </address>  
</student>
```

2.5. XML Schemas

XML Schema [1][4] is an XML based alternative to DTD. Users can exploit XML Schema to depict XML document structure. XML Schema Definition (XSD) is regarded as XML Schema language.

XML Schemas are the successors of DTDs, and can take the place of DTDs in most of Web applications. Unlike DTDs, XML Schema documents are written in XML. Instead of using another notation, XML Schemas are well-formed XML documents. XML Schemas support data types. Some basic simple types such as string, date and time can be directly used for the elements. Furthermore, XML Schemas can also be used to construct complex types.
The following is an example of an XML Schema:

```xml
<?xml version="1.0"?>
  targetNamespace="http://www.uow.edu.au/students"
  xmlns="http://www.uow.edu.au/students"
  elementFormDefault="qualified">
  <xs:element name="student">
    <xs:complexType>
      <xs:sequence>
        <xs:element name="first name" type="xs:string"/>
        <xs:element name="last name" type="xs:string"/>
        <xs:element name="address" type="xs:string"/>
      </xs:sequence>
      <attribute name="ID" type="string"/>
    </xs:complexType>
  </xs:element>
</xs:schema>
```

2.6. Graphical interpretation

It is obvious that the structure of an XML document is similar to a hierarchy-directed graph structure. An XML document can be modelled as a directed graph. The nodes in the graph represent XML elements or attributes, and the edges represent parent-children relationships.

For example:

