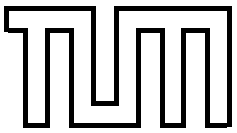

TECHNISCHE UNIVERSITÄT MÜNCHEN
FACULTY OF COMPUTER SCIENCE
Chair For Database Systems and Knowledge Bases
Prof. Dr. Rudolf Bayer

Diploma Thesis

Efficient Storage of XML Data Streams

submitted by
Richard Atterer
⟨atterer@informatik.tu-muenchen.de⟩

Supervisor: Prof. Dr. Donald Kossmann

December 15, 2002

Ich versichere, dass ich diese Diplomarbeit selbständig verfasst und nur die angegebenen Quellen und Hilfsmittel verwendet habe.

München, den 15. Dezember 2002

. . . . . .

Contents

1 Introduction **7**

2 Overview of Important Technologies **9**
2.1 XML . . . 9
2.1.1 Document Format . . . 9
2.1.2 Documents as Token Streams . . . 11
2.1.3 Document Structure . . . 13
2.1.4 Standards Related to XML . . . 15
2.2 Web Services . . . 20
2.3 XL . . . 24
2.3.1 Goals . . . 25
2.3.2 Language Features . . . 26
2.3.3 Current XL Implementation . . . 30

3 Storing XML Data **32**
3.1 Flat file . . . 33
3.2 Single Tokens . . . 34
3.2.1 Tokens as Records . . . 35
3.2.2 Tokens as Objects in an OODBMS . . . 36
3.2.3 Tokens as BLOBs in an RDBMS . . . 36
3.3 Token Stream . . . 38
3.3.1 Mapping XML Documents to Token Streams . . . 39
3.3.2 Variant Mapping Schemes . . . 40
3.3.3 Data Structures . . . 41
3.3.4 Granularity of Token Storage . . . 46
3.4 Tree Representation . . . 53
3.4.1 Tree Variants . . . 54
3.4.2 Storing the Tree On Disc . . . 60
3.5 Mapping the XML Data to an RDBMS Schema . . . 62
3.5.1 Edge Approach . . . 63
3.5.2 Attribute Approach . . . 64
3.5.3 Universal Table Approaches . . . 64
3.6 Improving Performance With Index Structures . . . 65
3.6.1 DataGuides . . . 65
3.6.2 Index Fabric . . . 66

3.6.3 Pre-/Postfix Order Node Numbering . . . 67
3.7 Conclusion . . . 69

4 Requirements **71**
4.1 Efficient Modification of XML Documents . . . 71
4.2 Persistent XML Storage . . . 72
4.3 Modular XML Storage System . . . 73

5 Design **74**
5.1 Components . . . 74
5.1.1 Persistent Storage . . . 74
5.1.2 Buffering . . . 76
5.1.3 Management of Storage Modules . . . 77
5.2 Programming Interface . . . 78
5.2.1 Storage Back-end Selection . . . 79
5.2.2 Document Iterators . . . 81
5.2.3 Token Identifiers . . . 82
5.2.4 Document Creation, Reads and Modification . . . 82
5.2.5 Making Values Persistent . . . 86
5.2.6 Buffering . . . 87
5.2.7 Test Framework . . . 87

6 Implementation **88**
6.1 XL_Value . . . 89
6.1.1 XL_Value Class Internals . . . 89
6.1.2 Module For Non-Persistent Storage: BufferedValue . . . 89
6.1.3 Required Changes in the Rest of XL . . . 94
6.2 Support For Persistent Variables . . . 95
6.2.1 Managing Storage Back-end Modules . . . 96
6.2.2 Changed Semantics of Variable Initialization . . . 96
6.2.3 Choice of Database System . . . 98
6.2.4 Module for Persistent Storage: LibdbValue . . . 100
6.3 Buffering Recently Accessed Tokens . . . 107
6.4 Further Issues . . . 109

7 Testing **111**
7.1 General Approach . . . 111
7.2 Low-level Integrity Checks . . . 111

7.3 Component Tests 112
7.4 XL Test Programs 112
7.5 Complex Test Application: auction.xl 113

8 Performance Tests **114**
8.1 Small Documents, Modifications 114
8.2 Large Documents, Reads 117

9 Conclusion **119**

1 Introduction

Like no other technology before it, the Extensible Markup Language (XML) has in the last years been embraced as the new standard format for information transmission and data storage, to the point of becoming the new "*lingua franca* of the Internet": An increasing number of applications uses XML as the format for saving documents, many web sites create and manage their content in XML format before converting it to HTML for display in a web browser, and new protocols use XML messages for communication.

One particular area of interest concerns itself with the use of XML for direct, automated communication between different entities. In contrast to the HTML interface intended for humans, so-called "web services" provide XML interfaces to the functionality that web sites offer (for instance, placing online orders or using a search facility). The ability to access sites via a common XML-based interface leads to new possibilities, ranging from more efficient business-to-business interaction to special "agent" services, for example to find the best price for a product among all the dealers it is available from, or to watch an online auction and act according to predefined rules.

Unfortunately, existing computer languages like Java or C++ do not lend themselves well to the task of processing XML data, e.g. as part of a web service implementation, because XML support is only available through add-on libraries and extensions, whose use often results in code that is difficult to read and unnecessarily complicated. This is even more unfortunate if we consider that a specification for a complete XML data query language named XQuery is available – however, the XML data model is so different from e.g. the Java type system that the seamless integration of XQuery into Java would prove a very difficult task.

As a result of this, the need for a separate language for XML processing was identified. The experimental language "XL" integrates XQuery into a general-purpose programming language and is being developed at the chair for database systems and knowledge bases at TUM. Its goals are to make the development of web services easy and less error-prone than with other languages – instead of spending his time with the issues of low-level XML data access, the programmer can concentrate on the application logic of the service.

In this diploma thesis, one of the tasks of XL is analysed and the corresponding functionality implemented: The persistent storage of XML data (in the form of XL variables) in a database. An XL variable can contain anything from a simple value (e.g. an integer) to a multi-gigabyte XML data structure, and in both cases a certain set of operations must be implemented reasonably efficiently for the language to be useful in practice. Additionally, is is also possible for *temporary* XL values to become too large to be stored in main memory. The developed program code also

addresses this problem, allowing temporary values (or parts of them) to be “swapped out” to disc under memory pressure.

The thesis can be broken up as follows: After a general summary in section 2 of the technologies mentioned above, the different ways of storing XML data in a database are discussed in section 3. As we will see, not all of them are suited for the purposes of XL – for example, XL requires that insertions/deletions in the XML document be efficient, so the straightforward approach of saving the XML data in a text file is not feasible. Section 3.6 shows that existing database systems are not ideal for XML storage in some respects, and gives an overview of different indexing methods and other low-level support structures.

Next, the requirements for the new XL storage subsystem are outlined (section 4), followed by a more concrete description of the component’s design (section 5). Finally, section 6 discusses the details of implementing persistent data storage for XL, and sections 7 and 8 describe the testing and performance measurement efforts.

2 Overview of Important Technologies

2.1 XML

The Extensible Markup Language [XML] is used widely by applications for storage and communication on the Internet. The World Wide Web Consortium (W3C) created the standard and maintains a web site (`http://www.w3.org`) from which it can be downloaded free of charge.

XML is both a very simple and a very powerful format to describe data: It is simple because it is text-based (so it can be edited with any ASCII editor) and the basic building blocks are easy to understand and memorize. Nevertheless, it is powerful because XML documents describe tree-like structures – together with the ability to insert references from one part of the described tree to any other part, this makes it general enough for the representation of arbitrary data structures.

The structure and format of XML documents is very similar to HTML documents – however, there is a major difference between the goals of these two markup languages: HTML is designed for the purpose of rendering documents in a browser for consumption by humans, and defines detailed semantics which describe how certain document contents influence the layout of the rendered HTML page. In contrast, XML does not attach any meaning to the content of the document, nor does it provide any means of rendering the content to screen; it merely provides a standardized way of representing information.

Both HTML and XML are subsets of SGML (Standard Generalized Markup Language, [SGML]), a markup language which is much older and slightly more complicated than XML. However, even though it is a superset of XML, in practice SGML has mostly been used for the description of human-readable documents, similar to HTML, and not for data exchange between programs. This difference with regard to the intended use and the goals of SGML and XML may be the reason why despite an initially mostly identical “feature set”, SGML has never been very popular, whereas XML generated a lot of interest from the beginning.

The following sections provide an introduction to the document format, the logical structure of documents and the special meaning that various character sequences have. Furthermore, a description of XML would not be complete if the various auxiliary standards which supplement it were not mentioned; even though many of them are not of interest for the purposes of this thesis, an overview of the most important ones is included.

2.1.1 Document Format

The XML specification published by the World Wide Web Consortium, [XML], is the authoritative standard which describes the format.

```
<?xml version="1.0"?>                                  XML declaration
<!DOCTYPE recipe SYSTEM "recipe.dtd">                  Document type declaration
<recipe>
    <!-- Do try this, it really tastes great! -->      Comment
    <title>Anikas Kürbisbrot</title>
    <ingredients>                                      Start tag
        <ingred name="flour" quantity="500–1000g"/>    Empty-element tags
        <ingred name="salt" quantity="pinch"/>         with CDATA attributes
        <ingred name="butter" quantity="25g"/>
        <ingred name="sugar" quantity="50g"/>
        <ingred name="yeast" quantity="40g"/>
        <ingred name="pumpkin" quantity="1 small"/>
    </ingredients>                                     End tag
    <time>3h</time>
    <instructions>
        Add mashed pumpkin to dough, knead thoroughly. PCDATA with <em> tag
        Add water/milk <em>or</em> flour for right     and a character entity for the
        consistency. Leave dough alone for some time,  degree symbol, i.e. "175°"
        knead once more, then bake at 175&#176; for 50
        to 60 minutes.
    </instructions>
</recipe>
```

Figure 1: *Example for an XML document (left), with short descriptions of the different parts (right).*

An XML document is a text document which contains whitespace, comments, "normal text" and special markup which creates the tree-like document structure. Figure 1 shows an example XML file to illustrate the different parts of the document.

At the lowest level, a document is composed of a sequence of simple units, each of which can be one of the following:

Character A single character like "x" or "∞". All of the characters described in the ISO/IEC 10646 standard ([ISO10646], a superset of the Unicode standard) can be used.

Character entity For convenience, characters can also be described by a special sequence of ASCII characters. Some characters can be referenced with a symbolic name (e.g. "<" for a less-than sign "<") and all of them can be specified using their ISO 10646 character code in decimal or hexadecimal notation (e.g. "<" or "<" for "<").

Whitespace A sequence of one or more space, tab, line feed or carriage return characters. It is worth noting that XML parsers do not distinguish between different amounts of whitespace, or between whitespace that contains line breaks and whitespace that doesn't. If information

about line breaks is to be stored, this can only be achieved by other means, such as with special tags like the <p> (paragraph) and
 (line break) in HTML.

Name A series of adjacent characters, starting with a letter and continuing with letters, digits and a few other characters. Whether a sequence which matches this description is parsed as a series of individual characters or as a name token depends on the parse context, as described later in this section.

Literal A string of text, enclosed in single '' or double "" quotation marks. The string can contain whitespace. Again, whether the characters are parsed as individual characters or as a literal is dependent on the context.

2.1.2 Documents as Token Streams

In the next step, an XML parser subdivides the document into a series of slightly larger parts which can be interpreted as a stream of tokens. A common characteristic of most of them is that the characters "<" and ">" are used to bracket areas of text to give them special meaning. The following different possibilities are defined in the XML standard:

Processing instruction A processing instruction can be recognized by the fact that the first character after the opening "<" and the last one before the closing ">" is a question mark. The only processing instruction defined by the XML standard itself is a line of the form

<?xml version="1.0"?>

at the start of a document, to identify the file as an XML document and to specify the version of the standard it complies to. All processing instructions starting with the letters "<?xml" are reserved for future use by XML, others can be used freely by applications.

The intended use for processing instructions is for the XML parser to pass their contents through to the application. This feature could be used in content management systems for web sites, where embedded commands could instruct the application to perform certain actions when the respective part of the document is processed. However, in practice systems like JavaServer Pages [JSP] tend to use the non-standard sequence "<%...%>" for this, not "<?...?>".

Comment It is often useful to make an XML processor temporarily ignore some part of the document without removing it completely, or to add short notes which are only intended for the person editing the document rather than the software that eventually processes it. For this purpose, XML comments can be used. They have the form

<!-- any text -->

The comment content can be arbitrary, with the exceptions that the characters "--" may not appear anywhere and that the character before the closing "-->" must not be a hyphen.

Start tag A simple start tag has the form "<recipe>". In addition to the tag name ("recipe"), any number of so-called attributes may be given, each of which has the form name="value", e.g.

<recipe qualification="rookie">

The names of tags and attributes are parsed as name tokens, the attribute values are parsed as literals. No attribute name may appear more than once in the same start tag or empty-element tag.

End tag For each start tag in a document, a matching end tag must also be present. It is distinguished from the start tag by a "/" character, e.g.

</recipe>

End tags must not contain attributes.

Empty-element tag XML documents frequently contain start tags which are immediately followed by the corresponding end tag, for example

<ingred name="sugar" quantity="50g"></ingred>

To improve readability, it is possible to combine these two into just one empty-element tag

<ingred name="sugar" quantity="50g"/>

The two forms can be used interchangeably; XML parsers are required not to distinguish between them. Note that the "/" character is moved to the end of the tag in empty-element tags, before the closing ">".

Looking at the XML example in figure 1, the tokens described above cover most of the document. The only remaining part is the "normal text" enclosed by the "<instructions>" tags. The standard distinguishes between two different kinds of such text:

PCDATA This acronym is used to describe an arbitrary sequence of text and XML tags; an example is the contents of "<instructions>", which contains an "" tag as well as normal text. Because of this property, such data is referred to as "mixed content" – the term "parsed character data" was was originally used for it in SGML, but it is misleading in the context of XML.

CDATA "Character data" is text which cannot contain tags. The most obvious example for this is the value of attributes. With a tag like

<ingred name="sugar" quantity="lots!"/>

the "" tags are not recognized. This example is not even well-formed XML: The value of the "quantity" attribute is not simply a string whose characters are "<", "e", "m" etc. because a special rule in the standard forbids the use of the less-than character in attribute values. To make it well-formed, all occurrences of "<" would have to be replaced with "<".

2.1.3 Document Structure

The XML standard distinguishes between *well-formed* and *valid* documents. A document is considered well-formed if its structure follows the rules set by the standard (e.g. tags are correctly nested), and a document is considered valid if the document structure additionally complies to the rules laid out in a *document type definition* which is referenced or included at the beginning of the document (e.g. restrictions for the content of elements).

Structure Constraints Imposed by the XML Standard In XML, an *element* is defined as a start tag followed by the corresponding end tag, and any content that the two tags enclose. Alternatively, an element may also be a single empty-element tag.

As mentioned earlier, XML documents can be regarded as tree-like structures. For this reason, it is required that all tags be nested correctly (including tags that are part of an element's content), and that the document only contain exactly one top-level element, or *document root*. Figure 2 shows the tree implied by the example XML document from figure 1.

Many of the terms used to denote relationships between elements are the same as those commonly used for tree graphs: The elements enclosed by another element are that element's *children*, and the enclosing element is the *parent*. Attributes and their values could also be considered children of the element node (and figure 2 shows them as such), but [XML] only defines these terms for elements. Furthermore, elements precede or follow other elements in *document order* if their start and end tags occur before or after the other element's tags in the XML document.

Structure Constraints Imposed by a User-Supplied Document Type Definition In addition to the restrictions mandated by the XML standard, it is possible to specify further criteria for the document content. This is done using a document type definition, or DTD, and is always specific to a certain application of XML – for instance, it allows you to specify the set of all recipes that

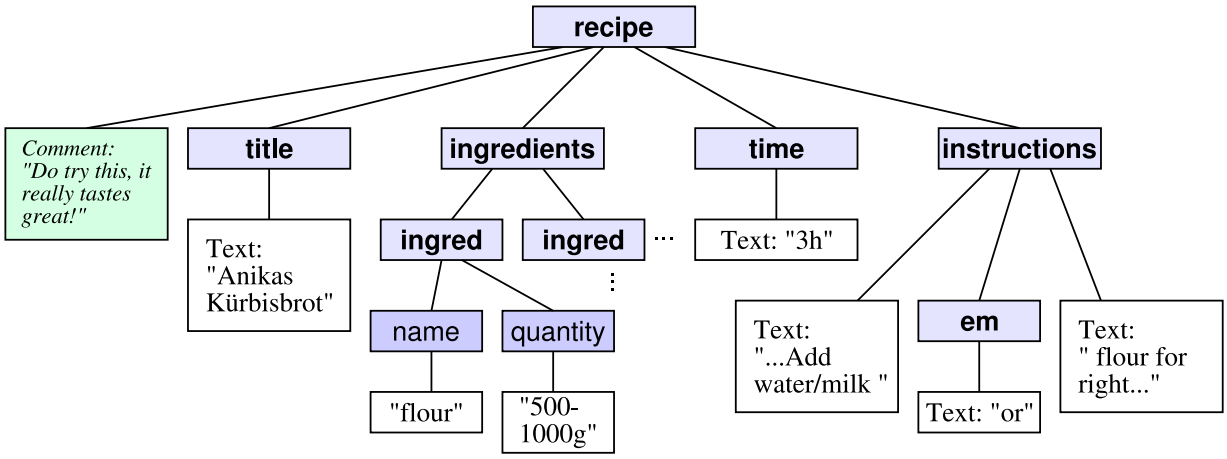

Figure 2: *Tree view for the example XML document*

look like the example from figure 1. DTDs are a feature which XML inherits from SGML; a very well-known DTD for SGML is the DTD which restricts the set of all possible SGML documents to the set of HTML documents.

The following aspects of the document content can be customized using a document type definition:

- Legal name(s) for the root element;
- whether an element is empty or not;
- if it is not empty, which element(s) or sequences of elements may appear as children of the element. The element type declarations essentially represent an LL(1) grammar, so quite detailed descriptions of the element content are possible;
- whether an element may contain PCDATA, and if so, whether child elements are allowed (i.e. whether mixed content is possible) and what names any children may have;
- the set of legal attributes for an element.

Furthermore, DTDs allow the definition of additional character references (like “<”, but with any other sequence of letters instead of the “lt”). To improve readability, there is also a way to give symbolic names to entity definitions and to refer to them elsewhere in the DTD with a so-called parameter-entity reference.

DTDs can either be referenced indirectly by a document type declaration at the start of an XML document (as in figure 1), or they can be defined directly in the document; documents with an in-lined document type definition are called *standalone*.

For the exact syntax of DTDs, please refer to the XML standard.

Document type definitions are a quite powerful way to provide rules for the content of XML or SGML documents. Still, in the case of XML, they did not prove sufficient – XML Schema (described below) introduces other, more advanced ways to specify constraints for the document structure.

2.1.4 Standards Related to XML

As we have seen in the previous sections, the structure of XML documents is quite simple and straightforward – an important feature which has doubtlessly promoted its widespread use. However, as a result of attempts to use XML for more complicated applications, a growing number of related standards has evolved, and while these standards' features make XML much more powerful, they also introduce additional complexity.

In this section, the most important of these XML-related standards are introduced briefly, and the motivation for their existence is explained. A detailed description of all these standards is outside the scope of this thesis – please refer to the respective W3C documents for further information.

XML Schema The XML Schema facility pursues the same goals as the SGML DTDs mentioned in the previous section: To specify rules for the structure of a document, the tags it may contain and the way they can be nested. However, its features go far beyond those of DTDs.

As the subtitle of the ISO SGML standard indicates, SGML is only intended for the area of "text processing" – consequently, the only primitive data type it supports are strings of characters. In contrast to this, XML documents hold data of many other types. While the actual representation of that data is still a stream of characters, the applications that process these characters need to know what type to associate with it, e.g. to distinguish the case where the string "0172–98443" is a mathematical expression from the case where it is a telephone number.

Some further features of XML schema which are not present in DTDs:

- You can specify what types of characters are allowed inside a certain tag, e.g. that telephone numbers consist of digits and "–" characters, or that the value enclosed by a "<size>" tag and its corresponding end tag must be one out of "small", "medium" or "large".
- A number of primitive data types are predefined, e.g. byte, float, date or int, some of which allow additional restrictions for the represented value, such as "an integer between 5 and 100".

- You can provide several alternatives for the contents of tags, for example "The <a> tag either contains <b/><c/> or <d/><e/> (but not e.g. <b/><d/>)"
- Types can be based on other types, so very complex data types can be described easily.
- XML schemas are themselves written as XML documents, unlike DTDs, which use a completely different notation.

XML Schema is far more complex than most other XML-related standards, so no attempt is made here to describe it completely. The non-normative "XML Schema Primer" [XMLSchema0] is an introduction to the XML Schema language which is easier to read than the standard itself [XMLSchema1], [XMLSchema2].

XML Namespaces As a result of XML's ability to represent all types of data in the same standardized structure, it becomes possible to include in one XML document data that was defined for use by several different schemas (for example schemas defined by different companies), or by the schemas that accompany different W3C standards.

XML namespaces [XMLNS] make it easy to integrate schemas without name clashes. Such clashes can occur if tags with identical names are defined for different purposes in different schemas, and those schemas used in a single document. XML namespaces avoid problems first by assigning a *namespace prefix* to each schema definition, and second by qualifying names with that prefix (simply by prepending the namespace prefixes and a colon to tag names). For example, the XML Schema definitions can be assigned the prefix "xsd" with

<sometag xmlns:xsd="http://www.w3.org/2001/XMLSchema">

Within the content of the "sometag" element, XML Schema tags can now be referred to using e.g. "<xsd:complexType>" instead of just "<complexType>".

XML Stylesheets Even though SGML was originally intended to become the primary standard for the representation of documents like articles, books or web pages, nothing prevents XML from being used for this purpose as well. On the contrary, XML's advanced features make it a more powerful solution for text processing than SGML.

The primary task in this application of XML is the conversion of XML documents into other formats. A typical output format (also called "presentation format") is HTML (for online viewing of documents), but using available tools it is also possible to produce output which is not related to SGML in any way, such as PostScript or PDF (for printed documentation).

The Extensible Stylesheet Language [XSL] is a language for expressing "stylesheets". A stylesheet specifies how an XML document is to be presented, by providing rules which govern

transformation of the document into the output format. Stylesheets are collections of templates, each of which consists of a pattern to be compared to the element of the document and the replacement text to use for any instances of the pattern. As usual with XML standards, the pattern matching and replacement mechanism is quite flexible and powerful. The exact form of the patterns and transformation rules is given in the standard for XSL Transformations [XSLT], which is a part of XSL.

Today, XSL(T) is often preferred over SGML for document formatting, and is also being used internally on more and more web sites to separate the content of HTML pages from their formatting, by storing them separately (content in XML files, site layout in a stylesheet) and generating normal HTML on the fly when pages are delivered. However, modern web browsers also have built-in support for XSLT.

XLink and XPointer As we have seen, the XML standard does not attach any meaning to the data represented by an XML document. Consequently, it also does not define any semantics for hyperlinks, which would allow documents to reference other documents or parts of documents – in contrast to e.g. HTML, where "<a>" (anchor) tags can reference points in the document which were marked with name="..." or id="..." attributes.

The XLink [XLink] and XPointer [XPointer] standards provide hyperlink functionality for XML in a very flexible way. The references can have the following characteristics:

- They can be simple links like in HTML. These links can only point to elements, and they can only point to elements which have had a name attached to them by the author of the document.
- They can link to data which has *not* been prepared to be a hyperlink destination.
- Links to single characters in any PCDATA parts of the document are possible.
- Not only points inside the document can be referenced, but also regions of the document.
- Using the XPath language (see next section), points or regions in a document can be described indirectly, for example "the second-to-last <ingred> element in the document."

XLink specifies how to add XML markup for hyperlinks to documents. The XPointer language, whose expressions can be included in the XLink markup, is a way of specifying points or regions in a document.

Again, the details of the standards are outside the scope of this thesis, and not all aspects of the following example (taken from the XPointer standard) can be explained. The example

illustrates the use of XLink with an embedded XPointer expression (which in turn uses the XPath language) in a "<button>" tag. A program interpreting the document could follow the hyperlink when the button is clicked.

```
<button
    xlink:type="simple"
    xlink:href="#xpointer(here()/ancestor::slide[1]/preceding::slide[1])">
    Previous
</button>
```

XPath and XQuery As soon as XML was beginning to get used to store large amounts of data, a task which had previously been left to relational databases, the need for an XML query language arose. Because of the inherent differences in the way data is stored in XML documents and in relational databases, it was not possible to adapt the standard query language for relational databases, SQL (*Structured Query Language*) to XML – eventually, the W3C developed the XPath and later the XQuery language [XQuery], using ideas from a number of competing XML query languages.