```xml
<T1>
  T1 element content
  <T11 id="T11"> T11 element content
    <T111 id="T111">T111 element content</T1111>
    <T112 id="T112">T112 element content</T1121>
  </T111>
  <T12 id="T11"> T12 element content
    <T121 id="T121">T121 element content</T1211>
    <T122 id="T122">T122 element content</T1221>
  </T121>
</T112>
</T1>
```
The above XML document could be represented by the following graph.

![Directed Graph Diagram]

Figure 2.1: An example of a directed graph
3.1. Native XML database systems

Native XML databases (NXDBs) [5] [6] are special database systems that are designed for storing XML data. NXDBs store XML data by conserving the original structure of the document in the database.

A native XML database system usually includes the following three main points:

a) The storage object of NXDBs is XML data.
b) XML documents are regarded as the basic unit of storage.
c) NXDBs store XML in a "native" form, but may not be a standalone database.

There are some different products in terms of NXDBs at present, but most of them have common functionalities. First, they all store XML data at the document level, and many NXDBs manipulate XML data by managing the collections. Second, all NXDBs support at least one kind of query language and most of them support XPath. Furthermore, some NXDBs also support update functionality. Third, almost all NXDBs have indexing on collection level, and even on element and attribute values. With the continuing development of NXDBs, more and more functionalities will be added into future products.

3.2. XML-enabled database systems

XML-enabled databases [5] first break down XML documents into pieces of data elements, and then store these data elements as basic data objects in the relational tables. A form of interfacing software or middleware is obviously needed to transfer data between XML documents and XML-enabled databases. Through this middleware,
the database systems can efficiently map the whole XML document into relational tables.

XML-enabled database systems mainly include DB2 XML Extender from IBM, Oracle 9i XDB, Microsoft SQL Server 2000, Sybase ASE 12.5, Informix, etc. The main middleware packages include ODBC, JDBC, or OLE DB to access data in relational databases

3.3. Comparison between native XML and XML-enabled database systems

Both native XML databases and XML-enabled databases can be used to store XML data and documents, but they still have differences in storage methods [5].

NXDBs actually store XML documents in the databases. In NXDBs, XML document are regarded as a basic storage unit. Most of NXDBs manage the stored XML documents as collections, and can keep the whole structure of the XML document, whereas XML-enabled databases first break down the XML documents into many data objects, and then store them in the relational tables. In other words, XML-enabled databases have to map XML documents into another data structure that can be stored in the traditional database systems.

For the storage of XML documents with regular and simple data, XML-enabled databases can easily map such documents into relational tables. However, irregular and complex data might also exist in XML documents. If XML-enabled databases directly store these irregular data into relational tables, this will result in many null values and thus slow down the database system’s performance. On the other hand, NXDBs can conserve the original structure of the document. They can easily insert the data into databases and then retrieve it in original format, even for some complex data. In addition, unlike the queries in XML-enabled databases, the queries in NXDBs don’t need to rebuild the document.
3.4. Current native XML database products

In this section, we briefly introduce some typical native XML database products. This list comprises both open source (Lore, eXist, 4Suite, Xindice, and DBDOM) and commercial products (DOM-Safe, GoXML DB, XIS, Tamino, TEXTML Server, and Berkeley DB XML). More product information of native XML database systems can be found from [5].

3.4.1. Lore

Lore [7] is a database management system that is mainly designed for storing semi-structured data and XML data. Lore has been developed by Stanford University since 1995. Originally, Lore was only designed for semi-structured data that has no particular DTDs or schema, but now it can support XML data. Lore can run on SUN and Linux operating systems, and provides an independent query language named Lorel. Lore also supports some special technologies such as DataGuides, management of external data, and proximity search.

3.4.1.1. Lorel query language

The Lorel language is a query language that is designed for XML or other semi structured data. Based on OQL, Lorel includes some modifications and extensions. These changes are useful for querying XML or semi-structured data and can provide powerful path traversal and results retrieval.

3.4.1.2. DataGuides

DataGuide is a structural summary of all paths, and exists in each Lore database. Users can check the database structure and specify the queries from the DataGuide directly. Lore databases can also use the DataGuides for the storage of statistics, and improve the performance of query processing.
3.4.1.3. Managing external data

The external data manager can make Lore merge the information that is from one or more external data sources. This functionality can retrieve and integrate the data from the outside sources during the time of query processing. Users can also limit the amount of data by specifying some flexible parameters.

3.4.1.4. Proximity search

Lore supports proximity search technology, although traditional query languages do not support this concept. This technology can rank database objects in terms of their proximity to other objects. Thus, the users can easily find the relevant data in terms of this kind of fuzzy relationships.

3.4.2. eXist

eXist [8] is an open source native XML database. This database is entirely written in Java and can easily run on any platforms. eXist can run either as a stand-alone model, a inside servlet-engine, or a technology that is embedded into an application. It supports XPath, XUpdate, and XQuery.

3.4.2.1. Document management in collections

eXist manages the XML documents as collections. The documents have no DTDs or schema. The collections can contains different types of documents, and they can be mixed in the one collection.

3.4.2.2. Index-based query processing

The index in eXist can quickly identify the relationships in structure between nodes in a document. The relationships between these nodes are like the relationship between parent and child, ancestor and descendant, or siblings. eXist can process path expressions by indexing the relative information. An automatic index can be created by default.
3.4.2.3. Interfaces

The database server offers XML-RPC, HTTP, and SOAP interfaces. XML:DB API uses XML-RPC calls to access to a remote database or embedded use. Another uses the persistent DOM to directly access to a running database.

3.4.3. 4Suite, 4Suite Server

4Suite [9] is implemented in Python and C, and can run on Windows and Unix platforms. 4Suite is designed for XML processing and object database management. Users can rapidly develop web application programs by using 4Suite, which also supports multiple methods of data access, query, and indexing.

3.4.4. Xindice

Xindice [10] is an open source native XML database. It is designed to store collections of small or medium-sized documents. Xindice is written in the Java, and supports XPath and XUpdate. Xindice can index the element and attribute values of XML documents.

Users can manage and operate the database system by using command line tools. Also, Xindice allows the users to replace or insert the content in an XML document at the time of query process.

3.4.5. DBDOM

The Document Object Model (DOM) [1] is the standard API for accessing and manipulating the objects and their relationship. DBDOM [11] implements DOM on a relational database, and applies RDBMS on an XML application server. It defines a fixed database schema with a group of tables. These relational tables express the relative XML structures.
3.4.6. DOM-Safe

DOM-Safe [12] is a native XML database that works directly on XML files. XML data are stored as independent objects, and DOM-Safe can quickly retrieve the information at a low level. It allows multi-users to access and manipulate to one or more XML documents. DOM-Safe supports XPath, SAX, and XMLDB.

3.4.7. GoXML DB

GoXML DB [13] is a commercial database that comprises storage manager, index system, XPath query engine, and a Java API, developed by XML Global. GoXML DB supports Windows, Solaris, and Linux systems. It also supports XPath and XQuery.

GoXML DB can be a stand-alone XML database, or it can be combined with other products. GoXML DB provides a graphical schema manager that can be used to store and manage different schemas.

3.4.8. XIS

Extensible Information Server (XIS) [14] is a native XML database that has been developed by eXcelon. There are three main parts to XIS: a native XML database, an XSLT engine, and a suite of tools with different functions. XIS can run on Windows, Solaris, HP UNIX, and Linux. It also supports XQuery and XPath languages. The core of XIS is its Dynamic XML Engine (DXE). XIS provides XConnects Integration Engine to combine data and content sources. Another feature of XIS is its DXE Manager. DXE Manager is a graphical administration tool.

3.4.9. Tamino

Tamino XML Server [15] is a native XML database that comprises core services, enabling services, and solutions, developed by Software AG. The database supports XPath and XQuery languages. Tamino XML Server can run on all major operating systems. Users can extend some functionality by using Tamino X-Tension. Tamino
Manager is an administration tool that provides a graphical user interface to manage the database system. Tamino XML Server not only processes well-formed XML documents, but also processes the non-XML data, such as audio, video, and PDF files. The system supports EJB API and a Java API.

3.4.10. TEXTML server

TEXTML server [16] is a native XML database system that has been developed by IXIA. TEXTML server runs on the Windows platform and only supports XPath as its query language. TEXTML Server has a graphical user interface by which users can search XML documents through some given parameters.

3.4.11. Berkeley DB XML

Berkeley DB XML [17] is a native XML data manager that is based on Berkeley DB. Berkeley DB XML can run on Windows, Solaris, and Linux. It only supports the XPath query language. Berkeley DB XML can store and retrieve XML data and semi-structured data effectively, and it also provides flexible indexing for improving query performance. Berkeley DB XML stores XML documents as collections. A single application may operate on several collections at the same time. It can provide C++ and Java APIs.
3.4.12. Summary of native XML database products

The following table is a summary of native XML database products:

<table>
<thead>
<tr>
<th>Product</th>
<th>Developer</th>
<th>Platforms</th>
<th>Query Languages</th>
</tr>
</thead>
<tbody>
<tr>
<td>Lore</td>
<td>Stanford University</td>
<td>SUN Unix, Linux</td>
<td>Lorel</td>
</tr>
<tr>
<td>eXist</td>
<td>Wolfgang Meier</td>
<td>Windows, Unix, and Linux</td>
<td>XPath, XUpdate, and XQuery</td>
</tr>
<tr>
<td>4Suite, 4Suite Server</td>
<td>FourThought</td>
<td>Windows, Unix</td>
<td>XPath</td>
</tr>
<tr>
<td>Xindice</td>
<td>Apache Software Foundation</td>
<td>Windows, Unix, and Linux</td>
<td>XPath, XUpdate</td>
</tr>
<tr>
<td>DBDOM</td>
<td>K. Ari Krupnikov</td>
<td>Windows, Unix, and Linux</td>
<td>XPath</td>
</tr>
<tr>
<td>DOM-Safe</td>
<td>Ellipsis</td>
<td>Windows, Unix, and Linux</td>
<td>XPath</td>
</tr>
<tr>
<td>GoXML DB</td>
<td>XML Global</td>
<td>Windows, Solaris, and Linux</td>
<td>XPath, XQuery</td>
</tr>
<tr>
<td>XIS</td>
<td>eXcelon Corp.</td>
<td>Windows, Solaris, HP Unix, and Linux</td>
<td>XPath, XQuery</td>
</tr>
<tr>
<td>Tamino</td>
<td>Software AG</td>
<td>Windows, Unix, and Linux</td>
<td>XPath, XQuery</td>
</tr>
<tr>
<td>TEXTML Server</td>
<td>IXIA, Inc.</td>
<td>Windows</td>
<td>XPath</td>
</tr>
<tr>
<td>Berkeley DB XML</td>
<td>Sleepycat Software</td>
<td>Windows, Solaris, and Linux</td>
<td>XPath</td>
</tr>
</tbody>
</table>

Table 3.1: Summary of native XML database products
Chapter 4

Standard Database Benchmarking Techniques

Database benchmarking is a test that uses a representative set of programs and data to evaluate the performance of databases in a given configuration. At present several benchmarks are available that can be used to evaluate different kinds of database systems.

4.1. Relational database system benchmark

The common database benchmarks are usually targeted for relational database usage. The TP1 benchmark [18] and the Wisconsin benchmark [19] focus on the relational model and transaction processing usages. TP1 is mainly designed to measure transaction throughput, and the Wisconsin benchmark is designed to measure the performance of a relational query processor.

4.1.1. TP1 benchmark

TP1 defines three basic benchmarks: sort, scan and debit credit. The first two benchmarks measure the system's input and output performance, whilst the last one measures the performance of the online transaction processing.

The sort benchmark defines the input in one million records. These records are stored in a sequential disc file. The first ten bytes of each record are unique values, and these key values are random. When creating an output file, the sort program attaches the relative input file sorted in key values to this output file. In terms of the sorted key values, the output files are also sequential. The benchmark includes two parameters: elapsed time and cost.
The scan benchmark is written in Cobol and can scan each record in the sequential file. It breaks the total scan into a set of mini-batch transactions, and each mini-batch transaction includes a thousand records. The scans can process in online systems, so they need to prevent concurrent access to the relative file. The relevant measures are elapsed time and cost.

Debit credit is a simple transaction processing application. The debit credit benchmark defines the Transaction Per Second (TPS). The relevant measures are throughput and cost.

4.1.2. Wisconsin benchmark

The Wisconsin benchmark presents a comprehensive set of queries that can be used for the performance evaluation of relational database management systems. These queries measure a set of different relational operations, such as selection, projection, joins, updates, and aggregate functions. The benchmark system runs in single user mode, and runs in a fixed sequential pattern. All tests were conducted on three conventional relational database systems and two database machines. Elapsed time is used as the main performance measure.

4.2. Object relational database system benchmark

Bucky [20] is a query benchmark for object relational database systems. The benchmark tests the object features that are offered by object relational systems. There are five key features to be included in this benchmark: row types with inheritance, inter-object references, set-valued attributes, methods of row objects, and ADT attributes. The benchmark uses object relational Bucky and a relational mapped simulation to evaluate the current object relational database systems.

4.3. Object oriented database system benchmark

Unlike TP1 and Wisconsin benchmarks, the Sun benchmark [21] presents a benchmark to measure response time. The Sun benchmark consists of a simple schema and a set of basic database operations. In this benchmark, a performance metric is used
to measure the response time of some simple database operations. The response time refers to the elapsed time between the start and completion of the operation. Another benchmark [22] is also designed to focus on some important characteristics of object oriented database systems. This benchmark is intended for engineering applications, so it has some different features. The database is remotely located, and the programming language must be mixed with data operations. The OO7 [23] benchmark provides a comprehensive profile of the performance of an OODBMS. It is more complex than the OO1 Benchmark. In particularly, OO7 has been designed to support multi-user operations.

4.4. TPC benchmarks

The TPC [24] defines a set of transaction processing and database benchmarks. These benchmarks measure the performance of database transaction processing systems. The main measure parameter is the number of transactions that are processed in the unit of time.

4.4.1. TPC-C

TPC-C simulates an online system environment where a number of users run database transactions against a database. TPC-C not only describes an order-entry system, but also is good for many wholesale activities that need to sell and distribute products. The TPC benchmark can be designed for scaling, and at the same time also satisfy consistent requirements.

TPC-C involves a mix of five different types of transactions. These transactions can be executed on-line or lined up for a deferred execution. The TPC-C performance metric measures the number of orders processed per minute. The three main parameters include the transaction rate, the associated price per transaction, and the availability date of the priced configuration.