The major differences between XPath and XQuery are:

- XPath is a subset of XQuery, and significantly simpler.
- XPath operates on a set of just four types (node-set, boolean, number, string), whereas XQuery includes full support for XML Schema.
- Since XPath was created for integration with XLink/XPointer, its queries concentrate on selecting a contiguous part of the document, unlike XQuery which can extract arbitrary multiple sections of the document.
- XPath always queries only one document, whereas XQuery can process several documents.

The rest of this section concentrates on XQuery because it is part of the XL language, and so is of greater interest for our purposes.

XQuery is a functional language whose query expressions look similar to expressions in many general-purpose programming languages – above all, the queries are not themselves formulated in XML markup, but in a more concise notation. (However, an XML representation for queries has been defined by the W3C in a separate standard.) The language also supports many other features of normal programming languages, including arithmetic, comparison and logical expressions, function calls, variables, type checking and type conversions. In addition to these, special constructs oriented towards its use on XML data have been added, including so-called *path*

expressions (for navigation in the document tree) and the powerful FLWR expression (named after its keywords *for*, *let*, *where*, *return*) which can iterate over parts of the document and bind variables to intermediate results.

Every construct in XQuery is regarded as an expression. These expressions return sequences, i.e. ordered collections of zero or more values, and each of the values can either be a simple ("atomic") value such as an integer, or it can be another sequence, a union type, a string type, a name type or a document node. Nodes are the most interesting type, because they allow storing and addressing any part of the XML document tree in a very convenient way.

Here is an example for an XQuery expression. It assumes that the example document from figure 1 (page 10) is available on the system under the name "recipe.xml":

```
<info>
    {
        let $recipe := document("recipe.xml")
        return <emph>You only need
            {$recipe/ingredients/ingred[@name="pumpkin"]/@quantity}!</emph>
    }
</info>
```

After binding the document to the variable $recipe, the embedded path expression selects the "ingredients" node, then all "ingred" nodes contained in it, restricts that set of nodes to all nodes whose "name" attribute has the value "pumpkin" (only one node remains) and finally outputs the "quantity" attribute value of that node. The resulting output is:

```
<info>
    <emph>You only need 1 small!</emph>
</info>
```

The example above uses the so-called *abbreviated syntax* which is designed to be similar to the syntax of URIs. An equivalent unabbreviated syntax exists for all expressions – its longer expressions use more descriptive names for the single parts (called "location steps") of the path expression. In unabbreviated syntax, the part of the above expression that accesses $recipe would look as follows:

```
$recipe/child::ingredients/child::ingred[attribute::name="pumpkin"]/
    attribute::quantity
```

There are steps which access the parent of the current node, the root node of the document it is stored in, its children, descendants (i.e. all children including the subtrees accessible via the children) and its attributes and text content.

While an XQuery implementation processes a query, it moves through the document, building up sets of nodes for each location step. One aspect of this process to keep in mind for later sections of this thesis is that internally, the XQuery implementation can reduce all the complex navigation through the document to only a few simple movements in the document tree. One minimal set of operations is as follows:

- Move forward in document order. Note that this also implicitly includes moving from a node to its first child, since only a forward step is needed.
- Move forward to the right sibling of a node. Another way of interpreting this is that it skips the contents of the node, moving directly to the node that follows it.
- Move from a node to its parent. The special thing about this movement is that it is the only one that does not move in document order, but backwards, because the start tag of the parent appears earlier in the document than its child's start tag.

Strictly speaking, even the second of these three operations is not necessary (to skip an element's content, an XQuery implementation could just read all the content and ignore it), but its presence is advantageous for performance reasons.

2.2 Web Services

A web service can be defined as the interface to some program logic, accessible over the Internet and used for application to application communication [W3C-WS]. The program typically runs in an environment on a server machine in which a small HTTP (Hypertext Transfer Protocol) web server is also running. The HTTP server accepts connections and passes requests coming from these connections on to the program. Alternatively, the program may also decide to initiate connections itself to contact other web services. Messages exchanged between different web services are XML-based, and include information about the type of operation to be performed by the web service, as well as any data necessary to perform it. Usually, the SOAP protocol (*Simple Object Access Protocol*) is used for this purpose.

Features of Web Services Today, web services are considered by many to be a major topic in web-based technologies for the next years – the new web service related standards are embraced with the same startling speed as XML. The reasons for this trend are manifold, and become clearer when you take a closer look at the features of web services:

- The entire communication process is based on open standards of the World Wide Web Consortium.
- Due to the W3C's patent policy, the chances are high that none of the technologies are covered by patents. Thus, development costs for web services can be kept low.
- It is easy to adapt existing network infrastructure to support web services – for example, because the protocol used for communication is HTTP, no reconfiguration of the firewall is necessary.
- The communication can take advantage of other features of HTTP, like encrypted HTTPS connections or proxy servers.
- Because of the use of well-known protocols like HTTP, it is also easy to make web services accessible to many operating systems.
- Web services avoid incompatibilities between online services which use different component services, like DCOM and CORBA. (Obviously, this is achieved by introducing yet another standard...)
- They are embraced by all the major players.

Aim of Web Services The intended aim of web services is to ease the integration of publically available services. In the same way that HTML has made information and services available to *humans* (irrespective of the operating system or browser they use, or the fact that they might be visually impaired), web services are designed to make information flow between *programs* easy and uniform (regardless of the operating system they run on, the programming language they were written in, etc.). This overall goal can be broken up as follows (see also [Cerami02], [Vasudevan01]):

- The provided service can be described accurately: Any public interface provided by the service, including the data types it uses and signature of its methods, is described by a common XML grammar. This description is written using the Web Service Description Language (WSDL), another XML-based standard of the W3C.
- There is a way to publish the fact that a service is available, and to search for services. One way to achieve this is with an on-line directory of services – the UDDI, or Universal Description, Discovery, and Integration, is the directory currently used for this purpose, although it is possible to introduce other mechanisms for discovery of web services.

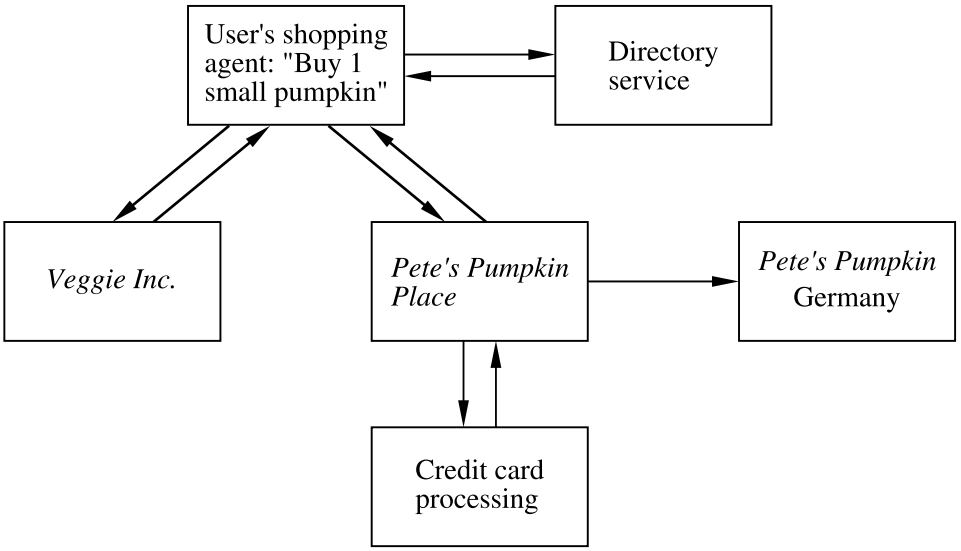

Figure 3: *Information flow between different web services: The user's shopping agent web service locates the best offer and places an order.*

- Once a service knows about another service, it can contact it, pass information to it and receive an answer using standard, well-defined protocols. In practice, these protocols are HTTP for the low-level transport and SOAP (or sometimes also XML-RPC, the XML Remote Procedure Call standard) to pass messages from one service to another.

- Optionally, the communication can be adapted to special needs of the involved parties. For example, if sensitive data is being exchanged, an additional encryption layer using SOAP-DSIG (SOAP digital signature) can be employed.

Scenario For the Use of Web Services In the previous explanations, the flexibility of web services may not yet have become apparent, since always only the communication between two services was discussed, one "client" service sending off a request and a "server" service which processes it. However, in fact the role of a service can change; it can sometimes contact a related service and sometimes be contacted by it, or it can always be a server to some of the services it interacts with, and always a client to others. Networks of web services resulting from such combinations of several services have the potential to deal with more advanced and complicated scenarios than the traditional client/server model.

Figure 3 shows an example for the interaction between several web services. Initially, an end user decides he wishes to find online shops which offer a certain product and to buy the article from the shop with the best offer. By running a "shopping agent" software on his computer, the

user starts a web service which can accomplish this task without further manual intervention:

- Compile a list of shops where the product might be available.
- Localize each shop in the list.
- Contact each shop and search for the desired product in its online catalogue.
- If the product is on offer, extract the price and shipping costs and other shipping details – for example, the shop might be in another country and might not allow international shipping.
- Pick the best offer.
- Contact the shop with the best offer and place an order, e.g. passing credit card information in the process.

In figure 3, the user's web service first uses a protocol like UDDI to inquire at a directory service about available services. UDDI can represent information about many different aspects of web services, including how to contact them. The answer of the directory server contains WSDL service descriptions for two services.

For simplicity, we assume that the programmatic interface to the two online shops is identical, so the user's "shopping agent" web service is able to contact all of them directly in the same way. In a more realistic scenario, the WSDL interface would be different – in that case (and unless knowledge about the different interfaces is built into the shopping agent) it would be necessary to make accesses to the two shops through yet another web service which specializes in "online shop services" and acts as a proxy to provide uniform access to the shops.

After searching for the product and extracting price information, the user's web service proceeds to buy from the second shop. In contrast to earlier queries, the order and subsequent exchange of credit card information is protected with authenticated and encrypted protocols, e.g. SOAP-DSIG or HTTPS.

Having received the order, the shop in turn relies on other web services to provide its own service: First, it contacts a credit card processing company to charge its customer's credit card, then it sends a message to the subsidiary nearest to the customer's address. The message specifies what product to ship and where to send it to.

Web services are still an evolving technology, with many new, often competing protocols being developed and presented all the time, so the overview in this section has concentrated on the basic principles rather than detailed descriptions of the various specifications and protocols. Hopefully, the ongoing struggle to reach a consensus about the different aspects of the technology

will result in a set of compatible, inter-operable standards rather than a fork into e.g. a “W3C web services” and a “Microsoft web services” world.

2.3 XL

XL is an experimental “XML programming language” that has been under development at the chair for database systems and knowledge bases at TUM since 2001. As described in [XL01] and [XL02], XL is a high-level, imperative, portable language. It is designed specifically for the implementation of web services and aims to eliminate problems that typically occur when other programming languages like Java are used for web service implementations:

- A mismatch exists between the type systems of the implementation language and of XML Schema. Rather than using features built into the language, the programmer needs to use additional libraries for access to XML documents, to XQuery or other XML-related technologies. The use of these libraries usually results in a larger amount of code which is more difficult to understand.
- Furthermore, if web services access a relational database (which is commonly the case for current web service implementations), the classical “impedance mismatch” between the relational data model and the implementation language is present.
- Additionally, the programmer is faced with the task of mapping the XML data received by his web service to the relational data model.

XL eliminates the need to work around these issues, making it possible for the programmer to concentrate on the application logic instead of implementation details. As a high-level language, it is suitable for implementing even complicated web services, while at the same time being simple enough and easy to use to make web service development proceed quickly. It uses the same data model as XML, so it is not necessary to convert information from one representation paradigm to another and back all the time. XL also contains direct support for a number of features frequently needed for the development of web services, such as logging, error handling, session management and retry of actions after timeouts. It is a superset of XQuery in the sense that all XL expressions are XQuery expressions – this reliance on W3C standards for important parts of the language provides a sound basis for the higher-level constructs.

The programming work carried out as part of the diploma thesis adds support for persistently stored values: XL variables containing any of the supported data types, including XML documents, can be written to disc and re-read at a later point in time, so their value is not lost when the XL program terminates.

2.3.1 Goals

[XL02] lists the following major goals which influenced the design of XL. As of this writing, not all features are fully supported, and XL as a whole should still be regarded as work-in-progress. Briefly, some important goals are:

- Compliance with W3C standards such as XML, XML Schema, XQuery and XSLT,
- Support for conversations between web services, i.e. "sessions" whose context is kept track of and which can last longer than the current connection,
- Support for reusing existing web services written in other languages, and composition of these services into a larger service,
- Support for standard features of web services, such as communication via messages and remote invocation over the Internet,
- Portability across environments (both across operating system and across database backends) and thus independence from the lower-level components of the system and the protocols/APIs (e.g. JDBC) used to access them,
- Reliance on the XML data model, and optionally strong typing,
- Special declarative constructs for frequently encountered problems; programs should be as high-level as possible,
- Support for exceptions and exception handling,
- Support for transactions, i.e. sequences of data accesses which are executed isolated and in an atomic way,
- Access control for all data possible, with authentication support,
- Support for automatic optimization, e.g. of database queries or of expressions/statements.

The general philosophy behind XL should become apparent from these goals: XL should provide all the features necessary for the development of web services, and these features should be easy to use – consequently, the increasingly complex worlds of W3C standards, databases and application logic should be connected in an elegant way with this high-level language.

2.3.2 Language Features

Many of the constructs in XL should be very familiar from other imperative programming languages, whereas others are specific to the programming language and geared towards its use with web services. Based on [XL02], this section introduces the main language features, concentrating on those that do not appear in other programming languages.

Overall Structure of XL Programs At the topmost level, an XL program is composed of one or more web services which communicate with each other by exchanging XML messages, using the SOAP protocol over HTTP.

Each web service is defined in a file – by convention, an `.xl` extension is used. Within the file, anything following the characters "!!" on a line is a comment and will be ignored by the XL parser. The basic structure of the file can be seen below, a more detailed description of the individual parts follows:

```
service http://localhost:3328/          !! Service declaration with service URI
    !! Namespace declarations
    namespace xsd = "http://www.w3.org/2001/XMLSchema";
    !! Definitions for global variables and conversation variables
    let $globalvar := 0;                !! Global variable with integer value
    context let $contextvar := <x/>;    !! Conversation variable with document type
    !! Declarative web service clauses
    init initOperation;                 !! Name of operation to run when WS starts
    conversationpattern mandatory;      !! Type of communication pattern
    !! Operation specifications
    operation initOperation
        !! Optionally, more declarations like conversationpattern here
        body
            !! Statements of the operation
            nothing;                    !! Surprisingly, this does nothing
        endbody
    endoperation
endservice
```

Namespace Declarations As mentioned before, the XL data model is identical to the XML data model. Consequently, it also needs to refer to Schema type definitions in W3C specifications, access tags defined elsewhere, etc. Just like the namespace declarations in XML documents, namespace declarations in this section of an `.xl` file make entities defined elsewhere accessible

in the program's context, and introduce a prefix for variable names:

```
namespace prefix = "http://url.to.associate.with.prefix/";
```

Declarative Web Service Clauses This part of the file describes general properties of the entire web service in a sequence of declarations. They specify the reaction to certain situations (such as the web service being invoked with a non-existent operation name, the violation of invariants or a modification to a variable that is being "watched"), the way to initiate conversations, and the behaviour when executing operations. A variety of different declarations are supported by XL – the complete list is available in [XL02].

An interesting aspect controlled by web service clauses is the concept of conversations. A conversation can be viewed as a "session context" which is valid over the course of more than one exchange of messages between two web service instances. This context is the same for all sent and received messages which contain an identical conversation URI in their envelope. The **conversationpattern** declaration allows the programmer to select between several different communication patterns, for example "this operation always requires a conversation to be present" or "if no conversation opened yet, create a new one". Using **conversationtimeout**, it is possible to have a conversation session expired (i.e. closed automatically) a certain amount of time after the last message exchange.

Global Variables and Conversation Variables The example above already showed that there are several ways to declare variables inside the web service declaration:

```
let integer $globalvar := 0;    !! Global variable with integer value, integer type
context let $contextvar;        !! Conversation variable with unspecified type, uninitialized
```

In general, a variable declaration begins with **let** or **context let**, optionally followed by a type for the variable (integer in the first line above), and then by the variable name. In XL, variable names always start with a dollar sign. It is possible to initialize the variable with a subsequent " := $anyExpression".

The semantics of global variables should be obvious: There is one instance of the variable, which is accessible from all operations in the web service. Typically, this will be some kind of "database variable" in which the web service stores larger amounts of data.

On the other hand, in the case of conversation variables one instance of the variable exists for each conversation. When the conversation opens, XL creates a new instance for each such variable and initializes it. Subsequent calls to other operations access this particular instance of the variable only if the operation call is made as part of the same conversation. As soon as the conversation terminates, its variables are also deleted.

Expressions All expressions in XL are XQuery expressions, as described in the XQuery W3C standard [XQuery] and also on page 18. This implies that they are based on the XML data model and that unlike with most other programming languages, powerful means to navigate through and search in XML documents are available as built-in XL language constructs.

Statements Being an imperative programming language, XL supports the usual set of standard control flow statements. In the examples below, $expression stands for any XL expression, $boolExpression for an expression which evaluates to a boolean value, and any other single statement can be used instead of the **nothing** statement – if several statements should be executed instead of just a single statement, they must be grouped in a block, bracketed by **begin** and **end**.

```
if ($boolExpression) then      !! Standard if-then-else construct
    nothing
endif
else                           !! The else branch is optional
    nothing
endelse;

switch                         !! Analogous to the Java/C switch statement
    if ($boolExpression1) then nothing end
    if ($boolExpression2) then nothing end
    ...
    default nothing end        !! Default branch is optional
endswitch;

while ($boolExpression) do
    nothing                    !! Executed as long as expression is true (≥ 0 times)
endwhile;

do
    nothing
until ($boolExpression);       !! Loop back to do as long as expression is false
```

Additionally, the language supports a **for**-**let**-**where**-**do** statement similar to the FLWR (**for**-**let**-**where**-**return**) statement in XQuery.

Local variables can be created with a simple **let** statement, which inserts the new variable into the innermost block. Modifications to variables are possible either by overwriting them, also with **let**, or by using one of the following five types of statement which update XML documents:

!! *Insert new node as last child of existing node, or as new left/right sibling*
insert $expression1 **into** $expression2;
insert $expression1 **before** $expression2;
insert $expression1 **after** $expression2;

!! *Delete node and all its children from document tree*
delete $expression;

!! *Move node from one place in the document to another*
move $expression1 **into** $expression2;
move $expression1 **before** $expression2;
move $expression1 **after** $expression2;

!! *Replace node with another node*
replace $expression1 **with** $expression2;

!! *Rename a node or an attribute*
rename $expression **as** "new-name";

The support for exceptions is similar to that in conventional programming languages: Using the **throw** statement, an exception is created. The exception can be any XL expression, it is frequently an XML document with a name for the error and an error message. To act on exceptions thrown during the execution of some statements, these statements are enclosed in **try**...**endtry** followed by a

catch $variable **do**...**endcatch**

clause which handles the error. Unlike with other programming languages, only a single **catch** clause is permitted.

Further statements supported by the language include logging statements (**logpoint**), waiting for events (**wait on event**) or changes in variable values (**wait on change**) and periodic retry of statements (**retry**). Statements can be executed in parallel or in non-deterministic order, by separating them with characters other than " ; ".

Operations Since XL is not object-oriented in the sense that Java or C++ is, the basic unit of code is called "operation". An additional property of operations is that they can have at most one input parameter and one return value, available through the $input and $output variables.

There is no simple function/operation call statement, only a construct to invoke other web services – but nothing prevents a web service from calling itself this way. Two variants of the web service invocation statement exist, a synchronous one which waits for the answer and stores it in a variable, and an asynchronous one which does not wait for an answer, instead it is possible to specify an operation to call when the answer is finally received:

```
!! Synchronous, optionally record answer in a variable
$expression --> http://service.com::operationName;
$expression --> http://service.com::operationName --> $variable;

!! Asynchronous, optionally call function with result
$expression ==> http://service.com::operationName;
$expression ==> http://service.com::operationName ==> operationName;
```

An operation can be paused for a while using the **sleep** statement, and terminated at any time with the **halt** and **return** statements – the latter passes the content of the $output variable back to the caller whereas the former terminates the operation without passing back a result. In the absence of either of these, XL returns the value of $output to the caller when the end of the operation body is reached.

2.3.3 Current XL Implementation

Because a significant portion of this thesis concerns itself not with just describing XML storage techniques, but with actually implementing them for XL, it is worthwhile also to take a look at the existing XL implementation.

The implementation has been under development for more than 1.5 years, and has reached a state where it is usable for small and medium-sized applications. The XL parser and interpreter is written in the Java programming language. As of this writing, it is comprised of about 150 Java source code files containing 23000 lines of code. This excludes the code for XQuery expressions, which was not developed at TUM, but is developed separately by XQRL, Inc.

Features A large number of the goals and features described in the previous sections is functional, including the following:

- Through XQRL's XQuery implementation, XL is compliant with the XQuery standard.
- Communication with XML messages is possible, web services can be called over the Internet by connecting to the built-in HTTP server.
- Full functionality for variables, whether global, operation-local or conversation variables.
- Most types of statements are implemented, including all the more frequently needed ones.

In addition, as part of this thesis, persistence for XL global variables and conversation variables and independence of any database back-end used for their storage has been implemented. Apart from global/conversation variables, this also affects large temporary variables, which can be swapped out to disc if they get too large to be stored in main memory.

Missing Features The following features are still missing, incomplete, or under development. However, it should be noted that most of these do not impair the usefulness of the language too much:

- Support for strong typing is missing.
- The following statements are unimplemented: **switch**, **do**...**until**, **for**...**let**...**where**, **logpoint**, **return**, **halt**, **sleep**, **wait on**..., **retry**.
- Transactions are not supported at the language level (e.g. grouping several **insert** statements and having them executed atomically).
- There are no built-in features related to security, such as encryption, authentication or access control.
- The implementation requires that XQuery expressions are enclosed in backslashes, e.g. "**let** $x := \5\;"
- Automatic optimization of XL programs is under development. During the time that this thesis was written, this area saw heavy development efforts, with two other diploma theses working towards allowing optimization by partitioning services across several machines, and by optimizing the XL programs themselves.

The reason why XL would probably not turn out to be suited for large programs at the moment is that its performance is too low in some areas. Additionally, the XL code is poorly commented and not organized very well, which makes XL development and improvements difficult.

3 Storing XML Data

How do you store XML data? The answer to this question seems obvious at first: As a simple text file – after all, the XML standard itself includes a description of how to represent XML documents as a stream of characters in ASCII or Unicode text files.

Thus, one would be tempted to choose a flat text file for storing XML documents and not spend further thoughts on the issue. However, it turns out that while this "standard" way of storing XML has advantages, it becomes infeasible when using large documents and regularly making changes in them.

Since the aim of this work is to find efficient ways of storing and updating XML data, with a special focus on implementing such a storage system for XL, this section discusses in depth the various different ways to store XML:

- You can store the data in a plain text file, as specified in the XML standard. (Section 3.1)
- Instead, you can also apply some of the knowledge about the file's formatting and store it as a stream of tokens in the sense of the definitions in section 2.1.2. (Section 3.3)
- Going one step further, you can also observe how tokens are nested in a well-formed XML document, and store the document as a tree-like structure. (Section 3.4)
- Alternatively, the XML document tree can be represented in relational databases with the use of a variety of mappings, without interpreting the data as a stream of tokens. (Section 3.5)

After looking at the pros and cons of each of these solutions to the XML storage problem in sections 3.1 to 3.5, section 3.6 provides an overview of ways to access XML data more quickly with the help of special index structures.

When we consider how to store XML data, an important aspect is that the data must also be easy to update. Section 2.3.2 describes the XL document modification statements (**insert**, **delete**, **move**, **replace**, **rename**) on page 29. A future version of XQuery which allows document updates will contain a very similar set of expressions.

Since the predominant way of reading (and in the future updating) XML is XQuery, the following sections will often discuss details of the storage approach with regard to how easily data that is stored this way can be processed by an XQuery engine.

3.1 Flat file

Using a normal text file to store an XML document can be regarded as the standard way to store XML. In fact, the format defined in [XML] and also detailed in section 2.1 is the only official way to represent XML. This is also the case why this format is used in XML-based communication protocols like SOAP or UDDI, despite concerns about waste of bandwidth because of the "verbose" nature of the representation as ASCII characters.

A flat file database can consist of a single file which contains all data that an application accesses, or it can come as several files, each of which is a complete XML document. In the latter case, the application must also manage the directory structure and the filenames used for the individual files.

In the literature related to XML databases, the "flat text XML database" is sometimes not even mentioned at all as a way of managing XML data, despite a number of obvious advantages (see also [Graves02, p. 18]):

- The database is often far smaller than an equivalent representation of the data that uses a different type of storage. Typically, these other representations introduce overhead which is larger than the overhead caused by the verbose XML format.
- An XML file is easy to create, since the data only needs to be written to disc in the standard format.
- An ASCII (or Unicode) file is also easy to update, in the sense that the only required operations to be supported are insertions and deletions of characters from the file.
- This is the only type of storage which allows a human to edit the document with standard tools, i.e. a normal text editor.
- Since XML-based communication always uses this format, there is no need to convert the data before sending it to another application.

Nevertheless, of course the tendency of XML database technology to ignore this type of data storage is justified, due to several serious disadvantages:

- Modifying the database is very expensive. Today, operating systems do not normally support the insertion and deletion of data in the middle of a file, so an implementation would have to re-write all data after the point of insertion/deletion in the file. With databases whose size ranges in the gigabytes or even terabytes, this is prohibitively slow.

- There is no way to navigate quickly through the document – if the program that accesses the file is not interested in the contents of an element, but wants to e.g. "jump" directly from the start to the corresponding end tag of an element, there is no way to achieve this; the whole content of the element must be read. Again, this makes flat text files unsuitable for very large databases.

- Many features commonly needed for databases are not present, including ACID properties (accesses are atomic, consistent, isolated and durable), transactions, access control, recovery and many more. Only for very simple applications, the operating system's abilities in this area (file locks, file access properties) might be sufficient.

In the light of these arguments against flat file databases, we have to conclude that they are not normally of interest for the efficient storage of XML data; they do not only cause problems in the case that the file needs to be updated, but read accesses are equally problematic, unless they happen to read the entire file sequentially.

All in all, such text file databases can still be useful if the XML documents are very small, and often accessed as a whole. Additionally, some of the approaches discussed in section 3.6 allow the creation of index structures which can be stored separately from a plain-text XML file and which contain offsets into the file – as long as the file changes never or only seldom, these indexes can significantly speed up read accesses.

As we have learned in section 2.3, the XL variables which we want to store on disc in the practical part of this thesis can both get very large and are also modified frequently, for example with "**insert**" or "**delete**" statements. This makes storage as flat text files an unlikely candidate for the XL storage subsystem.

3.2 Single Tokens

When looking at an XML document in its plain-text form, it quickly becomes obvious that as soon as we abandon the thought of also maintaining the database in this plain-text form, we can save a significant amount of complexity in the code that handles the XML data. The reason for this is that as long as the standard text format is used, the program must repeatedly read in character by character and interpret it, for example to distinguish between the different types of tags.

Furthermore, when the document is not saved in text form, there is no necessity to store the tokens immediately one after the other – a positive aspect, because in the previous section, storing the parts of the document adjacent to one another was identified as a cause of problems when making modifications to the document.

This section gives an overview of different ways to store single tokens on disc, particularly concentrating on using existing solutions like a relational database system. In sections 3.3 and 3.4, these ideas will be used to build more complex on-disc data structures.

3.2.1 Tokens as Records

Modern database systems can be divided into several layers, each of which deals with the stored data in a more abstract way: At the lowest level, the data is handled in pages whose size is determined by the hardware the database system runs on. The layer above it provides support for records; each record is identified by a record number and can hold a number of bytes. Usually, there is support for records which can get larger than one page. Finally, further layers provide platform-independent support e.g. for storing the data types supported by SQL, or for the transformation of high-level SQL queries into accesses to the lower layers.

Assuming that we have access to the "record" layer of a database system, it is possible to store an array of bytes in a database and subsequently identify that new record by a record number that the database system generates. Of course, similar functionality is also available at a higher level (see section 3.2.3 below), but much of the additional functionality provided by the upper layers is not useful for our purposes – on the contrary, it only introduces additional overhead and makes the operations slower.

The record layer has several important properties: It is possible to store arbitrary data as a sequence of bytes, the resulting record is uniquely identifiable via its record number (which never changes), and there is usually no limit to the size of a record. This way, each token of an XML document can be stored as a record, and references to tokens take the form of a record number. The sequence of bytes that represents the token will typically include the following fields:

- Token type (a code, for example for "begin element", "end attribute", "PCDATA")
- Additional data for this type of token. This is dependent on how exactly the tokens are stored (see section 3.3 below), but could include things like the tag name of an element, or type information about it.
- Bookkeeping information, for example the record numbers of other tokens, such as the record number of the token that follows this token in the document. Again, this is discussed below in detail.

Unfortunately, many database systems do not give access to the record numbers they use for storing the data, which can make the implementation of this type of token database impossible. One example for a system which does allow access to its "logical record numbers" is Berkeley DB, a low-level library for creating embedded databases.

3.2.2 Tokens as Objects in an OODBMS

In principle, an object-oriented database management system (OODBMS) is a good way of storing the tokens: Each token is represented by an object which is stored persistently in the database. References to other tokens can be implemented directly as object references without having to worry about finding a representation for them, such as the record numbers of the previous section.

Storing tokens this way is straightforward: After choosing a data structure, for example a linked list, the data for each token is stored in an object which is subsequently written to disc by the OODBMS.

The disadvantages of using an object-oriented database are primarily non-technical: OODBMS have a smaller market share, different companies' implementations are often not compatible with each other (so large applications which use a particular OODBMS will be dependent on the company providing the database system), and existing applications are usually not written for object-oriented, but for relational databases.

3.2.3 Tokens as BLOBs in an RDBMS

Apart from storing tokens as records and as objects in an object-oriented database, another way to maintain a token database is to use a relational database management system (RDBMS) and to map the data to one or more relations whose tuples describe the tokens.

This solution has several practical advantages: Relational databases are in widespread use, there is a range of both commercial and non-commercial products to choose from, and it is possible to use them from almost any programming language. Furthermore, by now even the free solutions offer advanced features like transactions, recovery and concurrent access, all of which can be put to good use by a token storage system.

Unfortunately, using relational databases for our purposes is made difficult by the fact that SQL, the structured query language supported by these systems, only includes a limited number of simple data types, and that the layout of all tuples in a table must be identical, whereas the layout of the tokens we want to store in the table differs from token to token – for example, a "begin element" token might include a token name, whereas other tokens do not. Thus, if only the standard SQL types are supported, an RDBMS is not very well suited for storing *single tokens*. Section 3.5 discusses ways of storing the *entire XML document* in a relational database, but the ideas in that section are based on a completely different approach to representing the XML document, and do not break it up into tokens.

Storing tokens is much easier if the database supports a common extension to SQL, in the form of the BLOB data type (*binary large object*, also CLOB for DB2 or LONG for Oracle).

Essentially, this allows the user to store sequences of bytes in the database, and as implied by the name, the size of these sequences is not limited to the size of one page.

Using the BLOB interface, the data for a token can be stored in the database in the same way as described above in section 3.2.1. The only difference is that the relational database system does not give access to the record number under which the BLOB with the token contents is available, so we need to create an extra "token ID" in the form of an integer to index the tokens. The resulting relation to be created in the RDBMS is simply a mapping from token ID to token data:

tokens: id:integer → data:blob

Generating a new token ID can be achieved by taking the maximum token ID of all tokens stored so far, and increasing it by one.

This way of proceeding is largely identical to the solution described in section 3.2.1 with its direct record access. In particular, it should be noted that many of the advanced features provided by relational databases cannot be fully exploited:

- The use of a BLOB for the token data makes it impractical to create an index over the token data or to have the RDBMS search through it, which means that queries like "return all tokens which begin an element named <tag>" cannot be performed by the database, but must instead be implemented by the code which manages the token database.

- Similarly, the choice of a BLOB makes it impossible to use the RDBMS directly for queries. For example, the query "return all children of this element" would need to examine whether a "parent" field in the BLOB has a certain value.

To alleviate at least some of these problems, a slightly different approach can be taken: All the fields stored inside the data BLOB which do not vary from token to token, such as references to the previous or next token in a token stream, can be stored as separate fields. All other fields still need to be stored in a BLOB. The following schema would allow us to use the RDBMS to search for all children of a node, by examining the parentid field:

tokens: id:integer → previd:integer, nextid:integer, parentid:integer, data:blob

Other attempts to get rid of the BLOB and store everything using SQL's basic types mostly result in an awkward and inconvenient organization of the data:

- The table of tokens could include fields for all possible data types. For example, it could include a "token name" field which is set to the token name for "begin element" tokens, and to NULL for all others. But apart from wasting a lot of space for storing the NULLs, this can also easily lead to inconsistencies when updating the table.

- We could introduce separate tables for each kind of token, i.e. one "begin token" table, one "XML comment" table, etc. However, this would mean that lookups for a token would have to search in all of the tables if the token type is not known. Furthermore, as we will see in a later section, the number of such tables would be quite high because of the large number of primitive data types that an XQuery implementation must support.

- Similar to the point above, we could introduce separate tables for each token type, but not store all of the token data in them, only the part of it that is specific to this type of token. In other words, the token ID can be used to access the common token data in the main token table, and an additional "data ID" then identifies an entry in one of the other tables:

  tokens: id:integer → previd:integer, nextid:integer, parentid:integer, dataid:integer
  begindata: dataid:integer → ...
  commentdata: dataid:integer → ...
  ...

  Apart from the fact that the additional step of indirection is not as efficient as direct access to the data, this approach still suffers from the fact that one field (dataid) is an index into one out of a range of other tables. On the other hand, it is better than the solution proposed in the previous paragraph as long as only the "common" data fields of the tokens are accessed.

In summary, the use of a relational database system to store tokens is a possibility, but implementing it does not have any significant advantages over a token storage system which uses direct record access – it is even to be expected that using the high-level interface to the database system will only introduce additional overhead.

3.3 Token Stream

To avoid repeated parsing of the characters in the document for every access to it, we can perform the parsing only once and store the data as a stream of tokens, turning these back into characters when delivering them to the application again. This way, in the frequent case that an XQuery implementation needs to search through a significant portion of the document, but only returns very little data to the application in the end, it can take advantage of the faster access times of the pre-parsed token data.

Section 2.1.2 describes how an XML document can be interpreted as a stream of tokens. The following sections discuss alternatives when describing a document with tokens and possible

improvements to the token stream format, and list a number of different data structures which allow for efficient storage and retrieval of the tokens.

3.3.1 Mapping XML Documents to Token Streams

To repeat the points made in section 2.1.2, here is the list of tokens that the implementation has to distinguish between:

- Processing instruction
- Comment
- Start tag (including its attributes)
- End tag
- Textual content (PCDATA for entity text content, CDATA for attribute values)

Note that a special empty-element tag is *not* needed – as mentioned before on page 12, an XML parser need not distinguish between an empty-element tag and a start tag that is immediately followed by an end tag. Thus, a straightforward opportunity to reduce the complexity of any code that deals with the token stream is to convert empty-element tokens to start tags followed by the respective end tags during stream creation.

The document's text data is represented by special tokens for PCDATA and CDATA. When turning the text into corresponding tokens, there are two different ways to proceed:

- Every text token contains just one character. This has the advantage that operations like deleting parts of the text or adding characters to the end are simple, and that tokens stay small. The disadvantages are the larger number of tokens, leading to a significant amount of additional space as well as runtime overhead, and the fact that an XQuery implementation which is not interested in the text cannot just skip to the end, but has to access the tokens character by character to reach the end.
- Every text token contains all the characters between the tag that precedes and the tag that follows the text. In this case, adding new text to an existent text token becomes slightly more difficult, and the maximum size of the text data in the token is unlimited, which requires special attention from any program processing it – for example, the program may not be able to load all of the token into memory simultaneously. Additionally, an XQuery engine must take care always to merge two adjacent text tokens into one. On the other hand, storing text this way results in a lower number of tokens and a correspondingly lower

amount of overhead to manage them. Furthermore, if an XQuery engine is not interested in the text data, it can just move to the next token to skip all of the text.

The XQuery standard implicitly encourages that text data is stored and addressed as a single entity, e.g. by mentioning the necessity to merge adjacent text tokens – all in all, storing text as one big token should thus be considered a superior solution to the alternative of storing it character by character.

3.3.2 Variant Mapping Schemes

Using the tokens above, any XML document can be converted to a token stream and vice versa without loss of information. However, in practice it makes sense to introduce additional tokens, for instance to avoid that some types of tokens contain too much information which is better handled separately as a series of several tokens. In particular, start tags can contain an arbitrary number of attributes, so they can get very large. The following additions to the above list of token types seem worth considering:

- An "attribute token" can be introduced. It appears between the start tag and end tag of the element to which the attributes belong – to ease processing the token stream, it is advisable to demand that attributes always appear immediately after the start tag token, before the tokens for the actual content of the element.

- Both element names and attribute names use the same type of string to store the name; in the XML specification, the grammar non-terminal that refers to them is called a Name. Because of the similarity in handling of these two types of Names during XQuery query processing, it may be advantageous not to include this name information in the respective start tag or attribute token, but as a separate "QName" token which appears after the token.

- To make processing of elements and attributes still more uniform, it is also possible to represent attributes as separate "begin attribute" and "end attribute" tags which enclose the attribute's name and value.

- Finally, while XML itself treats all data as characters, making it possible to use this representation for all of the document, [XMLSchema2, 3.2] introduces a number of primitive data types. An XQuery implementation that adheres to the type checking rules laid out in the XQuery standard has to store type information for every element, so it could theoretically deduce the type and convert the PCDATA to the appropriate primitive type on every access. Still, for improved performance it will typically use a special token for some or all of the primitive types. They are: string, boolean, decimal, float, double, duration,

dateTime, time, date, gYearMonth, gYear, gMonthDay, gDay, gMonth, hexBinary, base64Binary, anyURI, QName, NOTATION.

3.3.3 Data Structures

After a look at the different XL document modification statements on page 29, we can see that the data structure we use for storing the token stream should have efficient support for insertion and deletion of an arbitrary number of tokens into/from the stored stream. "Efficient" in this case means that we do not want the cost of insertion/deletion to grow proportionally to the number of tokens that follow the insertion/deletion point in the document, as was the case when storing the XML document as a flat text file. Instead, the cost should only grow (roughly) proportionally to the number of tokens that are inserted or deleted. With this limitation in mind, there are several different data structures that can be used:

Singly Linked List of Tokens The token stream is represented as a list of token objects that are chained together in document order with a "next" reference which for each token points to the token that follows it. When an XQuery engine requests the token stream, the "next" references are traversed and the tokens returned in the right order. When inserting or deleting tokens, only a few "next" references need to be modified. However, when using this data structure, insertions and deletions can be problematic because to perform them, the XML storage system needs to know the token that *precedes* the point of insertion/deletion to update its "next" reference – the XQuery implementation must be able to provide this token.

It should be noted that the "references" used to retrieve tokens are not Java-style references or C-like pointers, but references to on-disc items, as detailed in section 3.2.

Doubly Linked List of Tokens As above, but the data for every token contains a "previous" as well as a "next" reference. The presence of the "previous" reference makes it much easier to perform insertions and deletions in the document data, since e.g. a reference to the first token to be deleted is sufficient to perform the deletion. On the other hand, the "previous" references take up additional storage space. Just like for singly linked lists, the cost for inserting or deleting tokens only grows proportionally to the number of tokens inserted. Most of the time, a doubly linked list will be necessary to support modifications to the token stream, because XQuery implementations will only be able to provide e.g. the token to delete, not the one that precedes it.

Doubly Linked List of Chunks of Tokens In order to reduce the amount of bookkeeping overhead and thus to achieve a bigger storage capacity as well as improved access times, it is possible to store many tokens in one "chunk" without extra "previous/next" references. Only

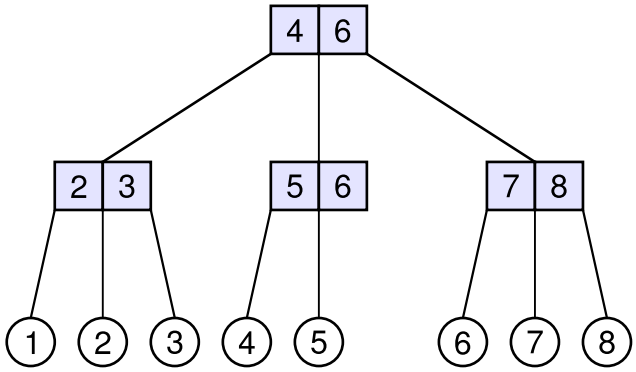

Figure 4: *B^+-tree with 8 tokens, indexed with integer keys. Even though no tree reorganization is necessary, inserting a new token after token 5 is expensive because tokens 6, 7 and 8 must be renumbered.*

each chunk of token data contains two references to the previous and next chunk. The pros and cons of such a solution are discussed separately in section 3.3.4 – generally speaking, it can result in improvements, but also implies an increased complexity of the code that handles the token stream.

B^+-Tree of Tokens Finally, the token stream can also be stored in a B-tree or B^+-tree. When using it, tokens which are adjacent to each other in the stream should also be adjacent to each other in the tree – this way, a significant increase in the speed of sequential reads can be expected.

Since one of the main ways of traversing the tree will be to read the elements in the order they appear in the tree, the normal B-tree is not as well-suited as the B^+-tree – with the latter, the leaf elements are chained to each other using "previous/next" references, making a sequential read of the token stream as efficient as with a linked list. Additionally, data is not stored in internal nodes of the tree.

When using a B^+-tree to store the tokens, the question arises what to use as the key value of the tokens. A first approach which gives the desired result of making tokens appear in document order in the tree would be to use an integer ID and to keep increasing its value for each token that is appended to the token stream. Unfortunately, this only works as long as no tokens are inserted in the middle of the document, because the key values of the inserted tokens need to be larger than the key value of the token that is to precede them and smaller than that of the token that follows them. For example, when trying to add a new token after token 5 in the (2,3) tree in figure 4, there is no choice but to insert the token as the new token 6, and to increase the key of the old tokens 6, 7 and 8 by one. However, the complexity of this solution is not acceptable: Every token after the point of insertion must be modified – and this has to be done even though

the B^+-tree itself does not require any reorganization! Thus, an insertion operation is in O(n), where n is the number of tokens in the document.

After observing this problematic behaviour, the next idea is to make the renumbering of tokens happen as seldom as possible. A simple way of achieving this is to allocate the integer token numbers "sparsely" at first, for example as 2, 4, 6, 8, 10,... then, when a token is to be inserted after token 6, no renumbering is necessary.

On second thoughts, however, this does not necessarily result in any increase in efficiency. Even if tokens were later inserted exactly in such a way that they would fill in the as yet unallocated token IDs, the frequency of renumbering operations would still only be reduced by a constant factor, which means that insertion operations are still in O(n). Furthermore, it is unlikely that insertions will happen in such a way that even this factor is reached, because of the way XML documents are usually built up by an application: First, the application creates an almost empty document, e.g. just a start tag followed by an end tag, then it keeps inserting new elements before the end tag. This means that the "spare room" for IDs between the end tag and the token before it is always used up quickly again. On the other hand, with this type of access pattern (modifications near the end of the document), the renumbering operation is cheap, so the overall performance of the data structure might be acceptable.

After these observations, it is necessary to take a step back and to reconsider the choice of an integer as the key for tokens. Clearly, it is not possible to store monotonically increasing key values with each token without having to perform a potentially very expensive renumbering operation from time to time. The goal of storing the tokens in document order in the tree could be abandoned, but the result would be identical to accesses via logical record numbers as described in section 3.2.1.

However, it is possible to achieve the desired properties of the data structure by modifying it slightly: We omit the key value of the tokens altogether, to avoid the problem of having to renumber the keys. Of course, a new problem is introduced at the same time – how can you specify the location of a read or modification without specifying a key value? The solution is not very complicated if only XQuery operations are to be made possible on the stored token stream: It is sufficient to implement support for a cursor, a "pointer" into the tree which can be moved from one token to its immediately preceding and following neighbours. An XQuery implementation always has to navigate through the document to find the destination of an insertion (or whatever operation has been requested), so the cursor can be moved to the right position at the same time. Internally, the cursor can represent the position in the tree using references to the chunk of data the token is in, and its position within its chunk.

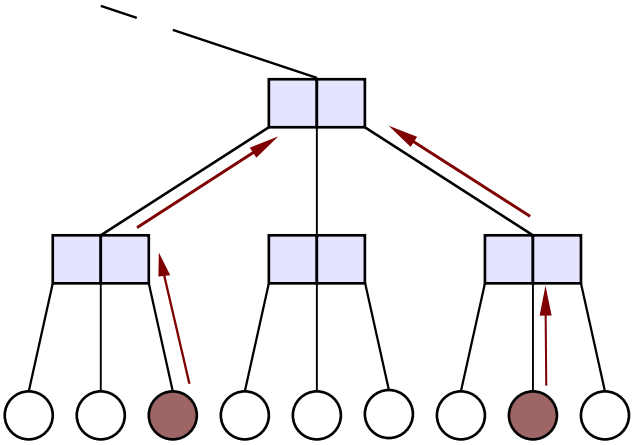

Figure 5: *By going upward from two leaves in the B^+-tree, we can determine which one appears earlier. The cost of this operation is $O(\log n)$ if n is the number of leaves in the tree.*

Once the position of an insertion and deletion has been given with the help of a cursor, the rest of these operations uses the same algorithm as the original B^+-tree.

The time taken for insertion and deletion of a token in a B^+-tree is proportional to the logarithm of the number of elements in the tree – it is not constant like with a linked list. However, this slightly worse complexity only occurs when making modifications, which are fairly rare for typical access patterns (a commonly cited rule of thumb is that 80% of all accesses are read operations). Furthermore, it is outweighed by a number of pleasant properties of the B^+-tree:

- Tokens which are next to each other in the token stream are also stored next to each other on disc, i.e. in the same chunk. When using the tree to store the tokens of an XML document, this is even more of an advantage than with other types of data, because sequential reads of shorter or longer parts of the document are very common.

- The B^+-tree is optimal for sequential reads, due to the way the leaf elements of the tree reference each other in a linked list.

- Given two cursors pointing into the tree, it is a quite cheap operation ($O(\log n)$ if the tree contains n tokens) to find out which of the two is nearer to the start of the token stream. This may be of interest for some implementations of an XML storage system because it can be used to implement the “is in document order” comparison of XQuery.
  To compare two tokens, we store a reference to the parent chunk for each chunk in the tree, and then later move upwards from the leaves corresponding to two tokens which are to be compared. As illustrated in figure 5, after zero or more upward steps to the parent of the

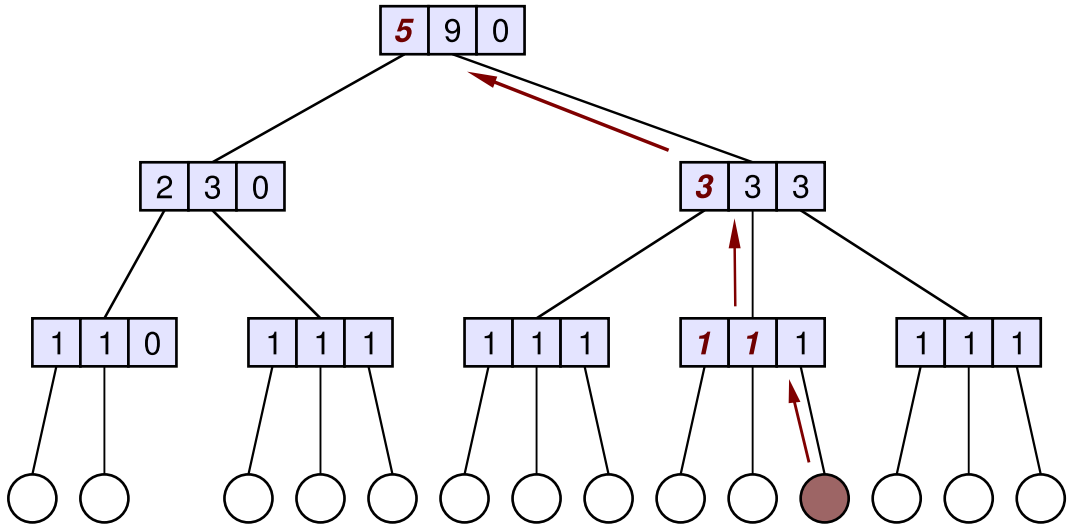

Figure 6: *To find the position of the marked leaf in the tree, each chunk entry needs to record the number of leaves stored "beneath" the entry, so we can go upward from the leaf, accumulating the left siblings on the way. In this example, the marked leaf has position* $(1+1)+3+5=10$*, i.e. it is the 10th entry in the tree, counting from zero.*

current chunk, the paths from the two tokens end up in the same chunk. At this point, it can be determined which of them appears earlier in the document by searching through the child references of the current chunk and finding out which one of the two paths reached it via an earlier child.