4.4.2. TPC-H

The TPC-H is a decision support benchmark that includes a group of queries and concurrent data modifications. TPC-H not only examines volumes of data, but also runs these queries at a high level of complexity. The performance metric is TPC-H composite query-per-hour. There are some measured aspects: database size against executed queries, query processing power for single stream, and query throughput for multiple concurrent users.

4.4.3. TPC-R

TPC-R is also a business reporting and decision support benchmark. TPC-R is similar to TPC-H, except that TPC-R can utilise the pre-knowledge of the queries to provide additional optimisations. These optimisations can make a system run standard queries more rapidly.

4.4.4. TPC-W

TPC-W is a transaction web e-Commerce benchmark that provides pricing and performance information for database applications. There two models within TPC-W measure: business-to-consumer and business-to-business. The TPC-W calculates how many web interactions will be processed in one second.

4.4.5. Difference between these TPC benchmarks

TPC-C simulates an online environment where users can execute transactions on a database. It also includes a group of concurrent transactions of different types and levels of complexity. The TPC-C performance metric measures the number of orders processed per minute. TPC-H is a decision support benchmark that consists of a set of business-oriented queries and concurrent data modifications. The TPC-H performance metric reflects the capability of the queries processing system. TPC-R is similar to TPC-H, but it allows additional optimisations for running standard queries very rapidly. The TPC-W performance metric measures how many web interactions will be processed per second.
Chapter 5
XML Database System Benchmarks

5.1. XBench

XBench [25] is a benchmark system that has been developed by the University of Waterloo for XML applications. The workload of XBench is based on functionality and document class, and XBench queries contain almost all of the XQuery functionalities. The XBench document generator can generate documents for four different types of applications: data-centric and single document, data-centric and multiple documents, text-centric and single document, text-centric and multiple documents. XBench can also scale XML documents from 10MB to 10GB in size.

5.1.1. ToXgene

The ToXgene data generator [26] is used as an actual data generator in XBench. Unlike most XML data generators, ToXgene provides a template for generating XML documents. An advantage of this is that users can efficiently control document size and structure by setting some parameters within this template. ToXgene was developed in Java, and the current version can be executed on Windows and UNIX.

By using the template, ToXgene can easily generate almost all basic XML elements, and can utilise skewed distributions to set the number of occurrences of elements. Therefore, users can design any XML documents with a specified structure. ToXgene supports the sharing of elements, which makes the generation of correlated documents easier and can help to keep the integrity of documents. ToXgene also provides tox-lists for the storage of temporary data. At the time of XML document generation, the data in tox-lists can be reused as share data.
5.1.2. XBench queries

The workload of XBench [25] is based on functionality and document level. XBench queries covers almost of all XQuery's functionality and can be described by XQuery expression. These functionalities include exact match, function application, ordered access, quantifier, regular path expressions, sorting, document construction, irregular data, individual documents retrieving, text search, references, joins, and data type cast. XBench also designs the queries that are based on four different types of document: data-centric and single document, data-centric and multiple documents, text-centric and single document, text-centric and multiple documents.

5.2. XMark

XMark [27] provides an XML document generator named Xmlgen and designs 20 queries that cover most of XQuery’s functionalities. The benchmark uses Xmlgen to generate an XML document that contains the relevant information of the Internet auction site, and then executes the benchmark queries against the database. The experiments have been processed on an internal research prototype called Monet XML.

5.2.1. Xmlgen

Xmlgen [27] [28] can generate XML documents that model an Internet auction website. Xmlgen provides a template to set the number and type of elements, and makes probability distributions by setting the parameters. The texts are chosen from 17,000 words from Shakespeare's plays. Xmlgen also provides DTD and schema.

Xmlgen was developed in ANSI C and can be used on Windows, Solaris, and Linux. Xmlgen can generate large XML documents with only low memory requirements. The current version of Xmlgen requires less than 2MB of main memory. The normal XML document size is 100MB with a scaling factor of 1.0, and users can create larger documents by changing the scaling factor.
5.2.2. XMark queries

XMark [27] provides 20 queries that cover most of XQuery’s functionalities. These queries mainly contain simple relational queries, order preserving, navigational queries, aggregate, references, and sorting operations. All of the benchmark queries are formulated in XQuery.

5.3. XMach-1

XMach-1 [29] is a multi-user benchmark that was developed at the University of Leipzig in 2000. This benchmark models a web application. The system architecture includes the following components: an XML database, application servers, data loaders and browser clients. The application server runs on a web server, and the interfacing software can process XML documents and communicate with the database. Xmach-1 was developed in Java and the current version has been implemented on some XML database systems.

5.3.1. XMach-1 document generator

The XMach-1 document generator [29] can synthetically generate XML documents. The system consists of two parts: XML files and a data-centric directory. The text contents are chosen from 10,000 of the most frequently used words in English. Each of the XML files simulates an article, which includes title, chapter, section, paragraph, and so on. The directory system is based on the XML files, and mainly contains metadata of the other documents.

The XMach-1 supports both schema-based and schema-less variants of the benchmark. The file sizes vary from 2KB to 100KB, and the structures vary from flat to deep hierarchy. XMach-1 controls the size of the whole database by changing the number of XML files. There are four database sizes in XMach-1, which are created by varying the numbers in the files: 10000, 100000, 1000000 or 10000000. XMach-1 utilises an algorithm to generate documents with different depth and size. The process of generating documents can be controlled through setting parameters.
5.3.2. XMach-1 queries

XMach-1 [29] provides eight query and three update operations. The queries operations cover most of processing features: reconstruction of complex documents, full text retrieval, navigational queries, sorting and grouping operations, etc. Update operations mainly include the inserting and deleting of documents. The performance metric of XMach-1 is the throughput, which is measured in XML queries per second.

5.4. XOO7

XOO7 [30] is based on the well-established OO7 benchmark. Furthermore, XOO7 adds some new elements and queries for testing XML-specific features. XOO7 tests the XML processing capabilities in both data-centric and document-centric modes. The current version of XOO7 is based only on a single user model.

5.4.1. XOO7 document generator

The basic data structure of XOO7 [30] is similar to OO7, although XOO7 has translated the ER diagram into the corresponding DTD. Furthermore, XOO7 utilises the pre-processing of the inheritance of attributes and relationships to replace ISA relationships in the ER diagram. There is only one root name for each file in XOO7.

XOO7 can generate different-sized XML documents by setting appropriate values on a set of parameters. Like OO7, the sizes of these documents range from small to medium and large. XOO7 can change the document size by varying the depth and breadth of the document tree. Furthermore, XOO7 also can set different numbers of text files.

5.4.2. XOO7 queries

XOO7 [30] queries are relevant to the characteristics of the XML data and can cover most of functionalities. These queries mainly include data order preserving, text or keyword search, data sorting, navigational queries, and aggregate function, etc. All
queries are simple, and each query covers a few functionalities. A subset of queries can be chosen to test the required features in a specific application.
In contrast to the database benchmarking techniques of the previous two chapters, graph grammars are exploited to generate benchmark documents in this research. Prior to the substantial work of chapters 7 and 8, we depict some basic concepts and other definitions of graph grammars. A specific definition of graph grammars is then provided for this research. Finally, we compare graph grammars with DTDs and XML Schemas.

6.1. Definitions

A graph includes nodes and edges. Nodes can be regarded as objects, and edges can describe the relationships between objects. The graphs provide an understandable data representation. The whole structure of the graph can depict the hierarchical relationships between all the nodes. A new graph can be generated by using special rewriting rules to modify a host graph. A host graph is the original graph to which some rules are applied. A rewritten rule provides the special rule that can transfer a host graph into a preferred graph.

The transformation of graph structures can be represented through graph grammars. A graph grammar is specified by a group of graph rewriting rules, which are also called productions. A start graph provides the initial host graph for rule application, and all the nodes of the start graph are labelled as “terminal” or “non-terminal”. Rewriting rules are used to replace non-terminal nodes in the start graph. In general graph grammars, all rewriting rules are applied to the start graph until a terminal graph is generated. A terminal graph contains only terminal labels and no non-terminal labels. However, there are some differences in applying rewriting rules to the start graph. According to special requirements, suitable rewriting rules for generating a terminal graph can be designed.
There are different definitions of graph grammars in different applications. McCreary [31] provides a definition of graph grammar. The author defines the sub graph to be replaced as mother and defines the replacement graph as the daughter graph in a host graph. Formally, let the mother node be $u$. For each vertex $v$ in the daughter graph, $(w,v)$ is an edge in the resultant graph whenever $(w,u)$ is an edge in the host graph, and $(v,w)$ is an edge in the resultant graph whenever $(u,w)$ is an edge in the host graph.

Fahmy, Holt, and Mancoridis [32] utilise graph grammars to solve the software style repair problem. The authors specify a group of graph-rewrite rules. Each rule is used to replace one sub graph by another. The authors also specify each application must identify some pattern, and then transform the host graph in terms of that pattern.

Metayer [33] proposes utilising graph grammars to define software architectures. The author specifies that the nodes of the graph represent the individual entities and the edges correspond to the communication links between entities. An architecture style is a class of architectures that is characterised by a graph grammar. Formally, the author defines a graph grammar as $H = [NT, T, PR, AX]$. $NT$ and $T$ are sets of non-terminal and terminal symbols, $PR$ is a finite set of production rules, and $AX$ is an axiom.

Najarian [34] defines certain complexity-oriented characteristics of graph grammars as follows. A production $P$ (of a graph grammar $G$) is localised, if and only if the left hand side of $P$ has no disconnected components. A graph grammar $G$ is localised, if and only if all its productions are localised; otherwise, the grammar is termed non-localised. A production $P$ (of a graph grammar $G$) is deleting, if and only if the left hand side of $P$ has at least one edge that is not present in the right hand side of $P$. A graph grammar $G$ is deleting, if and only if at least one production of $G$ is deleting.

Derk and Debrunner [35] apply graph grammars to reconfiguration for fault tolerance. Reconfiguration Graph Grammars (RGG) is applied to the definition of processor array reconfiguration algorithms. The authors specify that the nodes of a graph are associated with the processors of a processor array and the edges are associated with those inter-processor communication lines. Formally, the authors define
Reconfiguration Graph Grammars as \(\text{RGG} = (S, A, a, N, \beta, P)\). \(S\) is a finite start graph representing the initial, non-faulty configuration of the processor array. \(A\) is a finite, non-empty set of alphabetic node labels, indicating the possible reconfiguration states of the nodes \(V\). \(a\) is a mapping from \(V\) to \(A\). \(N\) is a finite, nonempty set of numeric node labels corresponding to the logical coordinates of the active processors of \(S\). \(\beta\) is an injective partial mapping from \(V\) to \(N\). \(P\) is a finite, non-empty set of productions.

Baldwin and Chung [36] introduce a formal graph representation of design methodologies. The user specifies the objectives by supplying the initial graph. Graph modification uses a graph grammar called design process grammar. The authors specify that the non-terminal task nodes be replaced by sub graphs with less abstract tasks and intermediate specifications. The productions in a graph grammar permit the replacement of one sub graph by another. The authors define a production as \(P = (G_{\text{LHS}}, G_{\text{RHS}}, \sigma_{\text{in}}, \sigma_{\text{out}})\). \(G_{\text{LHS}}\) and \(G_{\text{RHS}}\) are process flow graphs for the left side and right side of the production, respectively. \(\sigma_{\text{in}}\) indicates the correspondence between input specifications. \(\sigma_{\text{out}}\) indicates the correspondence between output specifications.

6.2. Basic concepts

**Graph**: A data representation that consists of nodes and edges.

**Rewriting rule (or Production)**: A rule that replaces one non-terminal node by another graph.

**Terminal node**: An end node in a graph. Productions can never be applied to a terminal node. “X” is used to represent this.

**Non-terminal node**: An intermediate node in a graph. Productions can be applied to non-terminal nodes and replace the positions of non-terminal nodes in the graph. “O” is used to represent this.

**Host graph**: An original graph to which a rule is being applied.
**Start graph:** An initial host graph for the application of rules. All the nodes of the start graph are labelled as “terminal” or “non-terminal”.

**Replacement order:** A sequence that describes the replacement relationship of all productions.

**Intermediate graph:** A graph that includes terminal nodes and non-terminal nodes. An intermediate graph is created after the first production is applied to a start graph. Similarly, a new intermediate graph is generated when another production is applied to the former intermediate graph.

**Terminal graph:** A graph that contains only terminal nodes and no non-terminal nodes.

**Replacement rule:** A rule that describes how to separately apply every production to a start graph or intermediate graph until a terminal graph is generated. The sequence of all productions is described in replacement order. We rule that each production is used to replace the position of only one non-terminal node in the start graph or intermediate graph at a time. Similarly, the other productions in replacement order can be used one by one to replace the positions of the remainder of the non-terminal nodes in the intermediate graph.

**Single level production:** A production that consists of \( m \) (\( m \) is an integer) terminal nodes (\( X \)) and \( n \) (\( n \) is an integer) non-terminal nodes (\( O \)). The left-hand side of single level production is a non-terminal node. The right-hand side of single level production can be regarded as a one-level tree.

\[
O \longrightarrow X
\]

\[
X_1 \quad X_2 \quad \ldots \quad X_m \quad O_1 \quad O_2 \quad \ldots \quad O_n
\]

**Figure 6.1: Single level production**
On the right hand of single level production, X nodes and O nodes are derived from a common terminal node (X). Note the sequence of X nodes and O nodes is optional, since the user can generate different productions by selecting any order of all nodes.

For example, when setting different values on a pair of parameters m and n, we can get the following single level productions $P_0$, $P_1$, $P_2$, and $P_3$.

![Diagram of single level production]

Figure 6.2: Examples of single level production

The right-hand of the above productions can be represented as the following XML format. We insert a fixed string “X is a terminal node” or “O is a non-terminal node” as the element content.

$P_0$:  