- With an additional small modification to the data structure, the cost of larger numbers of these "in document order" comparison operations (e.g. while sorting tokens in document order) can be reduced further. The idea is to find out the absolute position of tokens in the tree with a cost of $O(\log n)$. After that, the "in document order" comparison is reduced to a single integer comparison of the two token positions.
  In order to allow the position of a leaf in the tree to be determined, the number of tokens which is stored in a sub-tree is stored alongside the reference to the chunk that represents the top-level node of that sub-tree (see figure 6). Later, we move upwards from the token whose position must be found and for each chunk on the way to the B^+-tree root accumulate the number of tokens in the siblings to the left of the current chunk. Once the top-level node is reached, the position of the token is known.
  This modification to the B^+-tree has the disadvantage that an update always "ripples upward" from the inserted value to the very top of the tree, and never stops earlier, because the token count for each chunk must be updated. This means that the tree root has to be up-

dated on each insertion/deletion, which can adversely affect the performance during many concurrent modifications to the data.

- The same modification of the data structure also allows the retrieval of the token at any fixed position in the document, e.g. "return the 325th token in the document". Whether this feature is useful depends on the application, but it could be used to represent pointers to tokens in a much more compact way than with cursors. Obviously, the positions of some tokens change after an insertion or deletion, so the application either needs to take this into account or consider the positions invalid after a modification.

The reasons why the last modification above was discussed is that the XQuery implementation by XQRL, Inc., which is used for the practical part of this thesis, by default likes to attach a single integer ID to each "begin element" or "begin attribute" token and later uses these to sort tokens in document order. Another thing to note about this modification is the fact that an implementation of it is available in the popular Berkeley DB database library, in the form of its "recno" access method with mutable record numbers.

A final aspect of storing an XML document's tokens in a B-tree has not yet been looked at: Should the leaves of the tree just contain record numbers which identify the stored token, or should the whole token be stored in the leaf chunk? In the interest of efficiency, it would be desirable to eliminate the additional indirection through the record number and to directly store the token data in the leaf chunk, but as we have noted earlier, the size can differ a lot from token to token, and some tokens might even get larger than the size of the whole chunk. Consequently, it is impossible to store whole tokens as the leaves – however, depending on the access patterns, it might be a good compromise to store some basic data (token type, references to other tokens) in the tree leaf, and reference the rest (in particular PCDATA and element/attribute names) in an additional record.

3.3.4 Granularity of Token Storage

In the previous sections, solutions were outlined for ways to store token streams in databases. In those sections, focus was primarily on coming up with data structures which allow efficient insertion and deletion of tokens from the middle of a document. This section, while not giving up the goal of efficient modifications, instead concentrates on reducing the overhead introduced by the data structures.

As an example, consider storing a simple "end element" token. Apart from the fact that the token has the type "end element", no further information needs to be stored, so theoretically, a single byte is sufficient. In comparison to this, the overhead of storing the byte can be enormous:

Assuming that logical record numbers are only 32 bits wide (64 bits will often be a more realistic value) and that a doubly linked list is used, 8 more bytes of data are added to the token. The more elaborate data structures described in section 3.4 would add yet another 4 or 8 bytes to this. Finally, if the token is stored in a typical record-based database, another 2 bytes are used up for the offset of the tuple within its page, and another 12 bytes or more per record for the mapping of logical record numbers to locations on disc (i.e. segment, page number within segment and tuple number within page). Further overhead, caused by other data structures that the database maintains and by fragmentation, is more difficult to measure – but even when ignoring this, the overall cost of storing a single byte amounts to 31 bytes or more!

Reducing Overhead by Increasing the Storage Granularity There are two aspects to reducing the amount of space taken up by an XML document in the database system:

- Do not store single tokens in the database, but chunks of tokens. Then, instead of a doubly linked list of tokens, only maintain a doubly linked list of chunks of tokens.

- Create data structures to manage these chunks of tokens, which permit insertions and deletions anywhere in the chunked data.

To remove as much about the XML storage system's overhead as possible, we eliminate the "previous" and "next" references between individual tokens. The fact that a token no longer corresponds to a record also means that it is no longer necessary to create a record number for each token, and thus to maintain a large index of record numbers. Instead, many adjacent tokens are combined in one large chunk of data, and only the chunks are linked to each other with "previous"/"next" references.

Each chunk can only be read from the beginning; to access the contents of a token in the middle of the chunk, the data of all tokens before it must be decoded at least partially. The idea about using chunks is that performance is unaffected or even improved for the very common access pattern of a sequential read of the token stream. On the other hand, other types of operations become less efficient and more complex to implement:

- Insertions may cause the chunk to become too large, so it may have to be split into several chunks.

- Similarly, if many deletions take place, the chunks could become too small, so the benefits of packing many tokens into chunks would be lost. In this case, several smaller chunks will have to be combined into one larger chunk.

- Moving backwards in the stream is not possible. This is usually not too much of a problem, because XQuery does not require backward movement as long as the XQuery implementation keeps track of tokens' parents.

- It becomes difficult to generate "logical record number" style unique identifiers for tokens by which you can find their position in the stream again later. The main problem is that the physical position of the token (chunk number, position in chunk) *cannot* be used for this because it changes as modifications are made to the chunked stream, and a logical identifier *should* not be used because the mapping from logical token ID to physical token position would re-introduce the overhead we are trying to minimize by storing the tokens in chunks. Unfortunately, the data structures described in section 3.4 need such identifiers, for example to store in each "begin element" token a reference to the corresponding "end element" token.

There is another problem that needs to be dealt with: As mentioned before, some tokens (notably PCDATA tokens, but also comment tokens, the QName tokens that hold the names of elements, etc.) can get arbitrarily large. If they were stored "in-line" in the normal token data, a difficulty would arise: Some chunks of token data could get very large, which might conflict with limits imposed on the chunk size by the data structures in use. (This is the case with the solutions described below.) The only workaround for this would be to allow splitting the data for a single token across several chunks, which, while not impossible, would considerably increase the complexity of managing the chunked token stream.

Consequently, it is advisable not to let single tokens get very large. A simple solution which ensures this stores the data of PCDATA tokens in records which the database system manages separately from the chunks of tokens. However, in many cases this simple solution will be overkill because only very few PCDATA tokens actually become very large, and retrieving the text data from its record will be slow compared to reading in-line data from the token stream. To get the best of both worlds, it is a good idea to introduce two types of tokens for PCDATA (and QName, comment etc.) tokens: A "small" one which directly carries a limited amount of data in-line in the token stream, and a "large" one which refers to an additional record with the actual data.

Choosing a Chunk Size There are several issues to keep in mind when deciding about the allowed sizes of chunks:

- The limits on the chunk size could be specified in terms of bytes or in terms of the number of tokens in the chunk.

- The minimum size for a chunk could be zero, or it could be a certain value, for example half of the maximum chunk size.

- An absolute maximum chunk size could be imposed by the hardware or the data structure that is used to manage the chunks.
- The larger the chunk size, the less the average overhead per stored token is going to be, but on the other hand modifications will also take longer.

Finding the right values for the parameters is a difficult task, and should best be done by testing and benchmarking the XML storage solution in a real-world environment.

Data Structures For Managing Chunked Data Once the tokens have been serialized into a stream of bytes, they must be distributed over a number of chunks. Any solution must meet the following criteria to be optimally suited for our purposes:

- For simplicity, the bytes representing a token should not be distributed over two chunks – if this were the case, it would not be possible simply to start reading bytes and creating tokens at the start of any chunk.
- Storage should be segment-based: Because the whole data structure is intended to make sequential reads as fast as possible, consecutive chunks should also be stored in consecutive physical sectors of a segment on disc.
- Insertion and deletion of data at any byte offset in any chunk should be efficient, i.e. neither should require reorganization of the entire data following the point of modification.
- At the same time, fragmentation of the segments (and the amount of disc space wasted due to it) should be as low as possible.

One structure which might at first sight be able to meet these demands has already been mentioned: A normal B^+-tree can be modified so its "leaf chunks", i.e. the chunks which contain the leaf nodes, do not hold references to the data, but the data itself. The tree modification algorithms ensure that all chunks are at least half full all the time, and with an appropriate algorithm for allocating new chunks on disc, it can be assured that adjacent chunks are often also adjacent on disc.

On the other hand, a B^+-tree which is used this way also has a limitation: It is only intended for the case that all leaf nodes have the same size. With an XML token stream, some tokens will only need one byte whereas others might occupy dozens or even hundreds of bytes. In order to be able to stick to the standard algorithms for inserting into and removing data from the tree, one would have to assume the maximum possible size for all tokens. For example, if the maximum possible size of any token is 32 bytes, even a single-byte entry would occupy its own 32 bytes

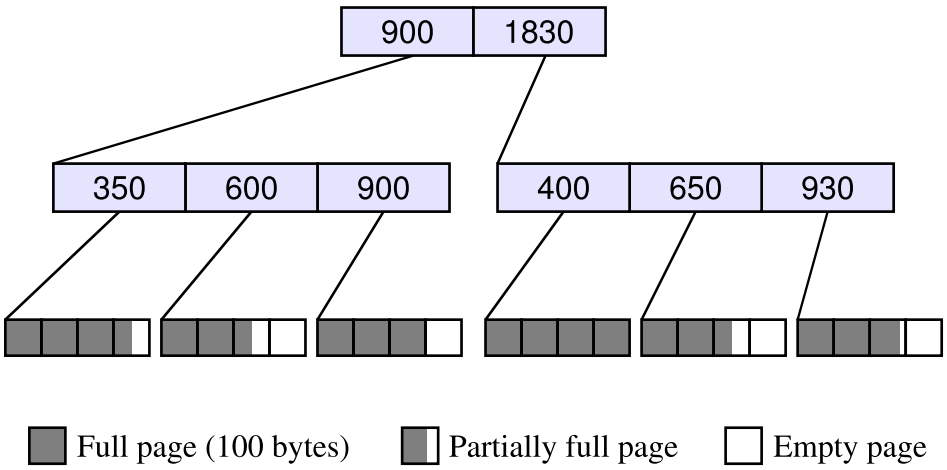

Figure 7: *1830 bytes of data, as stored by the EXODUS Storage Manager in six segments of four pages each. The page size is assumed to be 100 bytes*

of storage – 31 bytes of these would be wasted. Clearly, a B^+-tree is not the ideal solution for storing a chunked token stream in an efficient *and* space-preserving way.

Still, the idea of using a tree structure to manage the chunks of data is important, and a number of different mechanisms for storing chunked data have been proposed which use some kind of tree. [Biliris92] contains an interesting comparison of three of these storage structures and includes both a look at the algorithms and benchmarks for a range of different accesses. Below is a short overview of the three storage managers, EXODUS, Starburst and EOS.

EXODUS Storage Manager (ESM) The EXODUS Storage Manager supports the storage of large objects of unlimited size. The data is stored in a series of segments, each of which consists of a number of consecutive blocks on disc. The size of these segments (which correspond to the chunks of data we want to store the XML token data in) can be chosen by the application. During an update operation, the data in a segment needs to be moved to make room for new data or to overwrite deleted data – for this reason, the segment size should be smaller for data with a higher percentage of updates.

EXODUS indexes the segments of data with a tree structure not unlike a B-tree. The basic idea of this structure is illustrated in figure 7 with an example: The data is stored in segments which appear as the leaves of the tree. The size of a page is (unrealistically) assumed to be 100 bytes to make the example easier to follow. All non-leaf nodes of the tree contain a series of references to sub-trees as well as, for each such reference, the accumulated amount of data stored in the sub-tree and the sub-trees to the left of it. In figure 7, the first child of the root node contains

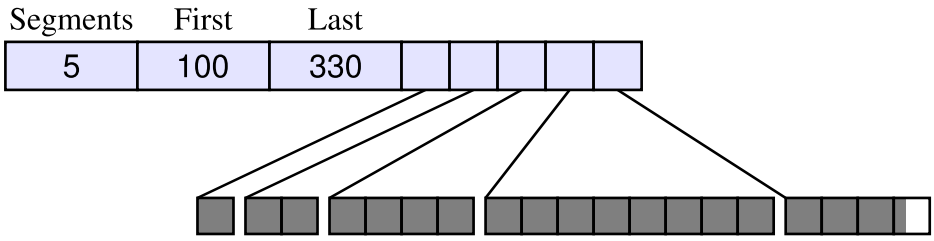

Figure 8: *1830 bytes of data, as stored by Starburst in five segments. Since the size of the object was not known in advance, Starburst started with single-page segment, then kept doubling the segment size for subsequent segments.*

900 bytes of data, and the second $1830 - 900 = 930$. Looking at the right child of the root, the first segment below it is completely filled with 400 bytes, the second one holds $650 - 400 = 250$ bytes, and the last one $930 - 650 = 280$ bytes of data.

When performing insertions and deletions, the data within each segment is always reorganized in such a way that all data is at the start of the segment, and any free space at the end. Just like for the B-tree, both the segments and the internal nodes of the tree are required to be at least half full all the time.

EXODUS does not include the "previous" and "next" references between segments that were described above, but by maintaining a stack which specifies the current position in the tree, access to the segments of the tree can be made as efficient as with these references.

The tests in [Biliris92] show that EXODUS can give good performance on all supported operations. However, it becomes apparent that it is usually very difficult to choose the right value for the algorithm's tuning parameter, the number of pages per segment: Large values give better read access times but also lead to larger amounts of wasted space, whereas smaller values cause read accesses to become slower because more disc seeks are necessary, but manage the available disc space more efficiently.

Starburst The Starburst manager for large objects is another data structure designed to allow the efficient storage of large byte arrays. Essentially, Starburst is nothing more than a simple data structure to keep track of where the chunks of data for a large object are stored. The underlying allocation system is assumed to be a binary buddy system, as explained e.g. in [Knuth73]. With its help, it is possible to allocate segments of consecutive sectors on disc, whose size is a power of two of the page size up to a certain, system-dependent limit. For example, if the page size is 4 kbytes, segments of 4, 8, 16, 32 etc. kbytes can be allocated.

If Starburst is faced with the task of storing a large object whose final size is not known, it starts by allocating only one page, then two, four etc. This behaviour is shown in figure 8 for

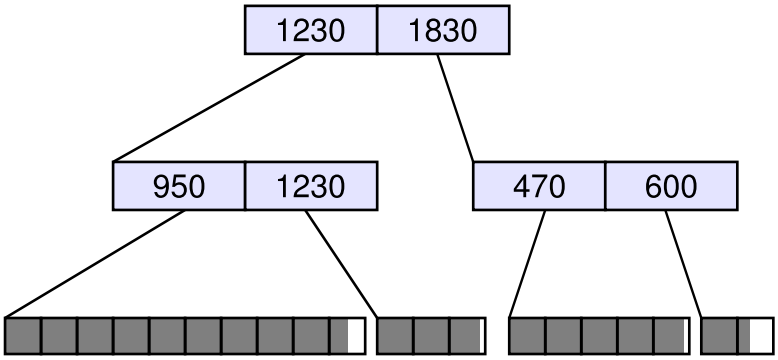

Figure 9: *When storing 1830 bytes using the EOS storage manager, at most one page per segment is filled partially, and a tree similar to EXODUS allows efficient insertion and deletion.*

an object which ends up 1830 in length. The Starburst data structure records the size of the first segment (1 page, again assumed to be 100 bytes), the sizes of the subsequent segments need not be stored explicitly because they can be inferred from the first segment's size. Finally, as soon as the final object size is known, the last, only partially filled segment is shortened to the smallest possible number of pages, and the number of bytes in this last segment also stored in the data structure.

In the case that a large object's size is known in advance, Starburst attempts to allocate the segments in a more intelligent way, using a segment of appropriate size, or a sequence of segments of maximum size. Again, the last segment is truncated to the right number of pages.

This explanation should make it obvious that Starburst suffers from a problem that might make it an inappropriate solution for an XML storage system: Insertions and deletions in the middle of the data are not possible in an efficient way; all the data that follows the insertion point would have to be copied to allow the modification to take place. Consequently, Starburst should only be used for data that changes rarely or never, or for which insertions and deletions only take place near the end of the data. According to [Biliris92], in scenarios where the data structure *can* be used, its simplicity and its property of not creating any fragmentation lead to very good performance.

EOS EOS attempts to combine the positive aspects of both EXODUS and Starburst by using a B-tree-like data structure like the former, but allowing segments which have to be (almost) filled completely, like the latter.

With EOS, the data is stored in a number of segments each of which consists of consecutive blocks on disc, again allocated using the binary buddy system. Unlike for EXODUS, each segment must be filled completely with data, except for the last page, which may only be filled

partially. Apart from this differing layout of the segments, the data structure with its tree of references and size information is identical to that of EXODUS. The tree in figure 9 is an example of what an EOS tree may look like after a few modifications to the data have taken place:

The length of the different segments can vary significantly from segment to segment because whenever an insertion or deletion takes place, the segment is split into two at the point of the modification. Because of the necessity to split segments, the insertion and deletion algorithm differs a lot from that of EXODUS.

If we kept splitting segments into smaller and smaller parts, the data structure could at one point deteriorate into a tree whose leaves are single pages – an undesirable state, since then the amount of space wasted because of partially filled pages rises, and sequential read times increase due to a non-sequential physical distribution of the segments on disc. For this reason, EOS allows the minimum size of segments to be restricted in the following way: If after an update, the size of a segment falls below a configurable threshold, then the segment is merged with one of the segments that are its logical neighbours in the tree.

The benchmark results in [Biliris92] show that EOS offers good performance both for read operations and for update operations. When storing an XML token stream, it offers all the flexibility outlined in the requirements near the beginning of this section. Furthermore, because (in contrast to EXODUS and Starburst) segments are allowed to get very large, an XML token stream storage system using EOS would be able to store even very large tokens “in-line” in the stream.

This section has shown that storing a token stream as a series of records, each of which holds one token, results in a large amount of bookkeeping information. To reduce the overhead, the token stream can be stored as chunks of data. The various storage managers introduced above help with the task of managing the data in such a way that modifications are possible in an efficient way. If an XML database which stores its data as a token stream is interested in the best possible performance, it should seriously consider storing the tokens in chunks.

3.4 Tree Representation

The previous section has introduced ways to store XML data in a way which encodes more information about the structure of the document: Whereas section 3.1 described storing the document as a series of characters without distinguishing between things such as start element tags or XML comments, the “token stream” idea pursued in section 3.3 allowed more efficient handling of the XML document by parsing the characters and creating higher-level objects – the tokens.

In this section, we take another step in this direction and use data structures which not only represent documents in tokenized form, but additionally encode the tree structure that all XML documents implicitly create.

But why is it necessary to store the data in a way which allows easy reconstruction of the XML document tree? As so often before, the reasons become clear after a look at the XQuery standard, and the way that both an XQuery implementation (executing queries) and XL (executing document modification statements) have to access the document:

- In many cases, an XQuery implementation does not want to read the entire token stream, but needs a way of quickly skipping certain areas. For example, when the query "$recipe/instructions" is executed, but an "<ingredients>" tag is encountered first in the top-level element of $recipe, then the XQuery engine is not interested at all in the contents of that element, but will want to skip its entire content.

- A similar situation arises for XQuery expressions which e.g. ask for the last child of an element; rather than going through all children to reach the last one, the implementation may want to skip to the end of the element first, and then go backwards by one child.

- For the implementation of queries like "child::*[position()=4]" (see [XQuery, 2.3.1]), the XQuery engine will want to directly navigate to the fourth token, and not skip the first three one by one.

- For the implementation of the XQuery "parent" axis (see [XQuery, 2.3.1.1]), it will be beneficial if it is possible to go "upward" in the tree and retrieve for any element the parent element that encloses it.

- With the XML "**insert**" statement, the user can specify where to insert the new data, and the different ways of inserting include to "**insert after**" a specified element and to "**insert into**" it, creating a new last child of the element. In both cases, the XQuery engine only provides the position of the "start element" tag and XL is required to jump quickly to the corresponding "end element" tag to perform the insertion. The same applies to the "**move**" statement.

For all these operations, the underlying XML storage system can only efficiently implement corresponding movements in the token data if it has knowledge about the document's tree structure.

3.4.1 Tree Variants

Before deciding on a data structure, let us summarize the operations it is required to support. The optional operations make an XQuery implementation's work easier for some queries, but will probably not affect the overall performance too much if not present:

- Read tokens sequentially in document order.

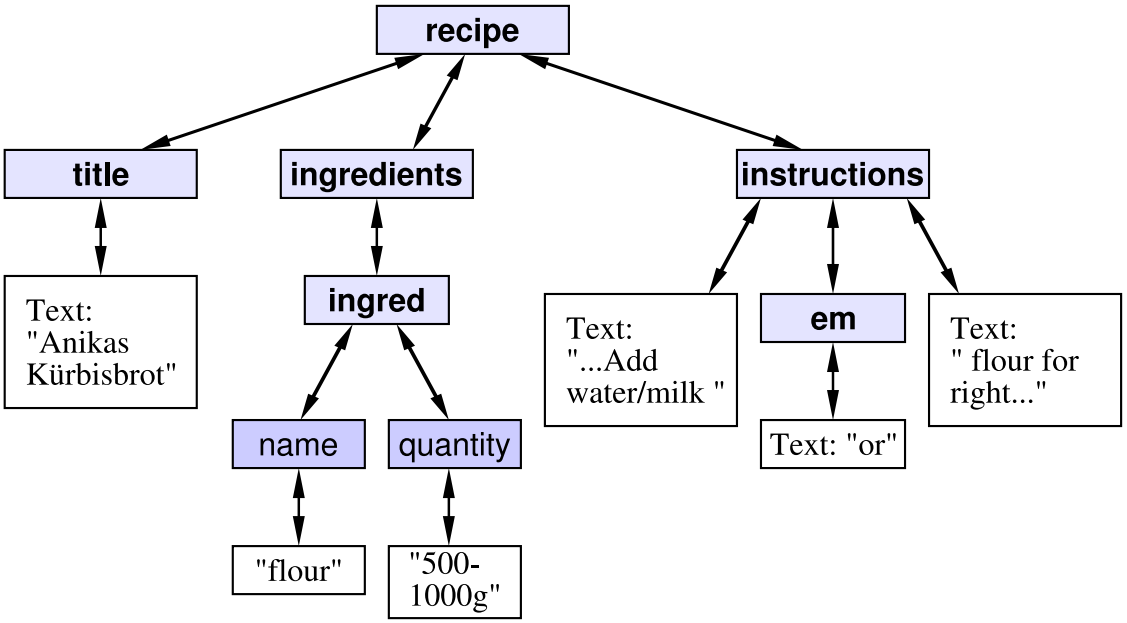

Figure 10: *With this tree representation, every parent holds references to all its children, and each child a reference pointing back to the parent. Surprisingly, it is not suitable at all for XML documents.*

- Jump from the beginning of an element to its end. (Alternatively: Jump from an element to its right sibling, and allow going backward from there.)
- Optionally: Move upward from an element to its parent.
- Optionally: Directly move to the *n*th child of an element.

There are many ways to represent a tree. In the rest of this section, a few of them are looked at and compared to each other, especially with their use for an XML storage system in mind.

All Children Reachable From Parent The type of tree shown in figure 10 is the "classical" data structure that one thinks of first when trying to represent tree-like data: Every non-leaf node in the tree contains a number of direct references to all of its children. Additionally, all nodes except the root contain a reference to their parent node.

In figure 10, no separate objects are shown for "begin" and "end" tokens of elements and attributes; for example, there is only one recipe and one quantity object. This reflects the fact that the data structure is not token stream based, but aims to be a direct representation of the document tree.

The tree in the example describes a "recipe" XML document based on the one on page 10 – for simplicity, parts of the document tree have been omitted.

When trying to store the XML document with this type of tree, a number of properties of the data structure become apparent:

A positive aspect is that direct access to the *n*th child of a node is possible, simply by following the *n*th child reference stored inside it. Additionally, all the operations listed above can be implemented, including backward movement in the document.

On the other hand, there are a few disadvantages to the data structure.

During updates, the size of a parent node changes whenever new children are added to it or deleted from it – because the number of children of an XML element can grow arbitrarily, it is not possible to allocate a fixed amount of memory for child references. As a result, whenever children are added or deleted, not just one, but *two* insertion/deletion operations have to take place; one to make room for the new child reference in the parent element, and one for the token data that is inserted or deleted. This is especially unhelpful if we consider that we do not actually need the second and subsequent child references to be stored in the parent node most of the time, the exception being optimization for a relatively rare query like "child::*[position()=4]" – for most applications, the costs of this property outweigh its positive aspects.

A further disadvantage of the data structure is that a speed/memory tradeoff decision needs to be made when implementing a cursor which points inside the document, and which is intended to be used for sequential reads of the document data:

- The first possibility is that the cursor only consists of a reference to the current token. In this case, moving it to the next document token is not as fast as it could be, because whenever the cursor wants to move to the right sibling of the current token, it must search through its parent's list of children to find out which *n*th child it currently points to, in order to be able to move on to the $(n+1)$th child. Because a linear search will have to be used and some documents could conceivably contain elements with millions of children, this is not an acceptable solution.

- The second possibility: The cursor also contains a stack of child offset numbers, which record the *n* at each level in the tree. That way, moving to the next token in the document is a fast operation which does not require any searches through the parent data. However, this also means that the cursor data does not have constant size, and can get very large for deeply nested documents – possibly even too large for it to be held in memory! Consequently, this solution is not suitable for an implementation either.

- An obvious third way to implement a cursor, and a compromise between the two alternatives above, is only to store the child's position for the last 10 or so levels. Benchmarks

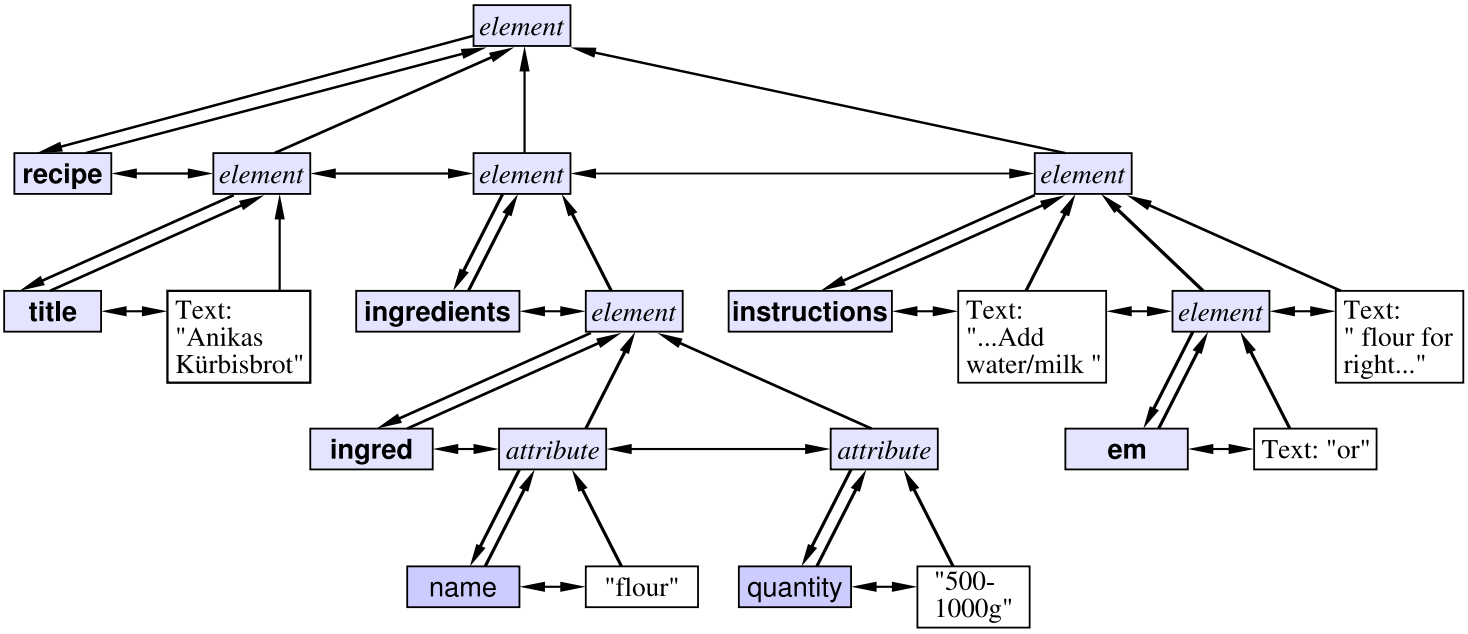

Figure 11: *With this tree representation, the parent only contains a reference to the first child, which is the start of a doubly linked list of children. The names of elements and attributes are represented as separate QName tokens.*

would have to be conducted to see how well this solution performs, but it seems probable that it would be quite fast for "average" XML documents. However, it is very easy to construct documents where it exhibits as poor performance as the first possibility above.

- A last attempt to make a simple sequential read of the data fast might be the following: Store the position n of the nth child with the child, alongside the reference to the parent. This seems an acceptable solution at first, although the additional 4 bytes or more of overhead per token are a considerable amount of storage. However, again there is a problem: n cannot simply be stored as an integer, because then inserting e.g. a new child at the start of a node would require that all children after it are renumbered (and again, there could be millions of them). Finding a different way of referring to the position might not be impossible, but will certainly prove a difficult task.

To summarize, this simple way of storing the XML document tree makes a sequential read – one of the most common types of access – difficult to implement, and with certain input documents the sequential read will either get relatively slow or require a large amount of memory. In practice, this means that the data structure should not be used for storing documents whose size exceeds the size of the main memory.

Node References First Child and Left/Right Sibling The previous section has shown that one common way to store a tree is not to be recommended for large XML documents. Figure 11 shows another commonly used way: Each parent node has only a single child reference, to the first child. The children are chained to each other in a doubly linked list. Furthermore, each child maintains a reference to its parent.

The graph in the figure contains separate nodes for the names of elements and attributes, but it would also be possible to include the name information in the parent node. Similarly, additional "begin element" and "end element" nodes could be added – in the example, they are not present.

This data structure does not have the disadvantages of the previous tree representation:

- Because each parent only contains a single reference to the first child, the size of parent nodes is constant. Additionally, the parent node does not need to be modified when a new child is added to it (except of course if that new child is added at the start).
- Inserting a new child is a constant-time operation, no linear search through a list of children is necessary.
- The "to next token" movement of a cursor can be implemented efficiently.

But there are also a few (minor) disadvantages:

- The nth child of a node is not accessible directly, all the $n-1$ siblings appearing before it must be visited to reach it.
- The implementation of the cursor's "next" operation will be relatively complicated, and in some cases will require the traversal of quite a few nodes to complete: Once the last leaf of a sub-tree (such as the text node "500–1000g" in the example) has been returned, the iterator must keep going upwards before it can jump to the sub-tree's right neighbour.
- It is not possible to jump to the last child of a node. This would prove useful to implement e.g. the XL **insert into** statement, which adds a new last child. Furthermore, it would allow the iterator to go backward in the tree.
  However, these operations can be made possible with a small modification: Instead of a doubly linked list of children, use a "doubly linked ring", i.e. the last child's "next" reference points to the first child, and the first child's "previous" reference to the last child. The first child can still be identified by the fact that the parent's "first child" field references it.
  (An alternative to this modification of the data structure would be to introduce a "last child" reference in the parent, but that would require additional storage space.)

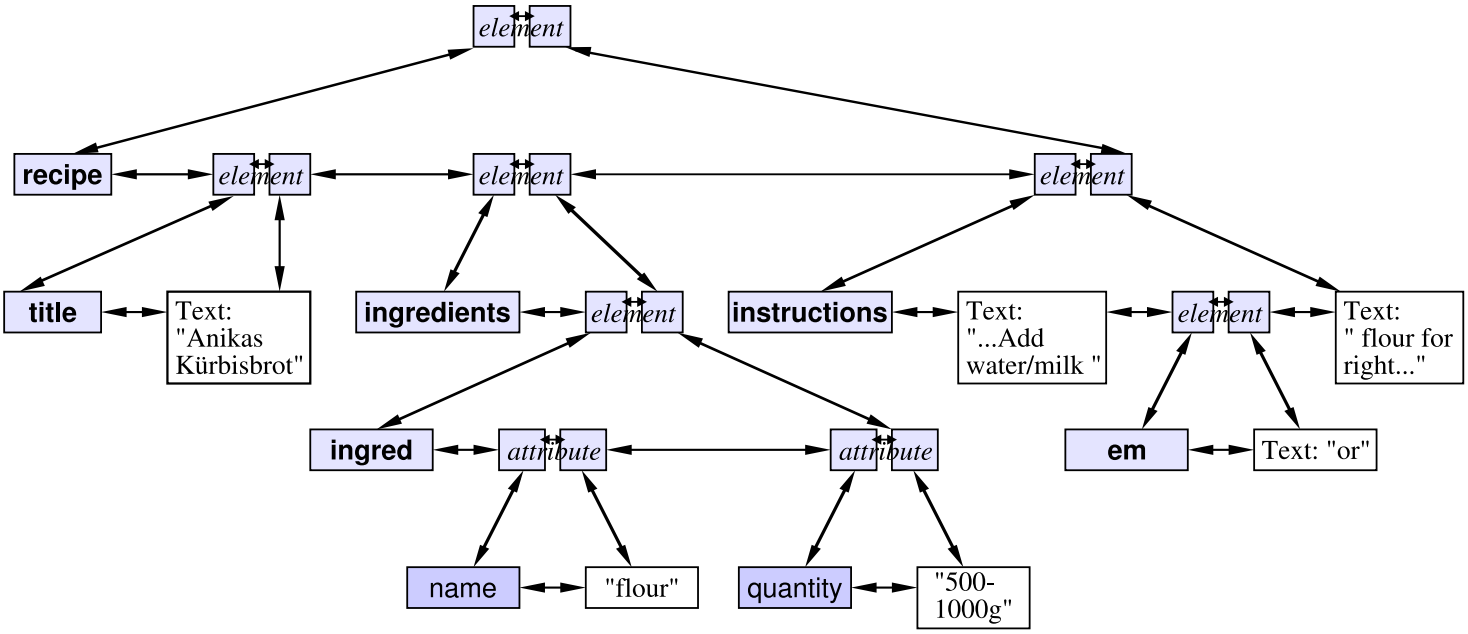

Figure 12: *In a doubly linked list, the tokens appear in document order. Additionally, the "begin" and "end" tokens of elements and attributes reference each other, making it possible to quickly skip the content between the "begin" and "end".*

This representation of the document tree is far better suited for storing the tokens in a database. Apart from the fact that only the first and optionally last child of a node can be retrieved quickly, and not the children between these, it outperforms the previously described "classical" tree representation for all operations, including modifications and sequential reads.

Token Stream With Additional References There is another way of storing the tokens of the document in tree form: First, we store the token stream (for example, as a doubly linked list) and then add further references from the first element of a sub-tree to its last element, to allow fast navigation through the tree. In figure 12, this is shown with the same example as before. This time, the contents of an element are assumed to be bracketed by corresponding "begin element" and "end element" tokens (the same is true for attributes), so in comparison to the two earlier cases, these nodes are split into two, with the left part representing the "begin" token and the right one the "end" token. The additional references which turn the linked list of tokens into a tree structure appear between the "begin" and "end" tokens. By using them, the required operations can be implemented:

- A sequential read of the token stream is very easy to implement – the cursor only needs to follow the "next token" reference.

- Jumping from the beginning of an element to its end is possible by traversing the reference from the "begin" to the "end" token.
- Insertions and deletions are possible due to the use of a doubly linked list.
- It is possible to go backward in the stream, which, apart from being necessary to insert a new last child, could be used for queries which ask e.g. for the second-to-last child.

The child nodes in figure 12 do not have references to their parent (except if the parent happens to be the next or previous token in the token stream) – if necessary, such references can be added, but this step should be considered carefully because of the additional space requirements.

Just like the previous tree structure, this variant does not allow direct access to all the children, and a linear search through the children is necessary to locate the desired child.

To conclude, this data structure is about as efficient as the previous one for most operations, but has an additional small advantage in that it is very much "token stream oriented" and thus well-suited for sequential reads of the data.

3.4.2 Storing the Tree On Disc

Most of the problems surrounding the storage of the tokens in a database on disc have already been covered in sections 3.2 and 3.3, so this section can restrict itself to describing the differences and necessary additions to support the storage of the above tree-like data structures.

When storing the data as single tokens with one record per token, the additional references, e.g. to parent nodes or the first child of an element, now need to be stored with the token data. Just like for the "previous" and "next" references, these references take the form of a logical record number, an OODBMS reference, or an index for a table of token BLOBs in an RDBMS.

On the other hand, when storing the document as chunks of tokens, we encounter a problem that has already been hinted at on page 48: The tree data structures require it to be possible to refer to tokens with a constant value through which the token can be retrieved even after it has been moved around on disc due to insertions or deletions. A logical record number would fit that description, but the per-token overhead introduced by logical record numbers is what prompted us to use chunk-based storage in the first place! There does not appear to be a completely satisfying solution to this dilemma – however, the following approach might be worth considering:

1. *Do* allow logical identification numbers to be created for tokens.
2. To save storage space, make every attempt to reduce the number of logical identification numbers that are actually *created*.

The logical ID will typically be an integer index for an (on-disc) array or B-tree, with the array/tree entry containing the physical location of the token: Segment number, page number in segment and byte offset within page. In the case of an on-disc array, a "free list" of deallocated record numbers ensures that their array entry is reused after the token they refer to has been deleted.

The entry for a logical ID has to be updated whenever the token's position changes. To ensure that this happens, the relevant operations of the data structures described at the end of section 3.3.4 will have to scan through the token data when copying it around, identify any tokens for which a number has been created, and update their entry.

How do we reduce the number of logical token identifiers? The normal "previous token" and "next token" references are not present for chunked token data (only "previous/next chunk" references might be), so not every token needs to be reachable via a logical ID. In the simplest case, only references to "end element" and "end attribute" tokens are needed, and consequently only "end element" and "end attribute" tokens need to be allocated numbers. (Depending on the requirements of the XQuery implementation, "begin element" and "begin attribute" tokens may also need to be given record numbers.)

A further significant reduction in the number of logical IDs could be achieved with the following trick: If the number of tokens between the "begin" token and its "end" token is very small (say, 5 or less), we can simply omit the reference to the end token (i.e. use a dummy value such as zero). The omitted reference would indicate to the storage system that the element/attribute is small and that the corresponding "end" token should be found not by looking up an ID, but by reading tokens sequentially until the next "end" token. Since a token lookup by logical number is potentially expensive (it might result in an additional disc seek), this solution might offer performance benefits even for larger values than 5 – the exact "break even point" should be determined using benchmarks.

The Natix Storage Manager Natix is an XML database project of the university of Mannheim, which includes a storage manager responsible for maintaining on-disc data structures which contain the XML data. The Natix storage manager is described in [KanneMoer99]. Geared towards its use with XML, it goes one step further than the solutions outlined in the part of section 3.3.4 which describe data structures for managing chunked data, and also takes the document structure into account.

Just like e.g. with EOS, the data is stored in chunks of tokens, allowing for a space-efficient encoding and for fast sequential scans. The size of each chunk could range from one to a few consecutive pages on disc, although the greater part of the paper only discusses a fixed size of

one page per chunk of data.

The special thing about the Natix storage manager is that it always stores one or more complete subtrees of the XML document in a chunk. This has several advantages: Nodes which are close together in the document also appear on the same physical page, and are thus quickly accessible. Moreover, it is easy to skip a whole subtree one is not interested in.

Similar to the other "chunked data managers" from section 3.3.4, the storage manager internally relies on a tree structure not unlike a B-tree to keep track of the chunks of data. If new nodes are inserted into the document and the resulting chunk gets too large for its page, it is split up and its nodes are redistributed over two chunks.

[KanneMoer99] does not discuss what to do if the single chunks holding the document subtrees only contain very little data, e.g. after many nodes have been deleted from the document. In this case, the other storage managers make provisions for a chunk "re-joining" operation which reduces the amount of wasted space, keeps logically adjacent data together and thus reduces access times. Before implementing a storage manager like Natix's, it might be necessary to evaluate whether such an operation is necessary for it.

This special, XML-oriented storage manager is an interesting addition to the other solutions for the storage of chunked token data, and well-suited to the task of storing XML documents in a way which includes information about the document tree. For our purposes, it is also of interest because it allows for a more efficient way of creating logical identifiers for nodes of the document tree, as long as it is sufficient only to be able to address individual subtrees, and not single tokens:

First, the physical location of a "begin" token at the start of a document subtree is easier to describe: No byte offset into a chunk of data needs to be maintained, only a number which identifies one of the subtrees stored in the segment. Second, when performing insertions and deletions, the storage system does not have to worry about updating the physical location for any subtree which does not move into a different segment, since its subtree number remains unchanged.

3.5 Mapping the XML Data to an RDBMS Schema

The previous sections have concentrated on finding special low-level data structures which are well suited for the storage of the semi-structured data in XML documents. The implicit thought behind this search was that already existing database software is not optimally suited for this task because it is based on the relational data model. However, for a number of reasons it is important also to look at the different possibilities to store XML data in RDBMS:

- Whereas XML storage solutions are very new, available relational databases are built on a range of well-understood, mature algorithms and technologies, so they can provide a

stable basis for any system which uses them. For example, existing relational systems offer features like on-line backups and recovery as well as advanced indexing schemes, and often scale better.

- Developing a database system from scratch is expensive and takes time; both the time needed and the cost can be reduced by building on already existent systems.
- Existing database installations are dominated by RDBMS, so it may be convenient to use them for XML as well.

Based on [FlorKoss99], the following sections introduce different mappings of XML data to relational schemas. Another mapping of XML data to a "fine-grained relational schema" is described in [Graves02, 4.2].

3.5.1 Edge Approach

A simple way to store an XML document in a relational database in a way which preserves its tree structure is with the use of an "edge" table, which contains the entire document structure data and which looks as follows:

Edge(source, ordinal, name, flag, target)

Each tuple in the relation describes an edge in the XML document tree, i.e. the relationship between a parent element/attribute node and its child. The meaning of the different parts of the tuple is as follows:

- *source* is an integer identifier for the parent node
- *ordinal* is an integer giving the child number; for example, the first child could be given the number 1, the second the number 2, etc.
- *name* is the name of the child element, represented as a string.
- *flag* indicates what type of data the child node holds. For example, the child can itself be an element node or it can contain text data. There is a separate table for each different value of the flag.
- *target* is an integer identifier for the child node. Depending on the value of *flag*, this identifier is used to look up the child data in one of the different value tables. This identifier references the data that appears between the start tag and corresponding end tag of the element whose name is given in *name*.

If the above schema is used, additional lookups are required to retrieve the data stored in the child node. To avoid these lookups, a number of alternative mappings are possible. For example, the *flag* can be omitted and separate *target* columns introduced for each possible SQL data type. *target* columns whose data type does not apply to the current table entry are set to NULL.

3.5.2 Attribute Approach

This variant of the above scheme has nothing to do with XML element attributes, but gets its name from the fact that the edges in the document graph are also referred to as attributes in [FlorKoss99].

One observation with the edge approach is that the *name* strings take up a considerable amount of space in the database. At the same time, for any document the number of distinct element names will be fairly small (compared to the size of the document), so the attribute approach does not use a single large table to hold all the data, but instead creates a new "attribute" table for each element name encountered in the document:

A_{name}(source, ordinal, flag, target)

Thus, compared to the edge approach, the *name* field is no longer present – all the other fields have the same meaning as before.

3.5.3 Universal Table Approaches

It is also possible to store the data in a single, very "wide" table which has columns for each possible kind of child. Apart from the *source* field, there is one set of *ordinal/flag/target* fields for every possible element name. This *universal table* can be generated by performing an outer join on all the attribute tables of the previous section.

Obviously, the universal table typically contains many NULL values. Additionally, it is not normalized, which can cause problems e.g. with inconsistencies introduced by updates. On the other hand, often all children of an element are available with just a single lookup in the table – if the element has several identically named children, multiple entries for the same parent element (i.e. the same *source* value) are necessary.

With the *normalized universal approach*, multiple entries for the same *source* value are eliminated by storing them in separate "overflow" tables. One such overflow table exists for each possible element name. If an element has two or more children with the same name, the *flag* field for the element name of these children in the universal table contains a special value which indicates that the actual values are to be looked up in the overflow table.

3.6 Improving Performance With Index Structures

There are many similarities between indexes for XML databases and indexes for "classical" relational databases: The aim of index structures is to speed up database access times by choosing a special data structure which makes certain operations more efficient. However, at the same time the index structure itself must be updated whenever the indexed data changes, and it occupies storage space, so it should be decided on a case-by-case basis whether to index certain data or not. It is possible to create indexes which are oriented towards the data values stored in a database, allowing e.g. quick access to a customer given his name, or which provide "shortcuts" for the structure of the database, e.g. to speed up the XQuery "child" operator.

When looking at indexing XML data, the special properties of XML (compared to relational databases) come into play: XML documents can have a much more flexible structure and XQuery allows for complicated ways of navigating through this structure, whereas with relational databases the organization of data is restricted to relations, and only the operations of the relational algebra, such as joins, are used on the relations.

XML indexing is an area in which a lot of active research is being conducted. Because XML is so flexible, it is difficult to create an index structure which is easy to maintain, usable for documents that are updated from time to time, and which covers as many aspects of XQuery as possible. This section attempts to give an overview of the latest developments in XML indexing, by describing the ideas behind the index structures in recent papers. Its primary aim is to show the many different ways in which XML data can be represented, and the clever means employed to speed up queries with certain data structures. In order not to confuse with too many details, only the basic ideas are outlined – see the respective papers for more detailed information.

3.6.1 DataGuides

The DataGuide concept introduced in [GoldmWid97] is inspired by the authors' wish to automatically generate a structural description of XML documents (comparable to a schema for a conventional relational DBMS), but it can also be used to build an index for documents which are instances of the structural description. The resulting index can be updated when the document changes. However, when using the data structure this way, it appears to take up a significant amount of storage space.

DataGuides can be used to check whether the result of a XQuery path expression like "$document/node" is non-empty, and more accurately, exactly how many nodes it contains. When using them as the index of a particular document, they can also return the set of nodes for the path expression in question. Furthermore, they can act as an aid when formulating queries, since they give a concise, up-to-date summary of the database structure.

Essentially, a DataGuide is a tree very similar to a document tree, except that a union operation is performed on all identically named siblings anywhere in the tree, so that no node in the DataGuide ends up having two children with the same element name. Additionally, by default any data contained in leaves of the tree is removed; only the document structure is preserved. For example, if part of an XML database of restaurant reads

```
<restaurant>
    <name>Chili's</name>
    <phone>555-1234</phone>
</restaurant>
<restaurant>
    <name>Darbar</name>
    <manager>Smith</manager>
</restaurant>
```

then the resulting part of the DataGuide would only contain the information that <restaurant> elements can contain <name>, <phone> and <manager> children:

```
<restaurant>
    <name/>
    <phone/>
    <manager/>
</restaurant>
```

The problem of creating DataGuides from document instances has been shown to be equivalent to the generation of deterministic finite automata from nondeterministic ones – a well-researched problem which only takes linear time if the starting structure is a tree, but which can get more expensive if it contains cycles. To make generation of DataGuides possible for the latter case, [GoldmWid99] discusses ways to create "approximate DataGuides", which may not always be entirely up-to-date and correct, but which still represent the document structure reasonably well.

All in all, although DataGuides can be used for indexing XML data, they appear to be more useful for other tasks, such as making estimations about the result size of queries or as a help for humans to explore the structure of large documents. [MiloSuciu99] builds on the experiences with DataGuides and introduces a data structure which is better suited for the building of indexes.

3.6.2 Index Fabric

In [Cooper+01], the authors present an index structure called "Index Fabric" which speeds up XQuery path expressions such as "$document/invoice/buyer/name/text() = "ABC Corp"",

and which also works if several queries like this are *and*ed or *or*ed together or contain wildcards. However, the user will have to specify the names of elements involved in *and*ed expressions prior to creating the index. Additionally, the index structure could be adapted to other, specialized XQuery queries if necessary. The index supports updates to the XML data.