```
<X> X is a terminal node </X>
```

$P_1$:  

```
<X> X is a terminal node
  <O> O is a non-terminal node </O>
</X>
```

$P_2$:  

```
<X> X is a terminal node
  <X> X is a terminal node </X>
  <O> O is a non-terminal node </O>
</X>
```

$P_3$:  

```
<X> X is a terminal node
  <O> O is a non-terminal node </O>
  <O> O is a non-terminal node </O>
</X>
```

**Multiple level production:** A production that consists of two or more single level productions. The left-hand side of multiple level production is also a non-terminal node. The right-hand side of multiple level production can be regarded as a multiple-levels tree.
A multiple level production can be generated by combining two or more single level productions according to a given replacement order. The right-hand side of the latter single level production replaces the position of one non-terminal node of the former production from top to bottom and from left to right. Similarly, the other single level production can be used to separately replace the remaining non-terminal nodes of the former production until a multiple level production is generated. The following examples of multiple level productions are considered.

The first example is a multiple level production with an unbalanced tree structure. We name it \( P_4 \).

\[
O \Rightarrow X
\]

\[
X_1 \quad X_2 \quad \ldots \quad X_m \quad X_1 \quad X_2 \quad \ldots \quad X_n
\]

\[
O_1 \quad O_2 \quad \ldots \quad O_n
\]

Figure 6.4: Multiple level production with unbalanced structure

The right hand of the above production can be represented by the following XML format. We insert a fixed string “X is a terminal node” or “O is a non-terminal node” as the element content.
The second example is a multiple level production with a balanced tree structure. We name it $P_5$.

![Image](p5.png)

**Figure 6.5: Multiple level production with balanced structure**

The right-hand of the above production can be represented by the following XML format. We insert a fixed string “X is a terminal node” or “O is a non-terminal node” as the element content.

```
<X> X is a terminal node
    <X> X is a terminal node </X>
    <X> X is a terminal node
        <X> X is a terminal node </X>
        <O> O is a non-terminal node </O>
</X>
</X>
```

**Final production:** An expression that is generated from a set of single level productions and multiple level productions in terms of a given replacement order. Its regular expression is $(P_1 \ast m_1, P_2 \ast m_2, \ldots, P_n \ast m_n)\ast k)$, where $P_n$ is either a single level production or a multiple level production. Final production can also be regarded as
a terminal graph, since it contains no non-terminal nodes that can be replaced by other productions. We can also utilise tree in data structure to depict a final production, and can transfer a final production into an XML format document.

In the regular expression of final production, the expression \((P_n^*m_m)\) is to call \(P_n\) for \(m_m\) times; for example, expression \((P_1^*3)\) represents \((P_1, P_1, P_1)\). The expression can be represented either as \((P_1 * m_1, P_2 * m_2, ..., P_n * m_m)\) or as \(((P_1, P_2, ..., P_n)^k)\); for example, the expression \((P_1^*3, P_3, P_4^*2)\) represents \((P_1, P_1, P_1, P_3, P_4, P_4)\), and the other expression \(((P_1, P_3, P_2)^2)\) represents \((P_1, P_3, P_2, P_1, P_3, P_2)\).

The regular expression of final production also means the replacement order of all production. The replacement of all production occurs according to this replacement order; for example, generation of the expression \((P_1, P_2, P_3)\) results in the following three steps. The intermediate graph is created after \(P_1\) is applied to the start graph. Then another intermediate graph is created after \(P_2\) is applied to the former intermediate graph generated by step one. Finally, the terminal graph is generated after \(P_3\) is applied to the former intermediate graph generated by step two.

6.3. Specific definition

Graph grammars can be defined as follows. First, a start graph is provided that has at least one non-terminal node. Second, we rule that the graph rewriting mechanism is to apply only one production to one non-terminal node in the start graph or intermediate graph at a time. Third, a graph grammar consists of a set of productions. The replacement order indicates the sequence relationship of all productions. An intermediate graph is generated after the first production is applied to the start graph. The latter productions are then applied to the intermediate graph one by one in replacement order. We only use one production at a time to replace one non-terminal node of the intermediate graph, until a terminal graph is generated. This process begins with the start graph, therefore, then generates several intermediate graphs, and finally results in a terminal graph.
More formally, a graph grammar is a system \( G = (S, P, R) \).

\( S \) is one of the non-terminal nodes in the start graph, and a rewriting rule is applied to this non-terminal node.

\( P \) is a pair of parameters \((V, T)\), where \( V \) is a vertex that represents one graph given on the left-hand side of the rule, and \( T \) is a tree that represents another graph given on the right-hand side of the rule.

\( R \) is the sequential graph rewriting mechanism, which refers to applying only one production to one non-terminal node in the graph at a time.

Based on the above definition of graph grammars, there are three steps to be undertaken. First, we choose a start graph, and at the same time we label the nodes of the start graph as terminal or non-terminal nodes. Second, we choose a set of productions and decide on the replacement order of all productions. Third, we apply these productions to the start graph or intermediate graphs according to the replacement order until a terminal graph is generated.

The following example, where “O” represents one non-terminal node and “X” represents one terminal node, is considered. The graph rewrite mechanism is to separately apply productions 1, 2 and 3 to only one non-terminal node of the start graph or intermediate graph at a time.

Start graph:

```
  X
 /|
/  |
X  O  O
```

Production 1: \( O \rightarrow X \)

```
  X
 /|
/  |
X  O
```

Production 2: \( O \rightarrow X \)

Production 3: \( O \rightarrow X \)