In a nutshell, the fabric works by performing a scan of the XML data, creating all possible queries that should be indexed, and storing them. For example, the index generation process stores all the paths from the root node to each leaf node in the document, and possibly also all paths from non-root nodes to leaves below them. Later, all the entries which match such a path can be looked up with a small number of accesses to the data structure (more accurately, one lookup per XQuery location step).

Normally, storing such an immense number of query strings and results would not be possible due to the excessive memory requirements. The fabric deals with this problem by using a special data structure which highly compresses data items entered into it which have identical prefixes: A Patricia trie.

Patricia tries are unbalanced trees which have the property that if data like the query strings above is stored in them, the size of the resulting trie shows linear growth with regard to an increase of the size of the data. Furthermore, lookups of strings as well as "insert" and "delete" operations only need O(l) time, where l is the length of the query [Heun00, 4.8]. The value of l is further reduced with the Index Fabric by not entering element names into the trie as ASCII, but instead by encoding them with an integer identifier.

The original Patricia trie was designed to be an in-memory data structure. Since its size would exceed available memory for many XML documents, the paper also describes how to extend and modify the data structure to be suitable for on-disc storage and updates.

3.6.3 Pre-/Postfix Order Node Numbering

[Grust02] contains the description of a different way to accelerate the execution of certain XQuery operations. It is intended to speed up XPath/XQuery location steps, like child or descendant (abbreviated "/" and "//", respectively). Unfortunately, there does not appear to be a way to modify the index structure when nodes are added to or deleted from the indexed document – the index has to be regenerated from scratch in this case.

The first step towards accelerating location steps is to represent the XML document tree in a different way. Figure 13 shows how this is done: The nodes of the document tree on the left are numbered both in prefix order and postfix order. To achieve this, a depth-first search through the tree is necessary. In order to generate the prefix order numbers, a parent is assigned a number

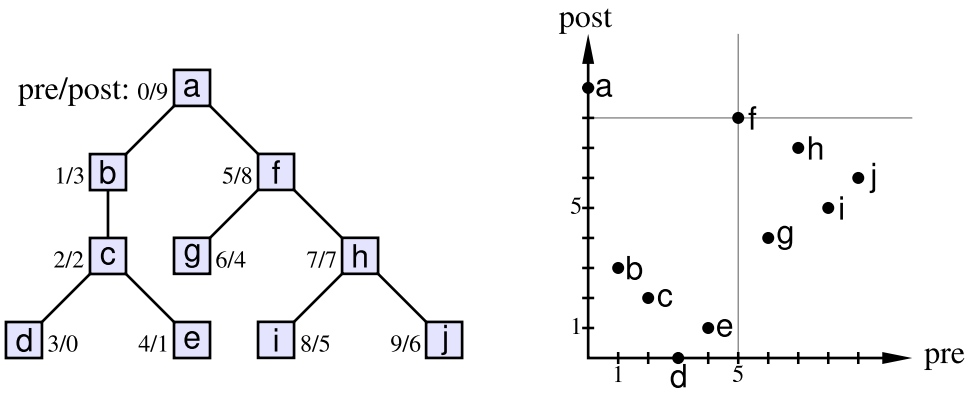

Figure 13: *For the index structure in [Grust02], the nodes of the document tree (left) are numbered both in prefix order and in postfix order. When interpreting these numbers as coordinates (right), each node divides the coordinate space into children, ancestors, preceding and following nodes.*

before processing its children, and to generate the postfix order numbers, it is assigned a number after its children. The resulting tuple of numbers is shown alongside each element's node in the figure.

Next, as shown in the right part of figure 13, the prefix/postfix numbers are interpreted as coordinates in a two-dimensional coordinate system, and the nodes entered as points in the system. Once this is done, the reason for using this mapping becomes apparent: Each point (or node) can be used to subdivide the coordinate space into four regions (illustrated in the example with the node for "<f>"), and the nodes in these regions have the following relation with the node:

- Nodes in the upper left region are ancestors of the node. In the example, "<a>" is the only ancestor of "<f>".
- Nodes in the lower right region are the descendants of the node. For "<f>", these are the nodes "<g>", "<h>", "<i>" and "<j>", as can be verified with a look at the document tree on the left side, where these nodes indeed appear in the subtree rooted at "<f>".
- Nodes in the lower left region precede the node in document order – this excludes any ancestors, in accordance with the XPath standard. For the example, "<f>" is preceded by "<b>" and its children, "<c>" "<d>" and "<e>".
- Finally, nodes in the upper right region follow the node in question. "<f>" has no nodes following after it in the example, so this region remains empty.

To make use of the data structure, the document nodes are stored in this "coordinate form" in a conventional database which can index them using a B-tree or an R-tree. Subsequently, location

steps like “return the children of this node” can be translated in a straightforward way into queries which only return the appropriate set of nodes. Because B-trees and R-trees are well suited for queries which request a range of key values (e.g. “return all nodes whose prefix number is larger than that of <f>”), the resulting lookups are very fast.

The paper goes on to refine the method outlined above by using further properties of the prefix/postfix mapped nodes: With additional knowledge about relations between the numbers, it is possible to narrow the “search area”, and instead of querying for “everything to the right and below <f>”, the search can be restricted to a quadratic region of the prefix/postfix coordinate system, or (under certain conditions) even to triangular or smaller regions.

An index scheme based on the same idea as the above is also proposed in [LiMoon01]. However, due to a different numbering scheme, it shows better performance when modifications are made to the data – the index only needs to be regenerated completely from time to time, not after every modification. Furthermore, the paper describes an alternative solution for the low-level storage of the numbered document nodes.

3.7 Conclusion

In this section, we have examined several different ways of storing XML documents in a database: A document can be stored in a plain-text file (the only standardized format), or it can be transformed into a stream of tokens which is subsequently stored as records, as objects in an object-oriented database, or as BLOBs in a relational database. Additional information about the tree structure of the document can optionally be added, making traversal of the document tree by XQuery implementations possible.

With regard to data structures, a variety of solutions exist for the storage of single tokens, chunks of token data or (sub)trees of the document. Moreover, there are several interesting data structures which allow the creation of indexes over XML data.

When making a decision about which type of storage to use for an XML database, no definitive recommendation for one of the described solutions can be made. Instead, the advantages and disadvantages of one solution over another depend on many factors, and all the following aspects should be taken into account:

- Does the stored data change frequently? If it changes rarely or never, it can make sense simply to store it in flat text form. Furthermore, the use of the more expensive index structures might become worthwhile.

- Will the data be accessed primarily by reading the whole document sequentially, or is quick navigation through the document to certain elements required? In the latter case, some kind of representation of the document tree must be stored in the database to allow XQuery engines to reach the required node quickly. Also, the implementation of an index structure might be advisable.

- What is the preferred output format? If the responses of the database are sent out on a network immediately (e.g. for a web service implementation), it might be advantageous to store the data in plain-text form, instead of being forced to deserialize it again and again for each query. On the other hand, an XQuery implementation will very probably work with a token stream or document tree model, so plain-text storage will slow it down.

- What is the average size of the stored documents? If the documents get very large, it is a good idea to use a storage format which stores chunks of token data, rather than adding bookkeeping information to each token.

- How much time is available for the implementation of the XML storage solution? The implementation of a system which is as reliable and feature-rich as existing relational databases is a very large task. Consequently, it will make sense in many cases to use an RDBMS as the underlying storage system and to implement XML functionality on top of it.

- Are other features required, e.g. concurrent access, recovery or distributed operation? These are also available in RDBMS and would take long to implement from scratch.

- What additional demands are imposed by the environment the database will be deployed in? For example, a customer may require that the data be stored in his relational database system, or he might be interested in the fastest possible execution times for certain XQuery queries, which might require adapting an index structure to work with these queries.

4 Requirements

An important part of this diploma thesis concerns itself with the implementation of an XML storage system. As part of the XL project of the chair for database systems and knowledge bases, this XML storage manager was added to the implementation of the XL programming language, which has already been introduced in section 2.3. By storing XL variables containing XML documents or simple data types on disc, the code adds support for *persistent global variables* and *persistent conversation variables*. This means that if a variable was declared appropriately by the XL program, its value is not lost when the XL program terminates. Instead, upon restarting the program, the old value is still present.

Various components are required for this functionality. Below, they are described together with the set of features they provide.

4.1 Efficient Modification of XML Documents

In the current XL implementation (which is written in the Java programming language), the values of XL variables, the content of any messages received over the network and also the results when evaluating expressions are stored as XML data in objects which are instances of the XL_Value class.

Previously, XL_Value was implemented as a simple array of token objects, and offered an interface which allowed the construction of a new value from an array and the creation of a token iterator which could be used for a sequential read of the data.

Since XL_Value is the central class concerned with the representation of simple values as well as XML documents, it must be re-implemented with the following requirements in mind:

- The API of the class is extended to allow the insertion of new XML data into the document tree, and the deletion of nodes from it. Furthermore, the code which implements the XL document modification statements (e.g. **insert** and **delete**) is modified correspondingly to make use of this new API. (The old implementation performed insertions and deletions simply by recreating the whole value with the changes in effect.)

- It is possible to create a token iterator for the XL_Value which retrieves its tokens in sequential order.

- It should also be possible to move token iterators forward through the document in greater steps: From a “begin element” or “begin attribute” token, the token iterator should be able to jump directly to the corresponding “end element/attribute” token.
  Support in the form of an API is not required at this stage because the XQRL XQuery

implementation which would make use of the API has not yet been modified to do so. However, the underlying data structures of the storage system should support navigation from the "begin" to the "end" in order to allow the implementation to be adapted easily later.

- It is easy to perform an "is in document order" comparison for "begin" tokens returned by an iterator, i.e. to determine which one of any two tokens appears earlier in the document.

In general, the idea behind these modifications is to make it possible to store large documents in XL variables by making operations more efficient, while keeping a large portion of the old XL_Value interface unchanged.

4.2 Persistent XML Storage

The modifications to XL_Value from above already constitute an improvement because they make insertions and deletions faster. But the main motivation for these changes is to make it possible to add support for persistent storage to the class.

XL variables are implicitly made persistent if they are declared in the global scope, either as global variables ("**let** $x;") or conversation variables ("**context let** $x;"). As implied by the absence of special "persistence declarations" or similar in the XL specification [XL02], the persistence support is transparent to the user; no special action is required to make variables persistent as long as they are defined in the scope of the **service** declaration.

If a variable is persistent, the XL storage system handles it in a special way, ensuring that any changes made to the XL_Value which holds its data are written to disc, ready to be accessed again if the XL program is restarted. The difference between global variables and conversation variables is that several instances of an identically named conversation variable can exist (one instance for each conversation), whereas there is only one instance of each global variable.

Large, persistent variables will often contain an XML document with a database maintained by a web service, and the data in that database can be quite valuable. Therefore, the XL storage support also makes every attempt to store the data as securely as possible, and it provides facilities expected of a "normal database", e.g. backups and recovery from catastrophic failures.

Furthermore, the storage system ensures that concurrent accesses to the same XL_Value by different parts of XL always result in a valid view of the value – in other words, modifications like insertions and deletions are atomic. This is important in the face of recent additions to the XL implementation, which attempt to increase performance by running several Java threads which work in parallel.

Persistent variables are not the only ones that may be written to disc: If a non-persistent variable gets too large to be held in memory, the storage manager may also decide to write parts of it or the whole value to disc. The disc space occupied by such "swapped out" variables is reclaimed as soon as the control flow leaves the scope that variable was defined in.

Finally, in order to ensure that the amount of token data held in memory never exceeds the available memory size, a buffering scheme is implemented. Its aims are to speed up the retrieval of frequently accessed values or tokens, write less frequently used values or tokens to disc if memory is tight, and to make accesses to small, temporary variables (which are very common in XL) as fast as possible.

4.3 Modular XML Storage System

Another property of the new XL storage system is modularity in the sense that it is possible to delegate the task of storing values to one of a number of different "storage modules". Each one of these modules provides the same functionality to XL, but implements it in a different way. For example, there could be one module which stores the data as flat text files, and another one which stores it in a relational database system.

As part of the practical part of the thesis, one storage module which allows on-disc storage of XML documents is written. Additionally, another "dummy" module provides the behaviour that previous versions of XL exhibited: It only stores the values in memory and does *not* write them to disc.

The user can select a storage module using a switch on the command line when he starts a Java virtual machine to execute an XL program. Each module can advertise special options supported by it, which are used for things such as specifying where on disc to store the persistent XL_Values.

Each storage module provides the same interface to the higher-level layers of the implementation. The token stream stored by it is suitable for processing by the XQuery implementation of XQRL, Inc.

The persistent storage support in XL is expected to be used in environments where loss of data must be avoided. Since XL applications will typically use one or more persistent variables as their "database", it is important that the implemented storage solution be as robust as possible. In particular, it should include the possibility of making backups and of recovery from crashes.

5 Design

5.1 Components

The overall architecture of the new XL storage subsystem is shown in figure 14. At the top of the figure, the XL runtime system represents the parts of the XL implementation which make use of the functionality provided by XL_Value. A lot of knowledge about the web service, conversations and whether variables are persistent is only known to the runtime system, and is made available to XL_Value when values are created. Furthermore, additional configuration information for XL_Value such as which storage module to use, is passed through by the runtime system to XL_Value.

Depending on the user's choice, XL_Value enables one of a number of available storage "back-ends" (figure 14 shows three examples for such modules) to enable persistent storage for values. The respective module is then completely in charge of all stored values (regardless of whether they are persistent or not) and responsible that the access operations passed on to it by XL_Value are executed as appropriate. It is also in charge of any buffering of the token data.

The following subsections describe the three major components of the XL storage system in detail: Persistent storage of values, buffering and the mechanism for exchangeable storage modules.

5.1.1 Persistent Storage

The persistent storage of XL variables has the following semantics:

- Each global variable in an XL program is uniquely identified by its name and the web service it is associated with. The web service is in turn identified by the URL given in its **service** declaration (see page 26).
- Each conversation variable in an XL program is uniquely identified by its name, the web service and the "context" it appears in. More accurately, each context is associated with its own conversation, and the conversation can be identified via its URI.
- Apart from the above identification, the variable values are not tied to a particular XL program source code file or similar, so it is possible (but generally not to be recommended!) to write two different programs which access the same global variables by using identical names and identical service URIs.
- If the declaration of a global or conversation variable does not include an initializer ("(**context**) **let** $x;", see [XL02, 4.2]), then the persistent variable is initialized with an

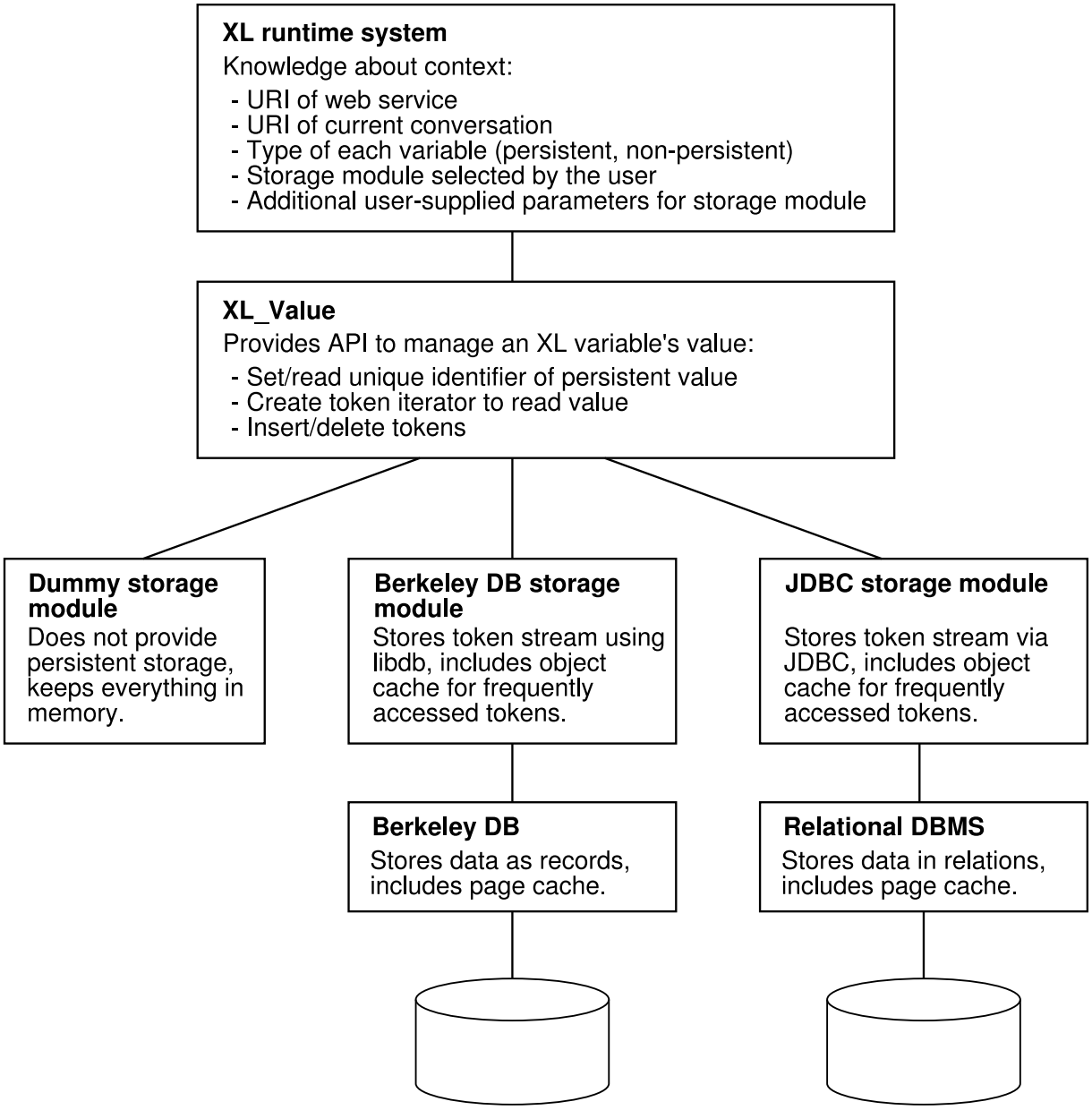

Figure 14: *Architecture overview for the XL storage subsystem, with three example storage modules. The XL_Value class is central to the design, providing a uniform API to upper layers and managing beneath it the modules which store the XML data.*

empty value the first time the XL program executes. On subsequent runs of the program, the initialization does not take place, and the value of the variable is restored to whatever it was before the previous program run terminated.

- Similarly, if an initializer is present in the variable declaration ("(**context**) **let** $x := <recipes/>;"), then the initialization with the supplied expression ("<recipes/>") only happens once, when the XL program is run for the first time, and for subsequent runs the value left behind by the previous run is restored. Furthermore, the expression of the initializer is not even evaluated in this case.

Internally to the XL implementation, persistent values are not distinguishable from non-persistent ones except by the fact that the "unique identifier" field of the former ones is non-empty, in contrast to the latter. Apart from this, the persistence layer is transparent; all operations have exactly the same result on persistent and non-persistent values.

New persistent values are created on demand by XL_Value whenever the runtime system requests a value whose unique identifier is as yet unknown. The default value of any such new values is empty.

Deletion of persistent values can be requested simply by clearing the value. Again, from the point of view of the XL runtime system there is no difference between the effect of the clear operation between persistent and non-persistent values. (Obviously there *is* a difference if *no* clear operation takes place before the XL runtime system terminates: The non-persistent value is lost, the persistent one preserved.)

It is up to the individual modules to decide about the exact way in which data is stored persistently. Section 3 contains descriptions of a variety of different ways to do this.

Although XL does not yet have any concept of transactions, a storage module may internally want to use transactions to allow easy recovery from system crashes, and to allow rollbacks in case of deadlocks.

5.1.2 Buffering

As mentioned before, keeping tokens cached in memory serves the double purpose of speeding up accesses to frequently used portions of large persistent values and holding small, temporary values in main memory during their entire lifetime.

How exactly the buffering of tokens is performed is left for the individual storage modules to decide; different modules may employ different strategies in this area. The following aspects of buffering need to be addressed:

- What is the granularity of the cache? Possible strategies include caching the individual Java objects which represent tokens, caching entire values (i.e. either all tokens of a value are on disc or all are in memory), caching chunks of the token stream in serialized form, or caching physical pages.

- Should caching be performed at multiple levels? Because the conversion of token objects into a stream of bytes and vice versa is expensive, it can make sense to buffer both at a higher level (Java objects) and a lower one (e.g. pages).

- What replacement strategy is implemented? Options range from LRU (*least recently used*) to LRU-2 (LRU, but counting the last two accesses) to special-purpose solutions, for example with heuristics for recognizing sequential reads of large amounts of data, so as to be able to treat them specially and not having them "push" other data out of the cache.

Another, related issue which the buffering of each storage module should solve is the "swapping out" of non-persistent values to disc if they become too large or are used only very rarely. Since both persistent and non-persistent values are handed over to the storage module in the same way, the module must maintain information not only about the amount of memory taken up by persistent values, but also by non-persistent values. When a non-persistent value becomes eligible for swap-out according to the replacement policy and is written to disc in whole or part, the module must also take care to clean up later when the value is discarded, and delete any on-disc data related to it.

The amount of memory to use for caching is specified by the user with a command line switch called "`-cachesize`". All persistent storage modules should honour the cache size setting, although the exact interpretation of the number passed to the command line switch can differ from module to module.

It should be noted that the design above is a revised version of the original approach, which included a single cache mechanism that all storage modules were intended to build upon. Section 6.3 briefly introduces the other approach and explain why it was abandoned.

5.1.3 Management of Storage Modules

The user is responsible for selecting a persistent storage module if he wants the global and conversation variables of his XL program to be preserved. This is possible using a "`-storage`" command line switch, followed by the name of the module to use and any additional parameters for that module. If "`-storage`" is not specified, the default is to use the dummy storage module which does not actually support persistent storage, but keeps all values in memory.

If an unknown name is supplied to "`-storage`", the XL runtime system prints out an error message which includes a list of module names and supported per-module options.

XL_Value allows switching from one storage back-end to another at any time – however, in this case, only newly allocated values are actually kept track of by the new storage module, older ones continue to be managed by the previously used storage module. Additionally, each one of the storage modules assumes the entire amount of memory specified with "`-cachesize`" is at its disposal, so the XL runtime system should only switch to a new storage module after discarding all the values maintained by the old one, to avoid that the different modules take up too much memory.

The functionality provided by the individual storage modules is identical to the functionality of the XL_Value class described in section 4.1: Essentially, they must support creation of new values, insert and delete operations, a way of clearing values, and they must provide their own iterator implementation for reading the value. When a module has been registered, XL_Value forwards requests for the creation of new values to that module, and once a value has been created, XL_Value forwards all operations on that value to its creator.

5.2 Programming Interface

In this section, the functionality which was described above is represented in a more concrete form as programming interfaces. Since the XL project uses the Java programming language, the interface definitions are given in Java syntax. However, the design could conceivably also be adapted to other programming languages. Most of the descriptions of the individual classes and functions are also available in the form of Javadoc comments in the source code files.

The different interface definitions are provided by the following classes and Java interfaces:

XL_Value This class provides "values" of tokens. Each instance corresponds to a stream of zero or more tokens. In many cases, a value only consists of a single token like "5", but the value can also be a complete XML document.
XL_Value also offers functionality for storage modules to register themselves and to select a storage module from the list.

StorageManager This interface is the main mechanism used for the creation of new values by the selected storage module. Each storage module provides an implementation for StorageManager. The implementation classes are singleton classes (i.e. only one instance is ever created). XL_Value uses the instances of different StorageManager implementers as object factories for StoredValue instances.

StoredValue This abstract class is also extended by each storage module. The module's StoredValue objects hold the actual token data. There is a one-to-one mapping between XL_Values and StoredValues.

XL_Value.Iterator This interface is implemented by each storage module. To read the value's content, the creation of iterator objects can be requested from an XL_Value.

Since the interface definitions below are not grouped by class, but by the different types of functionality present in the storage system, the method names are preceded by the class name, i.e. "XL_Value.size()" instead of just "size()".

5.2.1 Storage Back-end Selection

This section describes how storage modules can register themselves with XL_Value and how the selection of a particular module works, including passing parameters to the module.

XL_Value provides ways to register storage modules, to inspect the list of registered modules, and to set/read the module to be used for values:

public static void XL_Value.registerStorageManager(String name, StorageManager mgr);

Add a new StorageManager to set of known managers. A storage module provides a way of storing XL_Values persistently. For each StorageManager implementer, this method must be called to make the manager known to XL_Value.