```
  X
```

Figure 6.6: Start graph and a set of productions
Applying rewriting rule 1 to the first non-terminal node of the start graph, we obtained the following intermediate graph:

![Intermediate graph one](image1)

Figure 6.7: Intermediate graph one

Next, applying rewriting rule 2 to one non-terminal node of the above intermediate graph, we obtained the following intermediate graph:

![Intermediate graph two](image2)

Figure 6.8: Intermediate graph two

Finally, applying rewriting rule 3 to one non-terminal node of the above intermediate graph, we obtained a terminal graph:

![Terminal graph](image3)

Figure 6.9: Terminal graph
6.4. DTD, XML Schemas and graph grammars

DTD and XML Schemas both define the structure of an XML document through a list of legal elements. XML Schemas are written in XML. We utilise graph grammars to generate a final production. A final production can be transferred into an XML format document and can clearly reflect the whole tree structure of an XML document. There are two main reasons to use graph grammars to generate XML documents.

One reason is that this method can adjust the depth and width of a final production. A final production can be regarded as a tree. Not only we clearly know the positions of all the nodes in the tree, but we can also control its structure and size by adjusting the depth or width of the tree. We can adjust the depth of a tree by varying the times of repetition in every production or by designing multiple level production. We can also adjust the width of a tree by increasing or decreasing the total number of nodes in each production, and then applying these productions in order to generate a final production. For example, by repeating production 1 (in figure 6.6) three times, we can generate new a production that has three levels. By increasing the total number of nodes in production 1 (in figure 6.6) to five, we can increase its width. Utilising this method in every production that will be applied in the final production, we can easily control its whole structure and size. However, DTD and XML Schemas cannot do this easily.

Production 1: \[ O \rightarrow X \]

Increase the depth: \[ O \rightarrow X \]

Increase the width: \[ O \rightarrow X \]

Figure 6.10: Control of the depth and width
Another reason for using graph grammars is that productions have reusability. Every production can be regarded as a separate component, and one production can be considered as a basic unit for constructing the final production. By locating the positions of the non-terminal nodes in the current productions and designing more new productions, we can use replacement rules to generate new final productions to meet more user requirements. When utilising graph grammars to generate any other new final productions, we can reuse existing productions. We can choose one or more productions as part of the next input for generating new final productions, and we do not even need to modify them. For example, we can reuse production 3 (shown earlier in figure 6.6) to generate another new terminal graph without modifying it, as shown below. Production 3 (in figure 6.6) is applied to the start graph, then the non-terminal node is replaced in the start graph, and so a new terminal graph is generated. However, DTD has no good reusability and extensibility. On the other hand, XML Schemas are extensible because they are written in XML.

Production 3: \[ O \rightarrow X \]

Start graph: \[ O \rightarrow X \]

Terminal graph: \[ O \rightarrow X \]

![Diagram of Productions](image)

**Figure 6.11: Reusability of productions**

Based on the above discussions, we conclude that graph grammars are easier than DTD and XML Schemas in controlling the structure and size of XML documents. Furthermore, graph grammars have good reusability and extensibility, whereas DTD does not. Therefore, we utilise graph grammars to generate XML documents.
In this chapter, we utilise graph grammars to generate benchmark documents. A graph description of the document generation process is given first. Then we use Backus-Naur Form (BNF) notation for the generation of XML documents. Also, an algorithm for a XML document generator is provided.

7.1. Description of graphs

In this section, we describe how to use graphs to generate a specified final production step by step. We use $P_0$ to represent this final production.

![Diagram of graph]

Figure 7.1: An example of a final production
Step one

The final production is broken down into a set of single level productions and one multiple level production. The final production is represented through the regular expression \((P_1, P_1, P_2, P_3, P_4)\), where \(P_1, P_2\) and \(P_4\) are single level productions and \(P_3\) is a multiple level production. The multiple level production \(P_3\) is also broken down into two single level productions \(P_5\) and \(P_6\).

![Figure 7.2: A group of productions](image)

Step two

We generate multiple level production \(P_3\) by applying single level production \(P_6\) to one non-terminal node of another single level production \(P_5\). In this example, the replacement order is \((P_5, P_6)\).

![Figure 7.3: Generation of multiple level production](image)
Step three

The final production is generated according to the specified replacement rule and the regular expression of final production. The process is depicted as shown below. In this example, the replacement order is \((P_1, P_1, P_2, P_3, P_4)\).

![Diagram of final production generation](image)

**Figure 7.4: Generation of a final production**

### 7.2. BNF notation for document generator

In this section, we use Backus-Naur Form (BNF) notation \([37]\) as syntax for the generation of XML documents.
The meta-symbols of BNF are:
- `::=` means "is defined as".
- `|` means "or".
- `< >` means angle brackets used to surround category names.

We introduce some slight extensions of BNF:
- `;` means the end of this comment.
- `[ ]` means that optional items are enclosed in meta symbols.
- `""` means terminals of a character or string are surrounded by quotes (""") to distinguish them from meta-symbols.
- `*` preceding an element indicates repetition of this element. The full form `1*m <element>` indicates this element occurs at least one and at most m times. Default values are 0 and infinity so that "* <element>" allows any number of elements. "1* <element>" requires at least one element, and "1*2 <element>" allows one or two elements.
- `#` preceding productions indicates recursive call of these productions. The full form `#m <productions>` can be regarded as (`<P_1>`, `<P_2>`, ..., `<P_m>`). It indicates that recursive call of these productions occurs m times according to their sequence. That is, the latter production replaces the position of the non-terminal node of the former production until the last production is applied. The default value is one so that "# <productions>" allows only one production. "#3 <productions>" means that recursion of three productions occurs.

Based on the above BNF notations and their extensions, the elements shown below can be described. Following the example above, BNF notations are also used to describe the relative productions.

1) `<CHAR> ::= "a"|"b"|"c"|"d"|"e"|"f"|"g"|"h"|"i"|"j"|"k"|"l"|"m"|"n"|"o"|"p"|"q"|"r"|"s"|"t"|"u"|"v"|"w"|"x"|"y"|"z"|"A"|"B"|"C"|"D"|"E"|"F"|"G"|"H"|"I"|"J"|"K"|"L"|"M"|"N"|"O"|"P"|"Q"|"R"|"S"|"T"|"U"|"V"|"W"|"X"|"Y"|"Z" ;

CHAR is an uppercase letter or lowercase letter.
2) `<DIGIT>` ::= "0"|"1"|"2"|"3"|"4"|"5"|"6"|"7"|"8"|"9" ;

`DIGIT` is a digit number.

3) `<Tag>` ::= 
   (1*m `<CHAR>` | 1*m `<DIGIT>` | (1*m `<CHAR>` 1*m `<DIGIT>`)) ;

`Tag` consists of a string of characters and/or a set of digit numbers. For example, we use “X₁₁” to describe a terminal node and “O₁₂” to describe a non-terminal node.

4) `<Text>` ::= 
   (0*m `<CHAR>` | 0*m `<DIGIT>` | (0*m `<CHAR>` 0*m `<DIGIT>`)) ;

`Text` consists of a string of zero or more characters and/or a set of digit numbers. If the element content has no any digit numbers or characters, the value of the element is null. For example, we use “X is a terminal node” as an element content.

5) `<ID>` ::= (1*m `<CHAR>`) "=" (1*m `<DIGIT>`) " ;

`ID` consists of at least one and at most m digit numbers. ID is a unique identifier. For example, ID ::= “id = “11” “.

6) `<Attribute>` ::= 
   "@" (1*m `<CHAR>`) "=" (1*m `<CHAR>` | 1*m `<DIGIT>` | (1*m `<CHAR>` 1*m `<DIGIT>`)) " ;

`Attribute` consists of a string of characters and/or a set of digit numbers. For example, Attribute ::= “@age = “23” “.

7) `<Terminal node>` ::= 
   "<X" `<Tag>` [ `<ID>` ] [1*m `<Attribute>` ] "=" 
   [ `<Text>` ] 
   "</X" `<Tag>` " ;
Terminal node consists of a couple of tags, ID, attributes and text. The character “X” preceding tag indicates terminal node. ID, attribute and text are optional. For example,

```
<Terminal node> ::= 
  "<X1 ID = "11" @age= "23" >"
  "X is a terminal node"
  "</X1>" ;
```

8) `<Non-terminal node> ::= 

( "<O" <Tag> [<ID>] [1*m <Attribute>] ""> 
  [<Text>] 
  "</O" <Tag> ""> 
) | (<Single level production>) | (<Multiple level production>) ;

Terminal node consists of a string of descriptions or single level production or multiple level production. A string of descriptions includes a couple of tags, ID, attributes and text. The character “O” preceding tag indicates terminal node. ID, attribute and text are optional. For example,

```
<Non-terminal node> ::= 
  "<O1 ID = "22" @age= "32" >"
  "O is a non-terminal node"
  "</O1>" ;
```

9) `<Single level production> ::= 

  "<" <Tag> "">" 
  (1*m <Terminal node>) | (1*m <Non-terminal node>) | (1*m <Terminal node> 1*m <Non-terminal node>) 
  "<" <Tag> "">" ;

Single level production consists of a couple of tags and at least one and at most 
m terminal nodes and/or non-terminal nodes. The width of a single level production can
be adjusted by changing the value of m. For example, this single level production consists of one terminal node and one non-terminal node,

<Single level production> ::=  
   "<X>"  
   1*1 <Terminal node>  
   1*1 <Non-terminal node>  
   "</X>" ;

10) <Multiple level production> ::=  
   "<" <Tag> "">"  
   (#m <Single level production>)  
   "</" <Tag> "">" ;

Multiple level production consists of a couple of tags and a set of single level productions. "#m <Single level production>" can be represented by (<P₁>, <P₂>, <P₃>, ..., <Pₘ>), where P is single level production. We can adjust the depth of multiple level production by changing the value of m. The latter production replaces the position of the non-terminal node of the former production until the last production is applied. We generate a new production after P₂ replaces the first non-terminal node of P₁. P₃ replaces one non-terminal node of the resultant production in the last step, and then another new production is generated. Similarly, we apply the remaining productions until Pₘ is applied. For example, P₁, P₂, and P₃ are single level productions,

<Multiple level production> ::=  
   "<X>"  
   (#3 (<P₁>, <P₂>, <P₃>))  
   "</X>" ;

11) <Final production> ::=  
   "<" <Tag> "">"  
   (#m (<Single level production> | <Multiple level production | (<Single level production> <Multiple level production>)))  
   "</" <Tag> "">" ;
Final production consists of a couple of tags, and a set of single level productions and/or multiple level productions. “#m <productions>” can be represented by (<P1>, <P2>, <P3>, ..., <Pm>), where P is single level production and/or multiple level production. We can adjust the depth of the final production by changing the value of m. The latter production replaces the position of the non-terminal node of the former production until the last production Pm is applied. For example, P1, P2, and P4 are single level productions, and P3 is a multiple level production,

\[
\text{<Final production> ::= "<X>"}
\]

\[
(\#m (<P1>, <P2>, <P3>, <P4>))
\]

\["</X>" ;
\]

7.3. Algorithm of document generator

We now provide the algorithm for generating XML documents according to the specified document structure and size. A group of productions and the given replacement order to control the structure of XML document tree are utilised. We control document size by increasing or decreasing the size of each document, and also by changing the number of single level productions and multiple level productions. There are several steps as follows:

a) According to the special requirements concerning XML document structure, we use a final production to depict this XML document.

b) We break down this final production into a set of special single level productions and/or multiple level productions. They are the basic components for generating an XML document. The regular expression of the final production is \((P_n * m_1, P_2 * m_2, ..., P_n * m_n)\), where Pn is either a single level production or multiple level production. \((P_n * m_n)\) can also be represented as \((P_n, P_n, ..., P_n)\) that repeats \(P_n m_n\) times.

c) We decide the replacement order that represents the replacement sequence of all productions is decided on. The replacement order is represented as \((P_1, P_2, ..., P_n)\). The replacement order is important for an XML document, because
it reflects the tree structure of that XML document. Productions with a different replacement order will generate XML documents with a different structure.

d) According to the special requirements concerning XML document size, we decide the size of every production is decided on. We use XML format to express every production in the replacement order. Then their element, element content, ID, and attributes are also decided on.

e) The final XML document is generated according to the replacement rule and a group of XML documents. The replacement rule is defined as a rule where the latter production replaces the position of one non-terminal node in the former production from top to bottom and from left to right until the last production is applied. The ID and IDRefs are added to final XML document.

In the first section of this chapter, we have described how to use graphs to generate the specified final production $P_0$. Next, we give an example of how to utilise BNF notation to describe the generation of the final production $P_0$.

**Step one**

In the following descriptions, we use “X” or “$X_i$” as the name of the element, and omit ID and attribute, and use “X is a terminal node” as the element content.

We depict final production ($P_0$) as follows:

$$
\begin{align*}
<P_0> := &
\text{"<X>"} \\
& (\#5 \ (<P_1>, \ <P_1>, \ <P_2>, \ <P_3>, \ <P_4>)) \\
& \text{"</X>" ;}
\end{align*}
$$

We depict single level productions ($P_1, P_2, P_4$) as follows:

$$
\begin{align*}
<P_1> := &
\text{"<X>"} \\
& \text{"<X_1>"} \\
& \text{"X is a terminal node"} \\
& \text{"</X_1>"} \\
& <\text{Single level production}> \\
& \text{"</X>" ;}
\end{align*}
$$
Multiple level production $P_3$ consists of two single level productions $P_5$ and $P_6$. We depict multiple level production ($P_3$) and single level productions ($P_5$, $P_6$) as follows:

Step four

We apply single level productions ($P_5$ and $P_6$) to multiple level production $P_3$ according to the replacement rule.
Step five

We apply single level productions ($P_1$, $P_2$ and $P_4$) and multiple level production $P_3$ to final production ($P_0$) according to the replacement rule.

$P_0 ::=$

"<X>"

"<X>"

"<X>

"X is a terminal node"

"</X>"

"<X>

"<X>"

"X is a terminal node"

"</X>"

"<X>"

"<X>"

"<X>"

"X is a terminal node"

"</X>"

"<X>"

"<X>"

"X is a terminal node"

"</X>"

"<X>"

"<X>"

"<X>"

"X is a terminal node"

"</X>"

"<X>"

"<X>"

"X is a terminal node"

"</X>"

"<X>"

"<X>"

"X is a terminal node"

"</X>"

"<X>"

"<X>"

"<X>"

"X is a terminal node"

"</X>";
Step six

Finally, we transfer final production ($P_0$) to its relative XML document. And then we add ID and IDRefs to the final XML document ($T_{111}$ refer to $T_{11212}$).

```
<X>
  <X id = "T_{111}"> X is a terminal node </X>
</X>

<X>
  <X id = "T_{11212}" X = "T_{111}"> X is a terminal node </X>
</X>

<X>
  <X id = "T_{111}"> X is a terminal node </X>
</X>

<X>
  <X id = "T_{11212}" X = "T_{111}"> X is a terminal node </X>
</X>

<X>
  <X id = "T_{111}"> X is a terminal node </X>
</X>

<X>
  <X id = "T_{11212}" X = "T_{111}"> X is a terminal node </X>
</X>

<X>
  <X id = "T_{111}"> X is a terminal node </X>
</X>

<X>
  <X id = "T_{11212}" X = "T_{111}"> X is a terminal node </X>
</X>

<X>
  <X id = "T_{111}"> X is a terminal node </X>
</X>

<X>
  <X id = "T_{11212}" X = "T_{111}"> X is a terminal node </X>
</X>

<X>
  <X id = "T_{111}"> X is a terminal node </X>
</X>

<X>
  <X id = "T_{11212}" X = "T_{111}"> X is a terminal node </X>
</X>
```

According to this algorithm, we can utilise an XSLT stylesheet to implement an XML document generator. We will discuss this further in Chapter 9. In the next chapter we will examine how to generate benchmark queries.
Chapter 8
Benchmark Queries

In this chapter, we depict automatic benchmark query generator in terms of specific patterns. We discuss some basic concepts and then provide patterns and instances of benchmark queries. Also, a benchmark query generator algorithm is derived.

8.1. Basic concepts

Start node is the node at the beginning of the path. End node is the node at the end of the path. Level represents the different depths of the tree. For example, the level is one, if the locator moves from a parent node to one of the child nodes. The value of the level must obviously be less than or equal to the maximal depth of the tree at any time. First Level or second Level is the value of the level. The node whose level is equal to the first level is the start of the path, and the node whose level is equal to the second level is the end of the path.

In terms of these different combinations of parameters, there are four types of path models: Path (first level, second level), Path (first level, end node), Path (start node, end node), and Path (start node, second level).

![Figure 8.1: Path models](image-url)
Given any one of the above combinations of parameters, we can generate several paths that satisfy the requirements of this pair of parameters, and obtain all the nodes on these paths. We can then choose one preferred path from all these paths. The four types of paths are now described:

a) Given first level and second level, some paths can be generated that satisfy the requirements of first level and second level. We can then choose one preferred path from these paths, and name it: **Path (first level, second level)**.

b) Given first level and end node, some paths can be generated that satisfy the requirements of first level and end node. We can then choose one preferred path from these paths, and name it: **Path (first level, end node)**.

c) Given start node and end node, some paths can be generated that run from start node to end node. We can then choose one preferred path from these paths, and name it: **Path (start node, end node)**. For example, given start node \(P_{1X_1}\) and end node \(P_{3X_1}\), we obtain path \((P_{1X_1}, P_{3X_1})\).

![Figure 8.2: Path \((P_{1X_1}, P_{3X_1})\)](image)

d) Given a start node, an end node, and a path, another new path is generated that includes the given start node, end node and path. If all the nodes on this new path must abide by the rule that the former node is an ancestor of the latter node and the latter node is a descendant of the former node, we can call it **Composite path**. This composite path can be expressed in the following different ways:
Composite path (start node, Path, end node)
Composite path (Path, end node)
Composite path (start node, Path)

For example, given start node \((P_1X_1)\) and end node \((P_6X_1)\), and a Path \((P_3X_1, P_4X_1, P_5X_1)\), we can then generate a Composite path \((P_1X_1, \text{Path}, P_6X_1)\).

Given start node and second level, we can generate some paths that satisfy the requirements of start node and second level. We can then choose one preferred path from these paths, and name it: **Path (start node, second level)**. For example, Path \((P_1X_2, \text{second level} = 3)\), it generates five paths that satisfy the rules. We can also randomly pick up a preferred one (Path \((P_1X_2, P_7X_1))\) from these five paths.
Given a path and another value of level, the path can be extended by a value of level from the last node of the former path. We call it **Composite path** (Path, another level).

For example, we obtain a Composite path (Path (P₁X₂, second level = 3), another level = 2).

![Figure 8.5: Composite path (Path, 2)](image)

### 8.2. Classification of queries

In this section, queries are classified by their functionality. The examples of these benchmark queries given below are based on the following XML file named “file1.xml”. These examples were written and generated in XQuery [38].

```xml
<?xml version="1.0" ?>
<Australia>
  <NSW>
    <Wollongong>
      <Student id="P₁X₁" Teacher="P₁X₂" name="yang" age="30">
        yang is a student
        <Friend age="35" name="yuyang"> he is my friend
        </Friend>
      </Student>
    </Wollongong>
  </NSW>
  <ACT>
    <Canberra>
      <Teacher id="P₁X₂" Student="P₁X₃" age="28" year="1998">
        He is a teacher
        <Brother>
          <Friend> He is my friend
          </Friend>
          <Sister age="35"> she is a girl
        </Sister>
      </Teacher>
    </Canberra>
  </ACT>
</Australia>
```
Exact Match

This query requires the string to match exactly with some specified path expressions. It tests a database’s ability to deal with a search with a fully specified path. For example, the following query can return the text of the tag that has matching id value “P3X1”.

```
for $A_1$ in document ("file1.xml") / / NSW / Wollongong / Student [@id="P3X1"]
return $A_1 / text ()
```

Function Application

This query tests a database’s ability to handle aggregate functions such as count, avg, max, min and sum. For example, this query can find the tag by attribute age and separately calculate maximal age, minimal age, and the average age.

```
for $A_1$ in document ("file1.xml") / / NSW / Wollongong / Student / @age,
$A_2$ in document ("file1.xml") / / ACT / Canberra / Teacher / @age,
```
$A_3$ in document ("file1.xml") / / NSW / Sydney / Student / @age
return
<Output>
  <Max> max ($A_1$, $A_2$, $A_3$) </Max>
  <Min> min ($A_1$, $A_2$, $A_3$) </Min>
  <Ave> sum ($A_1$, $A_2$, $A_3$) / count ($A_1$, $A_2$, $A_3$) </Ave>
</Output>

**Order Access**

In XML documents, the order of the elements is important. The query should test how efficiently the DBMS can be exploited to handle queries with order constraints. For example, this query can find the text based on the current matching position whose attribute age is greater than another specified node’s attribute age.

for $A_1$ in document ("file1.xml") / / NSW / Wollongong / Student,
  $A_2$ in document ("file1.xml") / / ACT / Canberra / Teacher
where $A_2$ / Brother / Friend / Sister / @age gt $A_1$ / @age
return
  $A_2$ / Brother / Friend / Sister / text ()

**Regular Path Expressions**

This query investigates how well the query processor can optimise path expressions. For example, this query can return all text of the tag whose children and descendant nodes have matching id attribute value “P3X1”.

for $A_1$ in document ("file1.xml") / / NSW / Wollongong / Student [@id="P3X1"]
return
  <Output>
    $A_1$ / * / text (), $A_1$ / / text ()
  </Output>
Sorting

This query investigates how well the systems sort values by a given element. For example, this query can list the text and attribute name of the tag whose attribute age is greater than 20. The results are ordered by name.

```
for $A_3$ in document ("file1.xml") / / NSW / Sydney / Student
order by $A_3$ / * / @name
where $A_3$ / @age gt "20"
return
  <Output>
    $A_3$ / * / text (),
    $A_3$ / * / @name
  </Output>
```

Null Values

This query tests the ability of query processors to handle null values. For example, this query can list the attribute names of the tags whose text is empty under a specified path, whilst at the same time other tags also satisfy the specified conditions. A1’s attribute age is greater than 25, and A2’s attribute year is less than 2001.

```
for $A_1$ in document ("file1.xml") / / NSW / Wollongong / Student,
  $A_2$ in document ("file1.xml") / / ACT / Canberra / Teacher,
  $A_3$ in document ("file1.xml") / / NSW / Sydney / Student
where empty ($A_3$ / Friend / Teacher / Brother / text ())
  and ($A_1$ / @age gt "25") and ($A_2$ / @year lt "2001")
return
  $A_3$ / @name
```

Text Search

The systems should be able to handle how to search a sub string from a specified position. For example, this query can find a tag whose attribute name includes a specified string “yu”, and then return this tag’s text.
Data Type Cast

The element values in XML documents are in string type, but sometimes the queries might have to cast one data type to another. For example, this query can find the tags based on the current matching position, and then return the text of those tags where the sum of their ages is greater than 50.

Join On Values

XML documents usually need to identify the relationship between related data. Users sometimes wish to combine together separate information by using join on values. For example, this query can find the tags based on the current matching position, and then return the text of those tags whose attribute ages satisfy the special conditions. $A_1$’s age is greater than $A_2$’s age, and $A_2$’s age is less than $A_3$’s age.
References

References can allow more complex relationships than hierarchical element structures. A good query optimiser should take advantage of references in XML documents. For example, this query can return the text of the tags that have matching id attribute value “P3X1”, “P3X2” and “P3X3”, where one tag has the same attribute name value as another tag. In this file, P3X1 refers to P3X2, and P3X2 refers to P3X3.

```xml
for $A_1$ in document ("file1.xml")/ /NSW / Wollongong/Student [@id="P3X1"],
   $A_2$ in document ("file1.xml") / / ACT / Canberra / Teacher [@id="P3X2"],
   $A_3$ in document ("file1.xml") / / NSW / Sydney / Student [@id="P3X3"]
where $A_1$ / @name = $A_3$ / @name
return $A_1$ / text (),$A_2$ / text (),$A_3$ / text ()
```

8.3. Patterns and instances of benchmark queries

In this section, we design different patterns for these benchmark queries. The users can input a set of parameters for generating a benchmark query with a special function. Instances written in XQuery [38] are given for each kind of benchmark query.

According to the discussion above (in section 8.1.), there are four different types of path models in terms of the user input. In the following patterns, we presume the users specify two parameters that are first level and second level. We also express this path as Path (first level, second level).

For generating some types of benchmark queries, we sometimes need to refer to ID values, order name and attributes in XML documents. So we require the system to automatically insert some fixed ID values, order name, and attribute names and values in specified locations in XML documents. The program can also easily position and capture these fixed ID values, order name and attributes in XML documents.

Appendix A contains more details of benchmark query patterns and instances.
8.4. Algorithm of benchmark query generator

We provide an algorithm to generate benchmark queries.

a) We decide to generate some or all benchmark queries from those ten queries, and then break down these queries into some parts that are individually called for, let, where, order by, and return. The expressions of let, where, and order by are optional.

b) The user inputs a set of parameters for every benchmark query. These parameters mainly include nesting level, names of files, relationship, first level, and second level. Comparison operators are optional.

c) We use first level and second level to generate every path, until all the paths in the benchmark queries are generated. We also need to insert order name and some IDs, IDRefs, and attributes in specified locations of the XML document.

d) We generate all benchmark queries by applying these input parameters to the patterns of benchmark queries.
Based on the above discussion, the final production can be expressed by the following regular expression: 

\[(P_1 \times m_1, P_2 \times m_2, \ldots, P_n \times m_m) \times k\),

where \(P_1\), or \(P_2\), …, or \(P_n\) may be single level production or multiple level production.

The implementation consists of three parts. The first step is to separately use each production several times, thus obtain a group of separate productions such as 

\[(P_1 \times m_1), (P_2 \times m_2), \ldots, (P_n \times m_m)\). The expression \((P_n \times m_m)\) is to call \(P_n\) for \(m_m\) times. The second step is to combine all the productions obtained from the first step into a whole production according to a fixed production order, and thus obtain a production such as 

\[(P_1 \times m_1, P_2 \times m_2, \ldots, P_n \times m_m)\). The last step is to reuse the same method as the first step to use the production obtained from step two several times, and then result in a production such as 

\[((P_1 \times m_1, P_2 \times m_2, \ldots, P_n \times m_m) \times k)\).

The flow chart below gives more details about how to implement the final production.

**Step One:**

To choose a basic production as the initial value, a special XSLT stylesheet is used to process it. Then, if the execution times are less than or equal to a fixed value \(m\),
the output will be processed again by the XSLT stylesheet as the next input; otherwise, the output is (production * m). By using this special XSLT stylesheet to process a group of different basic productions several times, the output generates a group of separate productions (P₁ * m₁), (P₂ * m₂), …, and (Pₙ * mₙ).

The following is the XSLT Stylesheet for step one:

(XSLTSefCopy.xsl)

```xml
<?xml version="1.0"?>
<xsl:stylesheet version="1.0"
    xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
    <xsl:output method="xml" indent="yes" encoding="UTF-8"/>

    <xsl:template match="/ | * | @*">
        <xsl:copy>
            <xsl:apply-templates select="* | @* | text()"/>
        </xsl:copy>
    </xsl:template>

    <xsl:template match="O">
        <xsl:copy-of select="/"/>
    </xsl:template>

</xsl:stylesheet>
```

Whether it is a single level production or a multiple level production, an XSLT engine can transform them into another production by using a given XSLT stylesheet. According to the definition of multiple level production, the latter production will replace one non-terminal node in the former production from top to bottom and from left to right.

![Figure 9.2: An example of self-copy](output.xml)

73
So we execute the following command:

```
msxsl XMLSelfCopy.xml XSLTStylesheet.xsl –o Output.xml
```

(XMLSelfCopy.xml)

```xml
<?xml version="1.0" encoding="UTF-8"?>
<X>
  <X id="X1">X1 Element Content</X>
  <O id="O1">O1 Element Content</O>
</X>
```

(Output.xml)