Parameters:

name String like "`libdb`" identifying the storage manager. Used for example to select the type of persistent storage with the `-storage` command line switch.

mgr Object that XL_Value should use to process command line options for this StorageManager module and to create new StoredValues.

public static Set XL_Value.registeredStorageManagers();

Return read-only information about registered storage managers. Each element in the returned collection is a Map.Entry. The objects returned by Entry.getKey() are of type String, those returned by Entry.getValue() are of type StorageManager.

public static StorageManager XL_Value.findStorageManager(String name);

Find a storage manager by name, and return the corresponding object.

public static void setStorageManager(StorageManager manager);

Set the storage manager to use for persistent values. If persistent storage is required, this should be called as early as possible; until then, all XL_Values are buffered in RAM.

public static StorageManager XL_Value.getStorageManager();

Return the selected storage manager, as set with setStorageManager(), or null if it has not been set yet.

The StorageManager interface contains all the functionality required to set up a storage module and to have it generate StoredValues which hold the data of XL variables. Additionally, it allows the storage module to be notified when the XL runtime system shuts down:

public void StorageManager.printOptionInfo(PrintStream out);

Print out information about the command line options recognized by this storage module, writing it to the supplied PrintStream. This is used by the XL runtime system to print information about the options supported by all the modules.

public void StorageManager.parseOptions(String options, PrintStream err);

Set options recognized by this storage manager. The options are passed as a single string, the intention being that multiple settings are encoded in the string, e.g. separated by commas as "`file=foo,size=42,flag`". Error messages about invalid options can be printed to the supplied PrintStream.

public StoredValue StorageManager.createStoredValue(XL_Value owner, String name);

Create a new StoredValue. Called by XL_Value whenever storage for a new XL variable is needed. If name is empty, the value is anonymous and non-persistent. If name is non-empty, the old persistent value is reloaded from disc, or (if none of that name is present on disc) a new persistent, empty value of that name is created.

Parameters:

owner The XL_Value that the new value is to be associated with. This association remains unchanged until the StoredValue is discarded.

name The empty string if non-persistent, else a unique identifier of the value.

public void StorageManager.flush();

Is called when an XL operation has finished, and when the XL runtime system shuts down; if the storage module wants to, it can use this to sync its buffers to disc.

public void StorageManager.shutdown();

Is called when the runtime system closes down, providing an opportunity for the StorageManager to flush out its data and close the database.

5.2.2 Document Iterators

The iterator concept of XL_Value is mostly based on the iterators of the XQRL XQuery implementation. The corresponding class, XL_Value.Iterator, extends XQRL's com.xqrl.iterators.TokenIterator, inheriting the following methods:

public void XL_Value.Iterator.open() throws XQRLException;

Open the iterator, to prepare for reading data. Must be called before the first call to next().

public boolean XL_Value.Iterator.isOpen();

Check whether the iterator is open or not.

public void XL_Value.Iterator.close();

Close the iterator, freeing any resources maintained by it. Must only be called if the iterator is open.

public boolean XL_Value.Iterator.hasNext();

Check whether the end of the token stream is reached.

public Token XL_Value.Iterator.next() **throws** NoSuchElementException**;**

Read and return the next token, if any. The iterator must have been opened before this method can be called.

public Token XL_Value.Iterator.peekNext();

As above, but do not advance the iterator after reading the next token. In other words, repeated reads to peekNext() keep returning the same token.

public XQueryType XL_Value.Iterator.getXQueryType();

Return the type of data of the tokens returned by calls to this iterator's next() method.

Additionally, XL_Value.Iterator also offers these two methods:

public XL_Value XL_Value.Iterator.getValue();

For an iterator, return the XL_Value object whose data the iterator reads.

public Object XL_Value.Iterator.clone();

Clone an iterator. This can be useful to memorize iterator positions.

5.2.3 Token Identifiers

Token identifiers are another concept which is introduced by the XQRL XQuery implementation and used in XL. In the token stream, a token identifier is associated with every "begin" token. The token deletion and insertion methods introduced below use token identifiers to specify the position of modification, so all storage modules will have to replace the original XQRL identifiers with a version that continues to behave like the original XQRL identifier class, com.xqrl.tokens.Identifier, but which is extended to contain an internal reference to the modules' low-level data structures. Only with such a direct reference, the insertion/deletion position can be found quickly.

The Identifier interface is fairly simple:

void Identifier.markEnd() **throws** XQRLSystemException;

This method is called by the XQRL code when it encounters the "end" token which matches the "begin" token of the Identifier. (This is useful for the XQRL implementation of Identifier.)

Identifier Identifier.getParent() **throws** XQRLSystemException;

Return an Identifier for the parent of the "begin" token which this Identifier belongs to. An implementation may support this call, but it is not required to.

Identifier Identifier.createNewId() **throws** XQRLSystemException;

Generate a new Identifier on the basis of an existing Identifier.

URIToken Identifier.getDocumentURI();

Return the URI of the document to which the token of this Identifier belongs.

public int Identifier.compareTo(Object obj);

Perform a comparison between two Identifiers, returning a result which is less than, equal to or greater than zero depending on whether this appears earlier in the document than obj, is identical to it or appears later.

5.2.4 Document Creation, Reads and Modification

For convenience, many of the XL_Value methods for the creation and modification of values are overloaded, to allow their being used with different types of input. Apart from reading the token data via an iterator, it is also possible to inquire about its size.

public XL_Value.XL_Value();

Create a new empty value to hold a stream of tokens. By default, the value is anonymous

and “non-persistent”, i.e. although it may be swapped to disc under memory pressure, it is always deleted when the storage subsystem is shut down.
The type of the value is set to XQueryType.ANYTYPE by default.
Any allocation of an XL_Value object will also result in the allocation of a corresponding StoredValue object.

public XL_Value.XL_Value(XQueryType type);
Create a new value of the given type.

public XL_Value.XL_Value(TokenIterator iter);
Create a value and fill it with tokens from the supplied iterator.

public XL_Value.XL_Value(TokenIterator iter, XQueryType type);
As above, but also set the value’s type to the given XQueryType.

public XL_Value.XL_Value(Token[] tokens);
Create a value and fill it with tokens from the array.

public XL_Value.XL_Value(Token[] tokens, XQueryType type);
As above, but also set the value’s type to the given XQueryType.

public XL_Value.XL_Value(Token t);
Create a value containing just a single token. The token must not be a “begin” or “end” token.

public XL_Value.XL_Value(Token t, XQueryType type);
As above, but also set the value’s type to the given XQueryType.

public synchronized XQueryType XL_Value.getType();
Return the type of an existing value.

public synchronized void XL_Value.setType(XQueryType t);
Overwrite the type information in the value with the given XQueryType.

public synchronized Iterator XL_Value.getIterator();
Return an iterator for the value, which reads it starting with the first token.

public synchronized Iterator XL_Value.insert(Identifier destId, **int** beforeAfter, TokenIterator srcIter) **throws** XQRLException;
Insert tokens into the stored value. If the inserted data contains a “begin” token, it must also contain the corresponding “end” token, i.e. it is not possible to insert an unbalanced

structure. The method returns an iterator which points into the value, after the inserted data.

Parameters:

destId Identifier of a token somewhere inside this. Must be a "begin" token. Special case: Pass null to insert at the very beginning (if beforeAfter is BEFORE or INTO_FIRST) or end (if beforeAfter is AFTER or INTO_LAST) of the document – also pass null if the document is empty.

beforeAfter The different values which can be used for this parameter are defined as constants in the XL_Value class: BEFORE is used to insert before the token at destIter, AFTER to insert after the "end" token which corresponds to the "begin" token at destIter, INTO_FIRST/INTO_LAST to insert as the first/last sub-element, inbetween the "begin" and "end" tokens.

srcIter Iterator of the token stream to be inserted. If this iterator is not open, it is automatically opened before data is read from it, and closed before insert() returns.

public synchronized Iterator XL_Value.insert(Identifier destId, **int** beforeAfter, Token[] tokens);

As above, but do not insert token data from an iterator, but instead from an array of tokens.

public synchronized void XL_Value.delete(Identifier tokenId);

In the stored value, delete the designated node, i.e. from its "begin" token up to the "end" token. The tokenId parameter must point to a "begin" token somewhere inside this.

public synchronized void XL_Value.clear();

Delete the complete contents of the value.

public synchronized int XL_Value.size();

Returns the number of tokens stored in the value, or -1 if the number is not known.

public synchronized boolean XL_Value.isEmpty();

Returns true if the value is empty, i.e. contains zero tokens.

The following methods can be used to convert the XL_Value into other data types:

public String XL_Value.toString();

Convert the value to a string. If the value is a document, this includes converting the token data to standard XML tags.

public synchronized boolean XL_Value.toBool();

Convert the value to a boolean value using com.xqrl.runtime.logic.BoolEffValue.

public synchronized double XL_Value.toDouble();

Convert the value to a double value using com.xqrl.runtime.numeric.Any2Double.

As mentioned, the value's data is really stored in objects whose classes extend StoredValue. Many of the XL_Value methods above are mapped to the corresponding methods below. Since the functionality of most of these should be obvious from the name and signature, only short descriptions of the StoredValue methods are included:

protected StoredValue.StoredValue(XL_Value value, String name);

Create a new value. If name is empty, child classes return a new empty, non-persistent value. If name is non-empty and no value has yet been stored on disc with the given name, child classes must create a new, empty, persistent value associated with that name. Otherwise, they reload the data from the existent persistent value.
value is the XL_Value which is going to "own" this object.

public XL_Value StoredValue.getValue();

Returns XL_Value which owns this object, as set by the constructor. The default implementation simply returns the XL_Value reference from a private data member; it should not require overriding.

public XQueryType StoredValue.getType();

Returns the XQRL query type for the stored token stream, or null if the type is not set. The default implementations of this method and setType() below provides access to a private data member and should not require overriding.

public void StoredValue.setType(XQueryType t);

Set the XQRL query type of the stored token stream.

public abstract XL_Value.Iterator StoredValue.getIterator();

Create and return an iterator for the stored value.

public abstract XL_Value.Iterator StoredValue.insert(Identifier destId, **int** beforeAfter, TokenIterator srcIter) **throws** XQRLException;

Insert tokens from srcIter into the value at the position indicated by destId.

public abstract void StoredValue.delete(Identifier tokenId);

Delete the designated node from the value.

public abstract void StoredValue.clear();

Delete the complete contents of the value.

public abstract int StoredValue.size();

Return the size of this value.

5.2.5 Making Values Persistent

Sections 5.2.1 and 5.2.4 above have already mentioned that StorageManager.createStoredValue() is passed a name parameter (which it typically uses in a call to the StoredValue constructor) which is empty for non-persistent values and a non-empty, unique string for persistent ones. XL_Value creates non-persistent values by default. The following two methods of the XL_Value class allow values to be declared persistent, and to read their unique identifier, if any.

public synchronized void XL_Value.setName(String name);

By using this method to set the name of the value to a non-empty string, the value is declared persistent.

To reload a value that was previously declared persistent, create an XL_Value in the new runtime environment and set its name to the appropriate string, then the value's previous contents reappear.

If the old name compares equal to the new name, everything remains unchanged.

If the old name was non-empty, but the new name is empty, then the value of "this" is set to the empty value. However, the value available through the previous name remains unchanged and is not discarded.

If the old name was empty and the new name is non-empty, then the old value of "this" is discarded and replaced by any value previously stored under the new name. (If nothing was previously stored under the new name, a new, empty value is created.) All modifications made to the value now (until the next setName()) are persistent.

public synchronized String XL_Value.getName();

Return the value's name, as set by setName(), or the empty string for XL_Values whose setName() has not been called so far.

It should be noted that StoredValue does *not* provide any methods equivalent to the above: StoredValues are either created as persistent values or as non-persistent values, that state never changes during the lifetime of the object. This implies that XL_Value will create a new StoredValue during calls to setName(), by calling StorageManager.createStoredValue() for the currently selected storage module.

5.2.6 Buffering

Maintaining a buffer of recently accessed tokens, token chunks or pages as well as providing a replacement strategy for this buffer is mostly left to the individual storage module implementations. The only part of the XL_Value interface which is concerned with buffering are the following two functions. They provide a uniform way for the user to set the cache size, and for storage modules to read it:

public static synchronized void XL_Value.setCapacity(**int** size);

Set the cache capacity. The exact interpretation of the size parameter may differ from storage module to storage module – one possibility is to measure the cache size as the number of tokens in the cache.

public static synchronized int XL_Value.getCapacity();

Return the value passed to setCapacity(), or a default value if that method has not yet been called.

In addition to the cache size, some storage modules may require further cache-related options, for example to tune the replacement strategy. These can be passed to the module with the other module options – see the description of StorageManager.parseOptions() above.

5.2.7 Test Framework

For the most part, the different components which are used for testing do not require special interfaces – they are described in section 7. However, the following functions, present in the XL_Value and StoredValue classes, allow all storage modules to offer an internal integrity check of their data structures to XL_Value in the same way:

public synchronized void XL_Value.assertValid();

Check the integrity of the data structures for this XL_Value. Calls to this function are ignored if assertions are turned off for the Java virtual machine. A call to this method results in a call to isValid() of the StoredValue that this XL_Value corresponds to.

public boolean StoredValue.isValid();

Check whether the data structures for this value are correct. It is acceptable if this is a expensive operation for some storage modules, e.g. because it requires the traversal of large cache structures or similar. The method always either returns true or fails with an AssertionException.

If this function is not overwritten by a class deriving from StoredValue, the default implementation just returns true.

6 Implementation

The design of the functionality outlined in the previous section, as well as the implementation of the new XL_Value class and two storage modules required the greater part of the time available for this diploma thesis.

The first (and by no means insignificant) part of the implementation work was getting to know the internals of the current XL code, and also the major components of the XQRL XQuery code. After the parts of XL that required changes had been identified, the design of the new interfaces began – the resulting work on XL_Value is described in section 6.1. Previously, almost the entire management of values had been handled by a single class de.TUM.RTS.XL_Value – due to a cleaner separation of the code for value creation, access and modification, the new XL_Value class is now only one of many classes in the de.TUM.RTS.Value package.

The next major part of the persistent storage implementation involved writing the two storage modules mentioned in section 4.3: First, a simple version which keeps everything in memory allowed the new interface to be tested and the rest of XL to be adapted to it. Next, after an evaluation of the different possibilities for storing XML data on disc, a second module with persistence support was added. The modules are described in sections 6.1.2 and 6.2.4.

Unfortunately, the work related to adding a buffer of values to the second storage module necessitated a partial redesign of the buffer-related parts of the XL_Value class. Section 6.3 details the original approach as well as the reworked approach.

The XQRL Token Stream Format Before starting to describe the changes made to the XL code, we will first have a short look at the format of the token streams that the XQRL implementation expects.

As mentioned before, an XQuery implementation was not written from scratch for the XL programming language. Instead, XL uses the implementation by XQRL, Inc. The XQRL code is under constant development, with the result that the token format changes from time to time. However, the format largely follows the ideas laid out in section 3.3.2:

- Elements consist of a “begin element” token, a QName token with the name of the element, followed optionally by a number of attributes, then by the element’s content (represented by zero or more tokens), and finally by a “end element” tag.
- Attributes are not stored inside the “begin token” they belong to, but as separate tokens. They consist of a “begin attribute” token, a QName token for the attribute name, a single token containing the attribute value, and an “end attribute” token.

- Furthermore, special tokens exist for comments, namespace declarations, processing instructions, and for all the primitive data types defined in the XQuery standard.

At the time of writing, the XQRL token stream format was still in a state of flux with regard to how the type information associated with elements and attributes is stored in the stream. One possible solution for this is to store an additional token with type information after the "begin element" or "begin attribute", another to include the type information in the "begin" token.

6.1 XL_Value

The new version of XL_Value uses data structures which allow efficient insertion and deletion of tokens in the middle of the value. Apart from the corresponding insert()/delete() etc. methods, it also adds support for persistent values.

6.1.1 XL_Value Class Internals

The XL_Value class implementation is fairly straightforward, since most of its functionality is concerned with forwarding method invocations to the StoredValue instance whose reference is stored in the private store data member.

When an XL_Value is created, a call to the createStoredValue() method of the currently registered StorageManager is used to obtain a new object for the store data member. If a later call to setName() changes the object's unique variable identifier, store is replaced with a reference to a newly created StoredValue.

XL_Value contains static initialization code which automatically registers all StorageManagers in the de.TUM.RTS.Value package when the class is loaded by the Java virtual machine. Additionally, the static initialization code makes sure that shutdown() will be called before the JVM terminates, by registering with java.lang.Runtime.addShutdownHook() a thread which calls shutdown().

6.1.2 Module For Non-Persistent Storage: BufferedValue

BufferedValue is the first storage module which was written for the new XL_Value implementation. There are several reasons for creating a storage module which does not actually implement persistent storage:

- In many cases, it is possible that XL applications do not need persistent storage. Examples for such applications include cases where the amount of data stored never becomes very large (i.e. can be held in memory) and can be recreated when the web service is restarted. For these types of applications, it is easier to run XL without persistence support because no

database or other storage back-end has to be configured. Also, as we will see in section 8, BufferedValue performs better than other storage modules.

- The presence of BufferedValue quickly made XL usable again: Since several people work on XL in parallel, any changes committed to the code base must not completely break the system. Consequently, the new XL_Value code could not be committed as long as it was not a full replacement for the old code.
  The alternative way to proceed would have been to wait with committing the changes until a module for persistent storage was also implemented. However, this would have meant not to commit the new code for a long time, which is not desirable because it results in a lot of additional work related to keeping one's changes consistent with changes made by others.

- Writing the simplest module imaginable made it possible to test the whole new XL_Value component early, when the amount of new code was still relatively small (at least compared to the final code size). This meant that errors could be identified more easily.

StorageManager Implementation The BufferedValue implementation classes are located in the de.TUM.RTS.Value package. Like every storage module, BufferedValue provides an implementation of the StorageManager interface, which is contained in the file `BufferedStorageManager.java` and is very simple: BufferedValue does not offer any additional options or support the parsing of such options, nor does it have to take any special action (like flushing out changes to a database) when the runtime system closes down, so the bodies of most of the implemented methods remain empty. Only createStoredValue() has a non-empty body. In order to allow the BufferedStorageManager to be used as an object factory, it creates and returns a new BufferedValue.

All three of the implementations for StoredValue, XL_Value.Iterator and the XQRL Identifier are located in the same source file, `BufferedValue.java`.

Data Structure For Token Storage The data structure which is used for BufferedValue has already been introduced in section 3.4 – figure 15 is identical to the last tree variant in that section, "token stream with additional references". The figure shows the resulting structure for the example "recipe" document. With BufferedValue, every token contains a reference to its parent – more accurately, to the "begin" token of the element or attribute which directly encloses the current token. The parent references are not shown in the figure to prevent it from getting too confusing.

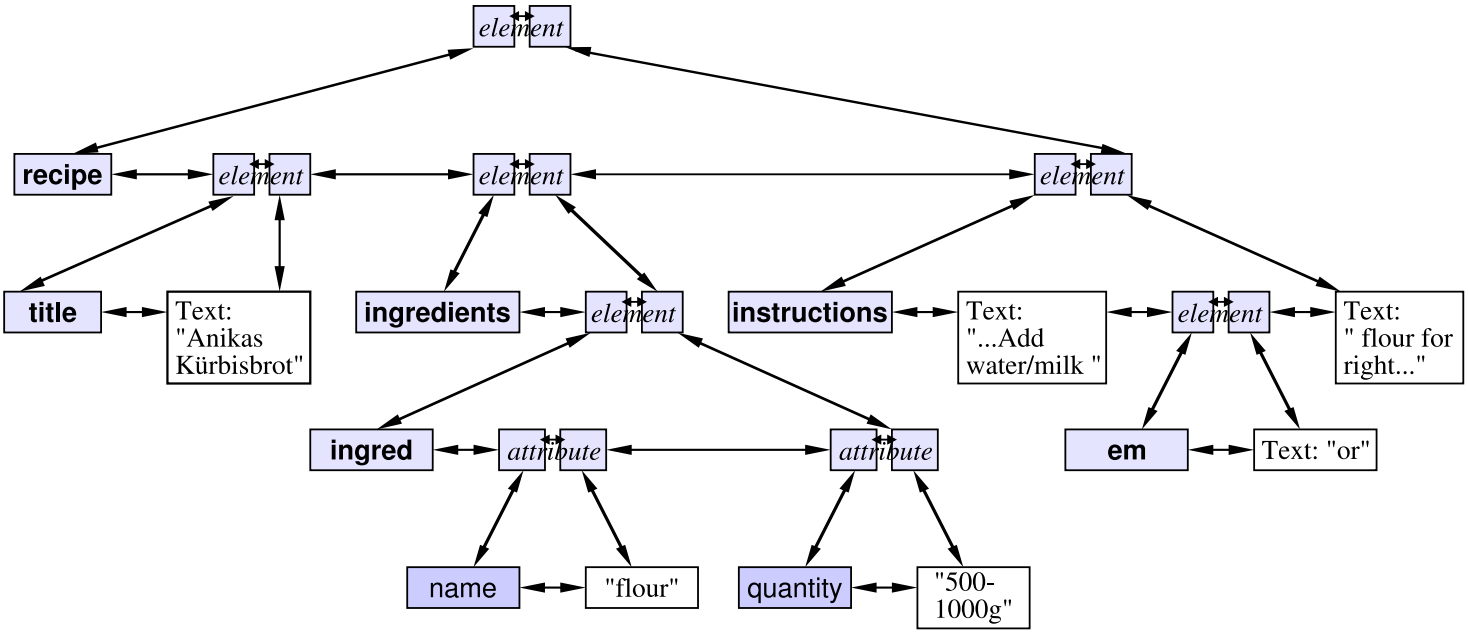

Figure 15: *Data structure for a value stored using* BufferedValue*. The token stream is stored as a doubly linked list with extra references from each "begin" token to the corresponding "end" token. The additional references from each token to its parent "begin" token are not shown in this figure.*

For the low-level representation of the data structure, BufferedValue does not rely on standard Java data types like List because it also needs to store the pointers to parent tokens and "begin" or "end" tokens. Instead, it uses its own doubly linked list implementation whose list objects also have room for these additional references.

Figure 16 shows the way the doubly linked list is implemented: In reality, it is a "doubly linked ring" of list entries of the private nested class BufferedValue.MyLinkedList, each of which contains "previous", "next", "parent" and "twin" pointers as well as a "data" field which references the actual stored token. For lack of a better term, "twin" is used to refer to the "begin" token which corresponds to an "end" token, and vice versa.

This type of doubly linked list is much better than an implementation where the first entry's "previous" as well as the last entry's "next" field are null, because none of the code for modifying the list has to take the special cases "at beginning/end of list" into account.

Token Iterators Just like the standard Java Iterator semantics, the XQRL iterator concept requires that it is possible to position an iterator before the first token or after the last token, i.e. there are $n+1$ iterator positions for a list of n entries. In order for the doubly linked list to allow this distinction, it is necessary to insert an additional "head of list" entry into each list. This also

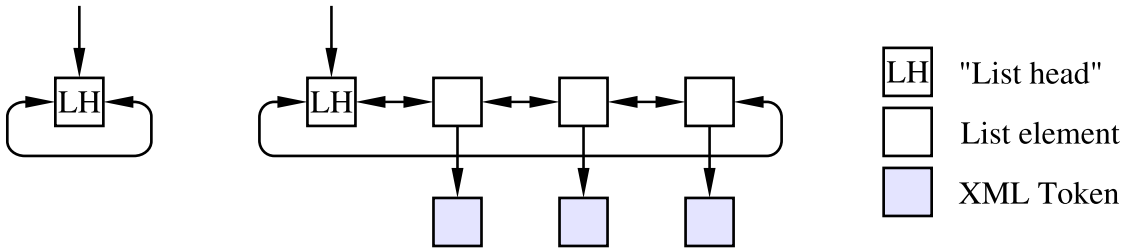

Figure 16: *Two examples of the doubly linked list used internally by* BufferedValue*: Empty list (left), list with three tokens (right). The list is really a "doubly linked ring" of values, with a "list head" object which marks the start and end.*

has the advantage that no special case code needs to be present to deal with empty lists, since an empty list is represented by a list head whose "previous" and "next" fields reference itself.