```xml
<?xml version="1.0" encoding="UTF-8"?>
<X>
  <X id="X1">X1 Element Content</X>
  <X>
    <X id="X1">X1 Element Content</X>
    <O id="O1">O1 Element Content</O>
  </X>
</X>
```

**Step Two:**

```
(P1 * m1) → Combine Function → (P1 * m1, P2 * m2, ..., Pn * mn)
```

* Figure 9.3: Combine function

In order to combine a group of separate productions into a whole production, another special XSLT stylesheet is used several times. The following flow chart expresses this process:
To choose \((P_1 * m_1)\) and \((P_2 * m_2)\) as the initial value, a special XSLT stylesheet is used to process them. And then the output \((P_1 * m_1, P_2 * m_2)\) and \((P_3 * m_3)\) will be processed again by the XSLT stylesheet as the next input. Based on this method, the output generates a special production \((P_1 * m_1, P_2 * m_2, ..., P_n * m_m)\) by using the special XSLT Stylesheet.

An XSLT Stylesheet for step two is shown below:

```
<?xml version="1.0"?>
<xsl:stylesheet version="1.0"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
  <xsl:output method="xml" indent="yes" encoding="UTF-8"/>

  <xsl:template match="/ | * | @*">
    <xsl:copy>
      <xsl:apply-templates select="* | @* | text()"/>
    </xsl:copy>
  </xsl:template>

  <xsl:template match="O">
    <xsl:copy-of select="document('added.xml')/*/ @*"/>
  </xsl:template>

</xsl:stylesheet>
```

Similarly, according to the definition of multiple level productions, the latter production replaces one non-terminal node in the former production from top to bottom and from left to right. For example,
So we execute the following command:

`msxsl Original.xml XSLTCombine.xsl – o Output.xml`

(Original.xml)
```xml
<?xml version="1.0" encoding="UTF-8"?>
<X>
  <X id="X_1">X_1 Element Content</X>
  <O id="O_1">O_1 Element Content</O>
</X>
```

(Added.xml)
```xml
<?xml version="1.0" encoding="UTF-8"?>
<X>
  <X id="X_1">X_1 Element Content</X>
  <X>
    <X id="X_1">X_1 Element Content</X>
    <O id="O_1">O_1 Element Content</O>
    <O id="O_2">O_2 Element Content</O>
  </X>
</X>
```

(Output.xml)
```xml
<X>
  <X id="X_1">X_1 Element Content</X>
  <X>
    <X id="X_1">X_1 Element Content</X>
    <O id="O_1">O_1 Element Content</O>
    <O id="O_2">O_2 Element Content</O>
  </X>
</X>
```
Step Three:

\[(P_1 \times m_1, P_2 \times m_2, \ldots, P_n \times m_n)\]

\[\times ((P_1 \times m_1, P_2 \times m_2, \ldots, P_n \times m_n) \times k)\]

By reusing the same XSLT stylesheet as in step one, the output generates a final production \([(P_1 \times m_1, P_2 \times m_2, \ldots, P_n \times m_n) \times k]\).

There are two main advantages to using the above method to generate XML documents. First, based on these three steps, the user can individually choose any basic productions and obtain a final production with a specified document structure. Second, the user can easily control the size of XML documents by changing the size of the basic productions or by using the given production more times.
Chapter 10

Implementation of Benchmark Query Generator

In this chapter, we show how to implement a benchmark query generator. First we generate a relational table that contains node information. We derive algorithms to generate paths, and then apply these paths to automatically generate the patterns of benchmark queries.

10.1. Generation of information table

In the process of using the MyBench XML document generator to generate XML documents, we also generate its corresponding information table. This table contains useful information on every element node in an XML document, with one record corresponding to one element node. The table will be helpful to quickly generate the paths of benchmark queries.

We now create a new relational table. This table includes three fields: Tagname, Level, and Path. The field Tagname is designed to store the tag names of all the element nodes in an XML document. The field Level is designed to store the depth value of the element node, and the field Path is a string that includes a group of tag names. One record is the tag name sequence of all the ancestors of the current element node. A backslash “/” is used to separate these tag names. By using the string of the path, the program can easily find an ancestor with any specified depth value. For example, “T_{11211}” is a record of Tagname, “4” is a record of Level, and “T_{1} / T_{11} / T_{112} / T_{1121}” is a record of Path. We also create two indices on column Tagname and Level individually.
CREATE TABLE XMLtoTable
{
  Tagname    CHAR (21),
  Level      NUMERIC (3),
  Path       CHAR (500)
}

CREATE INDEX Level_index ON XMLtoTable (Level);
CREATE INDEX Tagname_index ON XMLtoTable (Tagname);

In terms of the structure of this table, when generating an XML document, the following steps are used to generate a relational table, which includes information about all the element nodes in the XML document.

First, while the user inputs a group of small individual XML documents, we also create a group of relational tables, which correspond to these small XML documents. For example, there are two productions $P_1$ and $P_2$.

$$P_1: O \rightarrow X$$

![Figure 10.1: Production one](image)

Its XML format appears as follows:

```xml
<T_1>
  <T_{11} />
  <T_{12} />
</T_1>
```
We store the values of the tag name, level and path in the table XMLtoTable_P₁.

<table>
<thead>
<tr>
<th>Tagname</th>
<th>Level</th>
<th>Path</th>
</tr>
</thead>
<tbody>
<tr>
<td>T₁</td>
<td>0</td>
<td>null</td>
</tr>
<tr>
<td>T₁₁</td>
<td>1</td>
<td>T₁ /</td>
</tr>
<tr>
<td>T₁₂</td>
<td>1</td>
<td>T₁ /</td>
</tr>
</tbody>
</table>

Table 10.2: XMLtoTable_P₁

![Production two](image)

Figure 10.2: Production two

Its XML format appears as follows:

```xml
<T₁>
  <T₁₁ />
  <T₁₂>
    <T₁₂₁ />
    <T₁₂₂ />
  </T₁₂>
</T₁>
```

We store the values of the tag name, level and path in the table XMLtoTable_P₂.

<table>
<thead>
<tr>
<th>Tagname</th>
<th>Level</th>
<th>Path</th>
</tr>
</thead>
<tbody>
<tr>
<td>T₁</td>
<td>0</td>
<td>null</td>
</tr>
<tr>
<td>T₁₁</td>
<td>1</td>
<td>T₁ /</td>
</tr>
<tr>
<td>T₁₂</td>
<td>1</td>
<td>T₁ /</td>
</tr>
<tr>
<td>T₁₂₁</td>
<td>2</td>
<td>T₁ / T₁₂ /</td>
</tr>
<tr>
<td>T₁₂₂</td>
<td>2</td>
<td>T₁ / T₁₂ /</td>
</tr>
</tbody>
</table>

Table 10.3: XMLtoTable_P₂
Second, while generating a new production by applying the latter production \( P_2 \) to the non-terminal node of the former production \( P_1 \), we use the algorithm to update the information in the relational table.

\[
\begin{align*}
P_1: & O \Rightarrow X \\
& X \\

P_2: & O \Rightarrow X \\
& X \\
& X \\
& X
\end{align*}
\]

Figure 10.3: Production three

Its XML format appears as follows:

```xml
<T1><T11/> <T12><T11/> <T12><T121/> <T122/> </T12></T11></T12></T1
```

We can now provide an algorithm to update the records and merge the two relational tables into the whole one, while applying \( P_2 \) to \( P_1 \).

/* Start of algorithm*/

Get level (L) and path (P) of the non-terminal node from table XMLtoTable_P1;
Replace the level and path of the root node from table XMLtoTable_P2 with L and P;
Delete the record of the non-terminal node from table XMLtoTable_P1;
Update the level (level = level + L) of all the nodes in table XMLtoTable_P2, except for the root node;
Update the path (adding \( P \) ahead of the string of the path, That is, path = “\( P \)” + path) of all the nodes in table XMLtoTable_P2, except for the root node;
Merge table XMLtoTable_P1 and table XMLtoTable_P2 into one relational table;

/* End of algorithm*/
Through the above algorithm, we can obtain the following updated relational table:

<table>
<thead>
<tr>
<th>Tagname</th>
<th>Level</th>
<th>Path</th>
</tr>
</thead>
<tbody>
<tr>
<td>T1</td>
<td>0</td>
<td>null</td>
</tr>
<tr>
<td>T11</td>
<td>1</td>
<td>T1 /</td>
</tr>
<tr>
<td>T1</td>
<td>1</td>
<td>T1 /</td>
</tr>
<tr>
<td>T11</td>
<td>2</td>
<td>T1 / T1</td>
</tr>
<tr>
<td>T12</td>
<td>2</td>
<td>T1 / T1</td>
</tr>
<tr>
<td>T121</td>
<td>3</td>
<td>T1 / T1 / T12</td>
</tr>
<tr>
<td>T122</td>
<td>3</td>
<td>T1 / T1 / T12</td>
</tr>
</tbody>
</table>

**Table 10.4: The updated XMLtoTable_P1**

Third, through generating final production \(((P_1 * m_1, P_2 * m_2, ..., P_n * m_m) * k)\), we can then obtain the whole relational table, which includes information on all the element nodes in the XML document.

**10.2. Generation of paths**

Before paths can be generated, we depict how to search for specified records from the relational table through the index. During the course of generating paths, it is necessary to find records with a specified tag name or level.

We have already created two indices on column Tagname and Level in the relational table. The relational table is then sorted by ascent of tag name or level value. For example, we describe them in the following tables:
We search for the specified records through the index of Level,

<table>
<thead>
<tr>
<th>Index of Level</th>
<th>Pointer</th>
<th>Level</th>
<th>Tagname</th>
<th>Path</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td>0</td>
<td>T₁</td>
<td>null</td>
</tr>
</tbody>
</table>
| 1              |         | 1     | T₁₁     | T₁ /
| 2              |         | 1     | T₁₁₁    | T₁ /
| :              |         | 2     | T₁₁₁    | T₁ / T₁₁|
| depth          |         | 2     | T₁₁₂    | T₁ / T₁₁|

Table 10.5: Index of Level

We search for the specified records through the index of Tagname,

<table>
<thead>
<tr>
<th>Index of Tagname</th>
<th>Pointer</th>
<th>Tagname</th>
<th>Level</th>
<th>Path</th>
</tr>
</thead>
<tbody>
<tr>
<td>T₁</td>
<td></td>
<td>T₁</td>
<td>0</td>
<td>null</td>
</tr>
</tbody>
</table>
| T₁₁              |         | T₁₁     | 1     | T₁ /
| T₁₁₁             |         | T₁₁₁    | 1     | T₁ /
| T₁₁₂             |         | T₁₁₁    | 2     | T₁ / T₁₁|
| :                |         | :       | :     | :    |

Table 10.6: Index of Tagname

In terms of the user input, there are four different path models:

Model 1: (start node, end node)
Model 2: (start node, second level)
Model 3: (first level, end node)
Model 4: (first level, second level)

Next, we provide the algorithms for these different path models:

Model 1: the user inputs “start node” and “end node”
Model 2: the user inputs “start node” and “second level”
/*Start of algorithm*/

Through the index of Tagname, find m level values (stored in array level [m]) whose tag name is “start node”; 
If (model 1)

Through the index of Tagname, find n records whose tag name is “end node”; 
If (model 2)

Through the index of Level, find n records whose level is equal to “second level”; 
For (i=1; i<=m; i++)

For (j=1; j<=n; j++)

{ Find the tag name of the ancestor whose depth is equal to level [i], in the string of the path in record j; 
If (tag name of this ancestor is equal to “start node”) 

{ Get the sub string, which is from this ancestor to the end of the string; 
Save the sub string and end node as a new path; 
Store this new path in the string array paths collector; 
}
}

num = number of paths in the paths collector; 
One preferred path = randomly generate one path from 1 to num; 
Print out this preferred path; 
/*End of algorithm*/

Model 3: the user inputs “first level” and “end node”
Model 4: the user inputs “first level” and “second level”

/*Start of algorithm */

If (model 3)

Through the index of Tagname, find n records whose tag name is “end node”; 
If (model 4)

Through the index of Level, find n records whose level is equal to “second level”; 
For (j=1; j<=n; j++)

{ Find the tag name of the ancestor whose depth is equal to “first level”, in the string of the path in record j; 
Get the sub string, which is from this ancestor to the end of the string; 
Save the sub string and end node as a new path; 

84
Store this new path in the string array paths collector;
}
num = number of paths in the paths collector;
One preferred path = randomly generate one path from 1 to num;
Print out this preferred path;
/*End of algorithm */

10.3. Insertion of specified attributes

Based on the paths that have been generated and the patterns of benchmark queries, XLST is utilised to insert the following elements and their values into specified positions in XML documents. These elements, such as attributes, IDs, IDRefs, and order name, will be applied to generate benchmark queries.

For example, we need to insert one attribute name, whose attribute value includes sub string “yu”, into a specified position in an XML document.

```
for $A_1$ in document (“file1.xml”) / / NSW / Wollongong / Student
where contains ($A_1$ / / @name, "yu")
return $A_1$ / text()
```

We utilise XSLT to implement these operations. In XSLT, <xsl:attribute> can be used to dynamically add an attribute to an element in an XML document: <xsl:attribute name="attribute name">. Moreover, an <xsl:attribute-set> can be used for a related group of attributes: <xsl:attribute-set name="name of attributes set" use-attribute-sets="attributes set names">.

10.4. Output of benchmark queries

The program utilises the paths that have been generated and those relative attributes, order name, IDs, and IDRefs to replace specified positions in the patterns of benchmark queries. In terms of user input, these parameters have also been applied to relative positions in the patterns of benchmark queries (refer to section 8.3.).
Chapter 11

Related XML Benchmark Systems

In this chapter, we compare the MyBench system with other XML benchmarking systems. We focus on document structure control, document size control, and benchmark query generation.

11.1. Comparison of XML document structure control

In an XML document generator system, an important problem is how to efficiently control the structure of XML documents. Different benchmark systems utilise various methods to execute their document control strategies.

ToXgene [26] provides a template to describe the intended data. XML documents are generated in terms of this template, so the user can efficiently control the document structure. Furthermore, ToXgene can do some random control on the input templates, and it can generate XML documents with some irregular structures. In addition, ToXgene can generate some complex XML contents, such as elements with mixed content or attributes values.

In XMark [27], an Internet auction site is used to describe the structure of the benchmark document. The XMark benchmark also adds the references to the data. The text data that are used in XML documents are the 17,000 most frequently occurring words in Shakespeare's plays. XMark also provides a DTD and XML Schema for more efficient mappings.

XMach-1 [29] is a multi-user benchmark for XML database systems that comprise both structured data and text documents. The text document is collected from the web applications. Each of the XML files contains some elements such as title,
chapter, section, and paragraph. It also supports both a schema-based and schema-less variant of the benchmark.

In the MyBench system, the user can not only choose any kind of structure of XML document, but can also decide the contents of an XML document, such as elements, element contents, attributes and so on. XMark and XMach-1, on the other hand, only provide a relatively fixed document structure, and the user cannot make many changes to it. ToXgene provides the complex templates for an XML document; however, it is not easy for users to operate. MyBench benchmark utilises graph grammars to generate XML documents by choosing a group of different types of simple level productions or multiple level productions. The user can easily control the structure of an XML document by choosing suitable productions and deciding on the replacement order of all the productions. Furthermore, using an XSLT stylesheet to implement an XML document generator is an easy and understandable method.

11.2. Comparison of XML document size control

In an XML document generator, another problem that arises is how to scale the benchmark data set. Different benchmark systems also utilise different methods to control XML document size.

ToXgene [26] can store temporary data structures in a persistent object manager. The user can optimise the system performance by customizing the buffer management and taking advantage of parallel I/O. However, the current version of ToXgene takes a long time to generate large size databases (1GB).

XMark [27] controls the database size by a certain scale factor. Xmlgen can provide accurate scaling by changing some factors such as the number of items and persons.

XMach-1 [29] assumes that the XML files are small. It therefore varies the database size by increasing or decreasing the total files number. Four different kinds of database sizes (10000, 100000, 1000000 or 10000000) can be exploited for different system requirements.
XOO7 [30] allows users to change the XML document size in both depth and width.

MyBench XML document generator can generate the appropriate XML document in the following two ways. One is to change the total number of original XML documents, whilst the other is to increase or decrease document size by controlling the depth and width of every XML document.

11.3. Comparison of benchmark query generation

In benchmark queries, most XML database benchmarks focus only on the functionalities that benchmark queries can cover. These benchmark systems design different types of queries to check different kinds of features of XML databases. However, the MyBench system not only covers most of the functionalities, but also provides a pattern for automatically generating benchmark queries. Furthermore, the relational DBMS can be utilised to generate benchmark queries quickly.

XBench [25] exploits a natural language to express their queries. The workload contains almost all of XQuery functionality. XMach-1 [29] queries consist of eight query operations and three update operations. XMach-1 has relatively few benchmark queries compared with XMark [27] and XOO7 [30], and covers the least number of functionalities. All the queries in XOO7 are simple, and each query covers just a few functionalities. The user can choose a subset of queries to test the features that are required in their applications. XMark provides 20 benchmark queries, and the majority of the queries in XMark are complex and cover most of the features. However, it is not clear which query contributes a special functionality, because one query in XMark often covers several features.

MyBench queries can also cover almost functionalities. One difference is that the MyBench system can generate benchmark queries automatically after inputting a set of parameters. The Benchmark system sometimes needs to generate MByte or even GByte XML documents, so it is hard to locate the specified nodes in such large files. Automatic generation of benchmark queries is imperative for large XML documents, as
it can speed up the process of benchmarking. In the MyBench system, we first generate the relational table that contains information about an XML document. Second, given the start node, end node, first level and second level, the program generates paths that are used in the benchmark queries, through the relational DBMS. Third, the system automatically generates benchmark queries, in terms of the special functionality and the given patterns.
Chapter 12
Contributions, Conclusions and Open Problems

12.1. Contributions

In this thesis, we provide a new benchmark system called MyBench for the benchmarking of native XML database systems. MyBench system consists of two parts: XML document generator system and benchmark query generator system.

Unlike other XML document generator systems, the MyBench system utilises graph grammars to generate XML documents. Users can easily control the structure and size of XML documents by choosing a group of small original documents and deciding on the replacement order of all the original documents. Furthermore, using an XSLT stylesheet to implement the XML document generator is an easy and understandable method.

In terms of benchmark queries, other XML database benchmark systems only focus on designing queries with different functionalities. The MyBench system not only covers most of the queries’ functionalitites, but also provides a group of benchmark query patterns. As far as benchmark query patterns and a set of parameters that the user inputs are concerned, the MyBench system can automatically generate those benchmark queries. When the XML documents that are generated are large or even huge, automatic generation of benchmark queries will be necessary.

12.2. Conclusions

The aim of this work is to develop a methodology for benchmarking native XML database systems. The proposed methodology describes how to generate XML documents with different structure and size. The methodology itself includes four major parts.
The first part is to use graph grammar to generate XML documents with different structure and size. Based on graph grammar, this part defines a series of terms: single level production, multiple level production, and final production. The XML document that is generated is derived from final production. By changing the structure, size, and number of productions, it is easy to gain any kind of XML document that a user might need.

The second part is to generate benchmark queries automatically. By generating specified paths and setting parameters, the system can generate a group of benchmark queries automatically.

The third part is to use an XSLT stylesheet to implement the MyBench XML document generator. Based on the above description of using graph grammar to generate XML documents, the implementation is divided into three steps, and each step is implemented by a special XSLT stylesheet.

The fourth part is to use a relational table to implement benchmark queries. We create a relational table that contains information about the original XML documents. While generating an XML document, we update the information in this table. According to this table, we can easily generate the paths of benchmark queries. Then we can output benchmark queries in terms of the patterns of benchmark queries.

12.2. Open problems

Of course, there is still research to be undertaken in the future. First, using an XSLT stylesheet to generate an XML document is an easy and understandable method; however, the three-step processing is relatively complex and time-consuming. Furthermore, this implementation only provides an interface that can be utilised by an application program. It is necessary, therefore, for the MyBench system to provide an efficient and integrated solution. Finally, the MyBench system needs to be run on some products of native XML database systems.
Bibliography


[38] XQuery, Available at http://www.w3.org/TR/xquery/ [accessed 20/09/2004]
Appendix A

Patterns and instances of benchmark queries

We depict the parameters of the following patterns:

a) Nesting level means the number of variables to be used in this query.

b) Filenames means the names of one or more XML files.

c) Relationship means its relationship is either parent-child “/” or ancestor-descendant “//” or sub tree “/*/”.

d) First level, second level and composite path level are described in section 8.1.

e) Comparison operator means the operator (eq”, or “ne”, or “lt”, or “le”, or “gt”, or “ge”) to be used.

Exact Match

The input parameters of this query include nesting level, filenames, relationship, first level, and second level. The system inserts a set of IDs in some specified positions in XML documents.

Input:
Nesting Level = n
Filenames (m) = filename 1, filename 2, …, filename m
Relationship = “/” or “//” or “/*/”
First Level (n) = first level 1, first level 2, …, first level n
Second Level (n) = second level 1, second level 2, …, second level n

Output:
for
    $A_1$ in document (“filename 1”) Relationship Path 1 (first level 1, second level 1) [@id="ID 1"],
    $A_2$ in document (“filename 2”) Relationship Path 2 (first level 2, second level 2) [@id="ID 2"],
    … … …
    $A_n$ in document (“filename m”) Relationship Path n (first level n, second level n) [@id="ID n"]
return
$A_1 / text (),
$A_2 / text (),
... ... ...
$A_n / text ()

For example,
Input:
Nesting Level = 3
Filenames (1) = file1.xml
Relationship = /
First Level (3) = 1, 1, 1
Second Level (3) = 2, 2, 2
Output:
for $A_1 in document ("file1.xml") / P_1 X_1 / P_2 X_1 / P_3 X_1[@id="P_3 X_1"],
$A_2 in document ("file1.xml") / P_1 X_2 / P_2 X_2 / P_3 X_2[@id="P_3 X_2"],
$A_3 in document ("file1.xml") / P_1 X_3 / P_2 X_3 / P_3 X_3[@id="P_3 X_3"]
return
$A_1 / text (), $A_2 / text (), $A_3 / text ()

Function Application

The input parameters of this query include nesting level, filenames, relationship, first level, and second level. The system inserts a set of attributes in some specified positions in XML documents.

Input:
Nesting Level = n
Filenames (m) = filename 1, filename 2, ..., filename m
Relationship = “/*” or “//” or “/”
First Level (n) = first level 1, first level 2, ..., first level n
Second Level (n) = second level 1, second level 2, ..., second level n
Output:
for
$A_1 in document ("filename 1") Relationship Path 1 (first level 1, second level 1) / @attribute,
$A_2 in document ("filename 2") Relationship Path 2 (first level 2, second level 2) / @attribute,
... ... ...
$A_n in document ("filename m") Relationship Path n (first level n, second level n) / @attribute
return
<Output>
  <Max> max ($A_1, A_2, \ldots, A_n) </Max>
  <Min> min ($A_1, A_2, \ldots, A_n) </Min>
  <Ave> sum ($A_1, A_2, \ldots, A_n) / count ($A_1, A_2, \ldots, A_n) </Ave>
</Output>

For example,
Input:
  Nesting Level = 3
  Filenames (3) = file1.xml, file2.xml, file3.xml
  Relationship = /
  First Level (3) = 1, 1, 1
  Second Level (3) = 2, 2, 2
Output:
  for $A_1$ in document (“file1.xml”) / $P_1 X_1$ / $P_2 X_1$ / $P_3 X_1$ / @age,
    $A_2$ in document (“file2.xml”) / $P_1 X_2$ / $P_2 X_2$ / $P_3 X_2$ / @age,
    $A_3$ in document (“file3.xml”) / $P_1 X_3$ / $P_2 X_3$ / $P_3 X_3$ / @age

return
  <Output>
    <Max> max ($A_1, A_2, A_3) </Max>
    <Min> min ($A_1, A_2, A_3) </Min>
    <Ave> sum ($A_1, A_2, A_3) / count ($A_1, A_2, A_3) </Ave>
  </Output>
</Output>

**Order Access**

The input parameters of this query include nesting level, filenames, relationship, comparison operators (“eq”, “ne”, “lt”, “le”, “gt”, “ge”), first level, and second level. The system inserts a set of attributes in some specified positions in XML documents.

Input:
  Nesting Level = n
  Filenames (m) = filename 1, filename 2, \ldots, filename m
  Relationship = “/*/*” or “//” or “/”
  Composite Path Level = level n+1
  Comparison Operator = “eq”, or “ne”, or “lt”, or “le”, or “gt”, or “ge”
  First Level (n) = first level 1, first level 2, \ldots, first level n
  Second Level (n) = second level 1, second level 2, \ldots, second level n
Output:
for
  $A_1$ in document (“filename 1”) Relationship Path 1
  (first level 1, second level 1),
  $A_2$ in document (“filename 2”) Relationship Path 2
  (first level 2, second level 2),
  ... ... ...
  $A_n$ in document (“filename m”) Relationship Path n
  (first level n, second level n)
where ($A_n / Composite Path n+1 (Path n, level n+1) /
  @attribute)
Comparison operator ($A_1 / @attribute)
return
  $A_n / Composite Path n+1 (Path n, level n+1) / text ()

For example,
Input:
Nesting Level = 2
Filenames (1) = file1.xml
Relationship = /
Composite Path Level = 3
Comparison Operator = “gt”
First Level (2) = 1, 1
Second Level (2) = 2, 2
Output:
for $A_1$ in document (“file1.xml”) / P_1X_1 / P_2X_1 / P_3X_1,
  $A_2$ in document (“file1.xml”) / P_1X_2 / P_2X_2 / P_3X_2
where $A_2 / P_4X_4 / P_5X_4 / P_6X_4 / @age gt $A_1 / @age
return
  $A_2 / P_4X_4 / P_5X_4 / P_6X_4 / text ()

Regular Path Expressions

The input parameters of this query include nesting level, filenames, relationship, first level, and second level. The system inserts a set of IDs in some specified positions in XML documents.

Input:
Nesting Level = n
Filenames (m) = filename 1, filename 2, ..., filename m
Relationship = “/*/*” or “//” or “/”
First Level (n) = first level 1, first level 2, ..., first level n
Second Level (n) = second level 1, second level 2, ..., second level n
Output:
for
$A_1$ in document (“filename 1”) Relationship Path 1
(first level 1, second level 1) [@id="ID 1"],

$A_2$ in document (“filename 2”) Relationship Path 2
(first level 2, second level 2) [@id="ID 2"],

... ... ...

$A_n$ in document (“filename m”) Relationship Path n
(first level n, second level n) [@id="ID n"]

return

<Output>
    $A_1$ / * / text (),
    $A_1$ / / text (),
    ... ... ...
    $A_n$ / * / text (),
    $A_n$ / / text ()
</Output>

For example,

Input:
Nesting Level = 3
Filenames (1) = file1.xml
Relationship = /
First Level (3) = 1, 1, 1
Second Level (3) = 2, 2, 2

Output:
for $A_1$ in document (“file1.xml”) / P_1X_1 / P_2X_1 / P_3X_1
[@id="P_3X_1"],
    $A_2$ in document (“file1.xml”) / P_1X_2 / P_2X_2 / P_3X_2
[@id="P_3X_2"],
    $A_3$ in document (“file1.xml”) / P_1X_3 / P_2X_3 / P_3X_3
[@id="P_3X_3"]
return

<Output>
    $A_1$ / * / text (),
    $A_1$ / / text (),
    $A_2$ / * / text (),
    $A_2$ / / text (),
    $A_3$ / * / text (),
    $A_3$ / / text ()
</Output>

**Sorting**

The input parameters of this query include nesting level, filenames, relationship, comparison operators, first level, and second level. The system inserts a set of attributes, and order name in some specified positions in XML documents and chooses relative constant value.
Input:
Nesting Level = n
Filenames (m) = filename 1, filename 2, ..., filename m
Relationship = "/*" or "/" or "/" or "//" or "/*/"
Comparison Operator = "eq", or "ne", or "lt", or "le", or "gt", or "ge"
First Level (n) = first level 1, first level 2, ..., first level n
Second Level (n) = second level 1, second level 2, ..., second level n

Output:
for
$A_1$ in document ("filename 1") Relationship Path 1
(first level 1, second level 1),
$A_2$ in document ("filename 2") Relationship Path 2
(first level 2, second level 2),
... ...
$A_n$ in document ("filename m") Relationship Path n
(first level n, second level n)
order by $A_n / * / Order name
where
($A_1 / @attribute Comparison operators "constant value")
and ($A_2 / @attribute Comparison operators "constant value")
...............and ($A_n / @attribute Comparison operators "constant value")

return
<Output>
  $A_n / * / text (),
  $A_n / * / Order name
</Output>

For example,
Input:
Nesting Level = 3
Filenames (1) = file1.xml
Relationship = /
Comparison Operator = "gt"
First Level (2) = 1, 1, 1
Second Level (2) = 2, 2, 2

Output:
for
$A_1$ in document ("file1.xml") / P_1X_1 / P_2X_1 / P_3X_1,
$A_2$ in document ("file1.xml") / P_1X_2 / P_2X_2 / P_3X_2,
$A_3$ in document ("file1.xml") / P_1X_3 / P_2X_3 / P_3X_3
order by $A_3 / * / @name
where
($A_1 / @age gt "20")
and ($A_2 / @age gt "25")
and ($A_3 / @age gt "30")

return
<Output>
Null Values

The input parameters of this query include nesting level, filenames, relationship, comparison operators, first level, and second level. The system inserts a set of attributes in some specified positions in XML documents and chooses relative constant value.

Input:
Nesting Level = n
Filenames (m) = filename 1, filename 2, ..., filename m
Relationship = “/*/” or “//” or “/”
Composite Path Level = level n+1
Comparison Operator = “eq”, or “ne”, or “lt”, or “le”, or “gt”, or “ge”
First Level (n) = first level 1, first level 2, ..., first level n
Second Level (n) = second level 1, second level 2, ..., second level n

Output:
For
$A_1$ in document (“filename 1”) Relationship Path 1
(first level 1, second level 1),
$A_2$ in document (“filename 2”) Relationship Path 2
(first level 2, second level 2),
... ... ...
$A_n$ in document (“filename m”) Relationship Path n
(first level n, second level n)
where empty ($A_n$ / Composite Path n+1 (Path n, level n+1) / text ())
and ($A_1$ / @attribute Comparison operator "constant value")
and ($A_2$ / @attribute Comparison operator "constant value")
return
$A_n$ / @attribute

For example,
Input:
Nesting Level = 3
Filenames (1) = file1.xml
Relationship = /
Composite Path Level = 3
Comparison Operator = “gt”, “lt”
First Level (3) = 1, 1, 1
Second Level (3) = 2, 2, 2

Output:
for $A_1$ in document (“file1.xml”) / $P_1X_1$ / $P_2X_1$ / $P_3X_1$,
$A_2$ in document ("file1.xml") / $P_1X_2$ / $P_2X_2$ / $P_3X_2$,
$A_3$ in document ("file1.xml") / $P_1X_3$ / $P_2X_3$ / $P_3X_3$
where empty ($A_3$ / $P_4X_1$ / $P_5X_1$ / $P_6X_1$ / text ())
and ($A_3$ / @age gt "25")
and ($A_2$ / @year lt "2001")
return
$A_3$ / @name

Text Search

The input parameters of this query include nesting level, filenames, relationship, first level, and second level. The system inserts the attribute in some specified positions in XML documents and chooses relative constant value.

Input:
Nesting Level = $n$
Filenames (m) = filename 1, filename 2, ..., filename $m$
Relationship = "//*/" or "///" or "//
First Level (n) = first level 1, first level 2, ..., first level $n$
Second Level (n) = second level 1, second level 2, ..., second level $n$

Output:
for $A_1$ in document ("filename 1") Relationship Path 1
(first level 1, second level 1),
$A_2$ in document ("filename 2") Relationship Path 2
(first level 2, second level 2),
... ... ...
$A_n$ in document ("filename m") Relationship Path n
(first level n, second level n)
where contains ($A_n$ / / @attribute, "constant value")
return
$A_n$ / text ()

For example,
Input:
Nesting Level = 1
Filenames (1) = file1.xml
Relationship = /
First Level (1) = 1
Second Level (1) = 2

Output:
for $A_1$ in document ("file1.xml") / $P_1X_1$ / $P_2X_1$ / $P_3X_1$
where contains ($A_1$ / / @name, "yu")
return
$A_1$ / text ()
**Data Type Cast**

The input parameters of this query include nesting level, filenames, relationship, comparison operator, first level, and second level. The system inserts a set of attributes in some specified positions in XML documents and chooses relative constant value.

**Input:**
- Nesting Level = $n$
- Filenames (m) = filename 1, filename 2, ..., filename m
- Relationship = "/*/" or "//" or "/"
- Composite Path Level = level $n+1$
- Comparison Operator = "eq", or "ne", or "lt", or "le", or "gt", or "ge"
- First Level (n) = first level 1, first level 2, ..., first level n
- Second Level (n) = second level 1, second level 2, ..., second level n

**Output:**

for $A_1$ in document ("filename 1") Relationship Path 1 (first level 1, second level 1),
$A_2$ in document ("filename 2") Relationship Path 2 (first level 2, second level 2),
...
$A_n$ in document ("filename m") Relationship Path n (first level n, second level n)

let $B := A_n / Composite Path n+1 (Path n, level n+1)$
where ($B / @attribute + A_1 / @attribute$) Comparison operator "constant value"
return $A_1 / text (), B / text ()$

For example,

**Input:**
- Nesting Level = 2
- Filenames (1) = file1.xml
- Relationship = /
- Composite Path Level = 3
- Comparison Operator = "gt"
- First Level (2) = 1, 1
- Second Level (2) = 2, 2

**Output:**

for $A_1$ in document ("file1.xml") / P_1X_1 / P_2X_1 / P_3X_1,
$A_2$ in document ("file1.xml") / P_1X_2 / P_2X_2 / P_3X_2

let $B := A_2 / P_4X_4 / P_5X_4 / P_6X_4,$
where ($B/@age + A_1/@age$) gt 50

return
Join On Values

The input parameters of this query include nesting level, filenames, relationship, comparison operators, first level, and second level. The system inserts a set of attributes in some specified positions in XML documents.

Input:
Nesting Level = n
Filenames (m) = filename 1, filename 2, ..., filename m
Relationship = “/*/” or “//” or “/”
Comparison Operator = “eq”, or “ne”, or “lt”, or “le”,
or “gt”, or “ge”
First Level (n) = first level 1, first level 2, ..., first level n
Second Level (n) = second level 1, second level 2, ..., second level n

Output:
for
$A_1$ in document (“filename 1”) Relationship Path 1
(first level 1, second level 1),
$A_2$ in document (“filename 2”) Relationship Path 2
(first level 2, second level 2),
...

$A_n$ in document (“filename m”) Relationship Path n
(first level n, second level n)
where
($A_1$ / @attribute Comparison operator $A_2$ / @ attribute)
and ($A_2$ / @ attribute Comparison operator $A_3$ / @ attribute)
...

and ($A_{n-1}$ / @ attribute Comparison operator $A_n$ / @ attribute)
return
$A_1$ / text (),
$A_2$ / text (),
...

$A_n$ / text ()

For example,
Input:
Nesting Level = 3
Filenames (3) = file1.xml, file2.xml, file3.xml
Relationship = "/
Comparison Operator = “gt”, “lt”
First Level (3) = 1, 1, 1
Second Level (3) = 2, 2, 2

Output:
for $A_1$ in document ("file1.xml") / P_1 \( \times_1 \) / P_2 \( \times_1 \) / P_3 \( \times_1 \),
$A_2$ in document ("file2.xml") / P_1 \( \times_2 \) / P_2 \( \times_2 \) / P_3 \( \times_2 \),
$A_3$ in document ("file3.xml") / P_1 \( \times_3 \) / P_2 \( \times_3 \) / P_3 \( \times_3 \),
where ($A_1 / @age gt A_2 / @age$) and ($A_1 / @age lt A_3 / @age$)
return
$A_1 / text ()$,
$A_2 / text ()$,
$A_3 / text ()$

References

The input parameters of this query include nesting level, filenames, relationship, first level, and second level. The system inserts a set of IDs, IDRefs and attributes in some specified positions in XML documents.

Input:
Nesting Level = n
Filenames (m) = filename 1, filename 2, ..., filename m
Relationship = "/*" or "//" or "/
First Level (n) = first level 1, first level 2, ..., first level n
Second Level (n) = second level 1, second level 2, ..., second level n

Output:
for
$A_1$ in document ("filename 1") Relationship Path 1 (first level 1, second level 1) [@id="ID 1"],
$A_2$ in document ("filename 2") Relationship Path 2 (first level 2, second level 2) [@id="ID 2"],
... ... ...
$A_n$ in document ("filename m") Relationship Path n (first level n, second level n) [@id="ID n"]
where $A_1 / @attribute = A_n / @attribute$
return
$A_1 / text ()$,
$A_2 / text ()$,
... ... ...
$A_n / text ()$

For example,
Input:
Nesting Level =3
Filenames (1) = file1.xml
Relationship = /
First Level (3) = 1, 1, 1  
Second Level (3) = 2, 2, 2

Output:
(ID:P_3X_1 refer to ID:P_3X_2 and ID:P_3X_2 refer to ID:P_3X_3)
for $A_1$ in document ("file1.xml") / P_1 / P_2 / P_3
[@id="P_3X_1"],
    $A_2$ in document ("file1.xml") / P_1 / P_2 / P_3
[@id="P_3X_2"],
    $A_3$ in document ("file1.xml") / P_1 / P_2 / P_3
[@id="P_3X_3"]
where $A_1$ / @name = $A_3$ / @name
return

$A_1$ / text (),
$A_2$ / text (),
$A_3$ / text ()