BufferedValue includes its own implementation of XL_Value.Iterator, in the form of the private nested class BufferedValue.MyIterator. An iterator of this class contains a reference to the XL_Value it traverses, an integer member for numbering "begin" tokens (explained below) and a reference to the current MyLinkedList which indicates the position of the iterator in the token stream. By convention, this reference points to the object before which the iterator is positioned. For example, if the iterator points before the first element, its "current position" field references the first element (reachable via the list head's "next" field), and if the iterator points after the last element, its "current position" field references the list head.

Token Identifiers The integer field of the BufferedValue iterator, called minBeginMark, serves the following purpose: The compareTo() method (see page 82) must be able to perform an "in document order" comparison between any two "begin" tokens' Identifier objects. In order to reduce this operation to a simple integer comparison, the BufferedValue iterator allocates sequential numbers for all "begin" tokens and stores them in their Identifier objects before returning the tokens to the rest of XL.

Obviously, because the sequential numbers are per-iterator rather than per-value, they can become inconsistent in many cases. For example, if a first iterator starts reading a value and part of the document is deleted after it has passed over it, then a second iterator for the same value will assign different sequential numbers to identical tokens, overwriting the first iterator's values in the process.

However, nothing in the implementation prevents these inconsistent values, hidden inside their Identifier objects, from being compared, even though the result of such a comparison is generally unpredictable! The only safe way of using compareTo() is to use it exclusively for comparisons of objects which have been returned by the same iterator.

It is not clear whether all of the current XQRL XQuery implementation adheres to the restriction of never comparing Identifiers from different sources, nor whether future versions will adhere to it. However, in practice this simple scheme for the implementation of compareTo() has worked very well.

Deletions and Insertions The deletion of a node in the document, i.e. from the "begin" token which is passed to delete() in an Identifier up to the corresponding "end" token, is quite easy to implement: Once the destination MyLinkedList object has been extracted from the Identifier in the "begin" token, we only need to follow its "twin" field to find the corresponding "end" token's MyLinkedList, and can delete the part of the list between these two entries.

The implementation of insert() is more interesting: As specified on page 83, insert() is not only passed the Identifier (i.e. position in the stream) of a "begin" token and the data to insert, but also *how* to insert it. If null is passed as the destination position, the insertion is simple as the final position of insertion is either the beginning or the end of the document. However, if a non-null destination is given, some navigation through the document is necessary. In the low-level MyLinkedList, insertion is carried out by specifying the token *before* which to insert. This token must be found for all of the different possible values for the beforeAfter parameter:

BEFORE To insert before the "begin element" token which is passed to insert(), no further navigation is necessary – insertion takes place immediately before this token.

AFTER To insert after the "end element" token of the "begin element" which is passed to insert(), we first need to traverse its "twin" field to the "end" token, and then use the "end" token's "next" reference to reach the token before which to insert.

INTO_FIRST To insert a new first child of the "begin element" token which is passed to insert(), it is necessary to move forward in the stream (i.e. follow "next" references). First, we need to skip over the "begin" token itself, and then over a QName token which contains the element name. For an older version of the XQRL token stream format, this is already sufficient; the token after the QName token is the one before which to insert. However, in later XQRL versions, an additional, second QName token with type information follows. The BufferedValue implementation can deal with both formats, by checking whether a second QName is present in the stream, and skipping it.

INTO_LAST To insert a new last child of the "begin element" token which is passed to insert(), we only need to traverse the "twin" field to reach the "end element" token. Insertion of the new data then takes place before it.

6.1.3 Required Changes in the Rest of XL

With the new XL_Value implementation and the BufferedValue module in place, the XL runtime system became functional again. However, no code in the rest of XL yet took advantage of the new features of the class. Most notably, the modification methods like insert() were not used, instead the implementations of XL statements like **insert** still copied the entire data of an XL_Value to perform the insertion.

Eliminating Unnecessary Copying of Values Adapting the XL runtime system to make use of the changed XL_Value interface proved to be a task which, while not more difficult than the implementation of the new interface, nevertheless required as much time. Much of this time was spent browsing the classes of the XL implementation and tracing what happens to an XL_Value after its creation.

The following property of the old code turned out to be a problem: Internally to XL, variables were stored in a way which made it difficult to ensure that the same variable always kept referring to the same XL_Value: With the old implementation of the class, XL_Values were immutable, so various parts of the code would replace references to them with references to new objects.

With the new XL_Value implementation, modifications happen in-place, and unnecessary copying of XL_Values must be avoided, because the whole point of introducing the insert() etc. methods is to prevent that the XL **insert** and related statements have to process each token in a document only to perform a potentially very minor modification to it.

It turned out that during the execution of an **insert** statement, the value to be modified was copied no less than *four* times:

- The **insert** implementation copied all tokens, adding the inserted ones at the right position and appending all of them to a temporary java.util.LinkedList.
- To access the contents of the value to be modified, the **insert** implementation created an iterator using XL_varExp.getIterator(), which called XL_Expression.getIterator(), which in turn created a copy for no reason at all before returning the iterator.
- Once all tokens of the new value were in the LinkedList, it was converted to an array, and that array used for the creation of a new XL_Value.
- When writing back the modified value, the XL_localControlBlock.setValue() method added special “begin document” and “end document” tokens at the very start and end of the token stream. Because XL_Values were immutable, this required another copy of the entire value.

In the new XL code, all of the code making these copies is eliminated: A rewritten implementation of the **insert** and related statements (described below) made the first and third copy above unnecessary, a modification of Expression.getIterator() disposed of the second copy above (it is unclear why that function was written to take copies), and changes made by others removed the requirement that documents are enclosed in "begin/end document" tokens, which also made the last copy superfluous.

XL Document Modification Statements Section 2.3.2 has already introduced the five statements in XL which allow modification of XML documents:

- **insert** new tokens into/before/after an existing node
- **delete** a node
- **replace** a node or attribute with a different one
- **rename** an element or attribute
- **move** an attribute or node to a new position in the document

The previous implementation for all of these statements, located in the classes de.TUM.RTS.Stmt.XL_InsertStmt, XL_DeleteStmt etc., not only performed its work by copying the entire value, it also failed to work correctly for certain cases. The rewritten implementation behaves correctly – for example, **insert** is prepared to deal with all combinations of the cases that the source/destination expression evaluates to an element, attribute or nothing. As appropriate, it either carries out its work or raises an error, for example "cannot insert into an attribute".

Because a quite large number of different cases needs to be distinguished for the various modification statements, and since future modifications to the code could easily make some of them fail in a non-obvious way, a number of test XL programs were also written, one for each of **insert**, **delete**, **replace**, **rename** and **move**. They make different modifications to example documents and print out the results. A testing framework which was added to XL during the time this thesis was written proved to be useful for comparing the output of these test programs to the expected, correct output.

6.2 Support For Persistent Variables

To make XL support persistent variables, work was necessary in a number of areas:

First, there has to be a way for the XL user to select a storage module and pass parameters to it, as described in section 6.2.1. Above all, these will specify where on disc to store the data, but they might also include other tuning parameters.

Next, it is necessary to implement the changed variable initialization semantics: If upon starting the XL program a value for a persistent variable is found on disc, the variable is not initialized as usual, instead its value is reloaded from disc – see section 6.2.2.

Finally, there must of course be a storage module which actually writes values to disc – section 6.2.3 discusses the points which influenced the choice of the storage solution, and section 6.2.4 the implementation of the module.

6.2.1 Managing Storage Back-end Modules

The most important parts of the framework for selecting modules and passing parameters to them have already been introduced in section 5.2.1: Each storage module calls XL_Value.registerStorageManager() to add itself to the list of modules maintained by XL_Value. By implementing StorageManager.printOptionInfo(), the module is able to indicate which module options it supports. Later, when a particular module has been selected, its StorageManager.parseOptions() method implementation is called with a parameter string.

Once the relevant code in XL_Value and the storage modules was present, support for module selection was added to the runtime system. As specified in section 5.1.3, this involved extending the command line interface with a new "`-storage`" command line switch, by adding appropriate code to the class de.TUM.RTS.XL_RTS.

If "`-storage`" is present together with a string argument that follows it, XL_RTS breaks up the string argument at the first colon, treats the part before the colon as the name of the storage module to select, and the part after it as the parameters to pass to that module. The correct StorageManager is looked up using its name, and its parseOptions() method is invoked. (Optionally, the colon can be omitted – in this case, all of the string is interpreted as the module name and no parameters are passed to the selected module.)

For example, if a storage manager called XL_Value.registerStorageManager("`mymgr`", obj) from its initialisation code and the user specified "`-storage mymgr:option1=x,option2=y`" when starting XL, then obj.parseOptions("`option1=x,option2=y`") will be called.

6.2.2 Changed Semantics of Variable Initialization

Section 5.1.1 describes the way in which the initialization of persistent variables differs from the initialization of non-persistent ones. In the XL implementation, the relevant changes have to be made both for conversation variables and global variables.

Conversation variables Conversation ("**context let**") variables are uniquely identified by the web service URI, the conversation URI and the variable name. To create a unique identifier string, these three are simply concatenated in this order, separated by spaces. For example, with the `auction.xl` XL demo application, the unique identifier of the "$auction" variable will read

```
http://localhost:10000/ http://xl.in.tum.de/10 auction
```

Initialization of the variable now happens as follows: First, a new XL_Value is created and its name set to the unique identifier string. If the selected storage module supports persistence and the value was already stored on disc, the resulting XL_Value object is non-empty (i.e. contains at least one token) – in that case, no further action is taken. However, if the new XL_Value object is empty, the initialization expression (if any) is evaluated and the result assigned to the value. Finally, the mapping from the variable name to the new XL_Value object is set up to make the variable accessible by XL programs.

A difficult aspect of this modification was finding the part of the XL code which performed the initialization of conversation variables. A look at the most probable class, de.TUM.RTS.XL_Conversation, confirmed that no part of its implementation sets up conversation variables. Instead, the relevant code turned out to be hidden in the getContext() method of de.TUM.RTS.ControlBlock.XL_globalControlBlock. When changing the variable initialization as described above, the opportunity was seized to move the new code to XL_Conversation and call it from XL_globalControlBlock.

Global variables For global variables, there cannot be more than one instance per web service (i.e. XL program) and variable name, so the unique identifier does not include any conversation URI. Instead, it only consists of the web service URI followed by the variable name. Looking again at the `auction.xl` demo, the unique identifier of the "$bidder" variable reads

```
http://localhost:10000/ bidder
```

The remainder of the implementation work was analogous to that for conversation variables. Unfortunately, this included the problems with finding the code to replace with a new version: For global variables, initialization was not performed directly by evaluating expressions and assigning them to values, but indirectly by adding assignment statements to the start of the web service's initialization operation. This happened in the method de.TUM.RTS.XL_PreProcessor.generalWebserviceProcessing(). With the new code, initialization takes place in de.TUM.RTS.ControlBlock.XL_WebserviceControlBlock and the generalWebserviceProcessing() method is no longer needed.

6.2.3 Choice of Database System

Before writing a module for persistent storage, a decision had to be made which type of storage to implement. This section explains the reasons behind the choice of Berkeley DB as the database "back-end", and the storage of tokens in single records (i.e. one token per record).

The following properties are desirable for the selected storage solution (see also section 4.3):

- Retrieval of tokens should be fast.
- The overhead for storing the token data should be small.
- It should support backups and recovery if possible.
- It must be possible to implement in the time available for the thesis.
- It should be possible to navigate through the document, i.e. a sequential read from the beginning should not be the only operation supported by the data structure.

The last point is not strictly necessary since the XQRL engine currently always reads values sequentially from the start, but this is expected to change with a future version. Furthermore, being able to navigate through the document tree allows for a more efficient implementation of operations such as "insert a new last child of this node", because it is not necessary to read all other children before inserting the new child – instead, one can go directly to the end of the node data to perform the insertion.

When evaluating different solutions for the persistent XML storage, it became obvious that some of the above goals conflicted with each other. The following types of storage were considered:

- Store chunks of token data in a specially written low-level data structure, for example the EOS storage manager described in [Biliris92] (page 50) or the Natix one from [KanneMoer99] (page 61). The resulting storage system would have offered both space-efficient storage of the token stream and good performance. On the other hand, implementing either data structure would have taken up a considerable amount of time (maybe too long to finish it in time) because not only the on-disk management of the data chunks would have to be written, but also some kind of index to allow tokens to be addressed individually, in order to allow navigation through the document tree. Above all, it would have been too much work to also add logging and recovery facilities, so the resultant persistent storage work would not have been as robust as required.

- Store the token data as BLOBs in a relational database system, or in Berkeley DB. This seems to be a viable choice: The XL storage system can take advantage of the recovery capability of the used database.
  In the case of JDBC (*Java database connectivity*, the API to access relational databases from Java), there is no way to insert bytes in the middle of a BLOB, so an XML document has to be split up and maintained as a number of BLOBs, taking care to let these get neither too small (to reduce fragmentation and overhead) or too large (to make insertions/deletions efficient). In the case of Berkeley DB, insertion and deletion is possible at any byte position in the record, so a complete XML document can be stored in one large record.
  Still, this solution has its disadvantages: Building the management of chunked data on top of a database accessible via JDBC, while possible, does not seem a very elegant solution. Also, the unused capabilities of the RDBMS, especially SQL query processing, are bound to add a lot of overhead.
  With Berkeley DB, the management of the binary token data is much easier, but it suffers from a different problem: As long as the whole XML document is stored in one large record, it is very difficult to build an index through which individual tokens of the document are quickly accessible. The reason for this is that the only thing by which tokens can be located in the large record is their byte offset – but that offset can change whenever tokens are inserted or deleted. The index would very likely have to use a special-purpose low-level data structure.
  All in all, this type of Berkeley DB storage looks promising, and it should have been seriously considered for implementation. It was not chosen because of the problems with the index structure and the amount of work to write one, and also because it was simply discovered too late that Berkeley DB supports insertion and deletion in the middle of records.

- Store tokens as individual records and use record numbers as token references. As described in section 3.4.2 (page 60), this makes it easy to represent the XML document's structure. When storing one token per record, the disadvantages are that the amount of overhead for storing the tokens is large, and that the average retrieval time for a single token smaller than with a chunk-based solution.
  On the other hand, the solution has a number of positive aspects: It can build on existing, mature database solutions such as RDBMS via JDBC or Berkeley DB, and can use their logging and recovery facilities. Furthermore, insertion and deletion of tokens is a quite cheap operation. Generally, the solution is simple and can be implemented in the available time.
  Before implementing this type of XML storage, it also had to be decided whether to use the more low-level Berkeley DB library for storing the individual tokens, or whether to resort

to a full-featured relational database system. Since none of the advanced features of the RDBMS are needed and Berkeley "libdb" is known to provide much better performance for the operations it supports, the latter was chosen.

6.2.4 Module for Persistent Storage: LibdbValue

LibdbValue stores XL_Value objects on disc using the Berkeley DB database library, "libdb". It provides the features outlined above and in earlier sections, such as a cache of recently accessed token objects, logging/recovery and support for parsing a string of options and acting on the parameters set by it.

The implementation of the persistent storage module is distributed over several classes, all of them in the de.TUM.RTS.Value package:

LibdbStorageManager Similar to BufferedStorageManager for BufferedValue, this class acts as a factory of LibdbValue objects, and parses option strings passed to the XL runtime system by the user.

LibdbValue Main class, with code for value modification and management of the low-level data structures and the token cache.

LibdbPtr An abstraction of a reference to a token. A token can be in memory, which makes it accessible via a Java reference, or on disc, accessible via a logical record number, or it can be both on disc and in memory. This class allows the same types of operations no matter where the token is stored.

LibdbList Implementation of a doubly linked list, identical in layout to the one from BufferedValue (see figure 16 on page 92), but using both logical record numbers and Java references, depending on whether tokens are on disc or in memory. All entries of the list can be on disc, in memory, or both, except for the list head, which is always in memory (possibly in addition to being on disc).

LibdbIdentifier Implementation of com.xqrl.tokens.Identifier, which adds a new private class member to the object (a pointer into a LibdbList) to allow tokens to be found by the persistent storage module.

LibdbIterator An implementation of XL_Value.Iterator which iterates over a LibdbList.

LibdbSerialize Class for the conversion of a byte array into a token object and vice versa.

WeakList A simple linked list implementation similar to the standard java.util.LinkedList, but with the difference that it uses weak references (that is, references which do not prevent

the referenced object from being garbage-collected) and that an entry in the list, WeakList.Entry, can be deleted from it using just a reference to the Entry. This class is used for LibdbValue's token cache.

Supported Options The storage module registers itself with XL_Value under the name "libdb". LibdbStorageManager allows the following options to be passed to the module:

`dir=`*directory-name* Specifies the pathname of a directory on disc in which the data of the persistent variables should be stored. The directory must already exist. The name used by default if "`dir=...`" is not present is "`xl-storage`". A number of files is written to the directory; apart from the actual data (in a file called `xldata.db`), Berkeley DB creates files for temporary data and log files.

`init` Runs the libdb recovery, e.g. to clean up after a crash. It is no problem to specify this even if no recovery is necessary because the database was shut down cleanly. However, it is very important *not* to use "`init`" if another instance of XL is already accessing the database.

If there are two or more options, they must be separated with commas. Additionally, when specifying them with the "`-storage`" switch on the command line, the XL runtime system requires them to be preceded by the module name and a colon, so an example of how to select the `LibdbValue` module and let it run recovery on a Berkeley DB database stored in the `database` directory would be:

```
-storage libdb:dir=database,init
```

Managing the Database The database is closed down automatically when the Java virtual machine terminates, with the help of the mechanism described in section 6.1.1, which calls the LibdbStorageManager's shutdown() method. If the instance of XL whose virtual machine exits was not the only one accessing the database, Berkeley DB takes care not to close down the entire database.

The database is correctly shut down if XL is interrupted by pressing Ctrl-C on the Unix command line, or by sending a SIGINT signal to the JVM process with the "`kill`" command. It cannot be shut down if the process is terminated with a SIGKILL signal (i.e. with "`kill -KILL`") – in that case, the database recovery must be run when the database is next used.

If necessary, recovery can also be performed without restarting XL, by using the "`db_recover`" command line utility which comes with Berkeley DB.

To preserve the database even in case of a catastrophic failure (i.e. loss of all the data on the machine's hard disc), it should be backed up at regular intervals.

The details of taking backups of a Berkeley DB database are described in its documentation [Sleepycat]. Berkeley DB supports both "standard" backups which require you to shut down all programs accessing the database, and "hot" backups which work while there are active database transactions. Making backups requires the use of the "`db_archive`" tool to identify log files which can be backed up, "`db_dump`" and "`db_reload`" to save and restore the database files, "`db_recover`" to perform the recovery, and "`db_checkpoint`" for checkpointing the database.

Document Modification Since the type of doubly linked list used by LibdbValue is quite similar to the one of BufferedValue, so are the procedures to access and modify them, e.g. for finding the point in the list to insert at. However, the implementation of the insert() and delete() methods is complicated by the following aspects:

- All accesses to the database must be wrapped in code which begins and commits transactions. Since a transaction can fail, provisions must be made to retry it a certain number of times and only to abort with an error if it keeps failing.
- Since a transaction can abort at any time, it is not safe to update in-memory structures at the same time as structures on disc. Instead, the data in memory can only be modified after the transaction has successfully committed.
- Throughout the code, "double bookkeeping" is necessary because tokens are accessible both via references and logical record numbers. Despite the attempt to encapsulate this in the LibdbPtr class, it adds a considerable amount of complexity.
- The storage module includes its own cache of recently accessed tokens, and this must be kept up to date. The cache implementation interferes with many parts of the rest of the code – for example, token objects can get pushed out of the cache (and become invalid) while they are still referenced elsewhere.

In general, before accessing tokens, they are fetched into memory with a call to the internal cache code of LibdbValue. The implementation of delete() (remove a node from the value) and also of clear() (remove all tokens from the value) is an exception: It would not be appropriate to unserialize the data for each token and to create an object just to delete it, so these functions bypass the cache and directly work on the in-memory and/or on-disc data using a LibdbPtr.

Token Cache The token cache is a buffer which holds a number of Token objects – the exact number is configurable via XL_Value's setCapacity() method. There are several reasons why the tokens are held in the cache in this form rather than in serialized form, as an array of bytes:

- Unserializing the byte array to create a token is a relatively expensive operation, it should not happen all the time.
- If the data were stored in serialized form, then after unserializing it and creating the token object, that object would be "thrown away" immediately after use even though it is still in memory and would probably not be reclaimed by the garbage collection for a while – one should take advantage of the fact that the object is still present.
- A large number of XL_Value objects represent XL variables which are never written to disc. Storing them in serialized form in memory instead of in their "natural" form as objects would only slow down accesses unnecessarily. An alternative idea might be to store values as objects as long as the value will never be written to disc. However, it is not possible to know in advance whether a value will ever need to be written to disc (even non-persistent values can be "swapped out" if they become too big), and section 6.3 describes the problems which arise when trying to "convert" values from one type of storage to another.

The central function of the cache (called LibdbValue.getEntry()) is responsible for returning a token object given either or both of a reference or a logical record number. If the object is not already in memory, the method fetches the relevant record from the database and unserializes it. Additionally, both for the case that the token was already in memory and the case that it wasn't, it is marked as "accessed" for the replacement algorithm.

The caller of the getEntry() method typically stores the returned object reference alongside the logical record number which it used in the call to the method – due to the "double book-keeping" mentioned above, every logical record number in the LibdbList is accompanied by a corresponding object reference.

It is possible that only a logical record number is passed to getEntry(), but an object for the requested record is already cached in memory. For example, this can happen when following the "twin" reference from a "begin" to the "end" token, if the "end" token was still cached, but the "begin" token has just been read from disc. To allow the cached "end" token to be found, LibdbValue maintains a java.util.HashMap which maps logical record numbers to object references. (The HashMap also has the additional purpose of ensuring that unreferenced but cached tokens are not garbage-collected.)

LibdbValue is "lazy" with regard to updating in-memory references of stored values' tokens: Any references to cached tokens are only set up or refreshed when the reference is about to be followed. This can lead to slightly better performance in some cases – for example, when loading a persistent value from disc for a sequential read, the "twin" references will not be set up because they are not needed by the sequential read.

On the other hand, LibdbValue is not "lazy" when updating an on-disc reference: As soon as the data for a token has been entered into the database and libdb has assigned a logical record number to it, that logical record number is entered in all other tokens which reference the token (such as the "next"/"previous" and "twin" tokens). This is necessary because if a token is written out to the database, this implies that the replacement algorithm has decided to drop this token from the cache; apart from updating the logical record number fields, it is also going to set all direct references to the object to null, allowing it to be garbage-collected. Hence, the logical record number becomes the only reference by which the token data can be retrieved again.

The LibdbValue cache is "write-through" for persistent variables: If a persistent variable is modified, these changes are immediately written to the database. This is required because of locking problems (described further below), which preclude us from using e.g. one large Berkeley DB transaction for each insert() operation – instead, several smaller transactions are used to reduce the chances of deadlocks.

The replacement algorithm used by the cache is a simple LRU (*least recently used*) algorithm: Whenever a token is accessed, it moves to the head of a fixed-size queue (implemented with a WeakList). Later, as other tokens are accessed and added to the start of the queue, it keeps moving towards the end. The moment it "drops off" the end of the queue, it is written to the database (of course this is not necessary if it has already been written earlier, e.g. because it belongs to a persistent value).

The whole process is complicated by our requirement that the tokens of temporary values, which are only referenced by the XL runtime system for a short time, should never be written to disc. If the XL runtime system has finished working with a temporary value, all its Java references to it disappear, so the XL_Value object would usually be garbage collected. However, if the tokens are in the cache, a Java reference to them exists from within the cache's data structures, so the garbage collection will not reclaim the memory they occupy. If the cache did not make provisions to detect cache entries whose tokens are not referenced by any other part of the program, these entries would even eventually be written to the database – only once this has happened, their tokens would become unreferenced and thus eligible for garbage-collection.

The cache code's solution to the problem is to use weak references for the LRU list, then the garbage collection can remove the object if the weak reference to it is the only remaining reference. Furthermore, the finalize() method of the token data (more accurately, of the LibdbList object which in turn references the Token object) removes the entry from its current position in the LRU queue when the token becomes unreferenced. In general, this area of the cache management proved to be one of the more difficult parts of the implementation due to various subtleties surrounding weak references and the Java garbage collection.

The LibdbValue cache implementation also needs to deal with a number of other problems which are not explained here because describing them would require going into too much detail. As an example, here is one such problem:

What happens if a token is to be written to the database, but e.g. the token following it has not been written out yet? The on-disc token data needs to include some kind of reference in its "next" field. If the next token were already on disc, its logical record number could be used, but since this is not the case, no such number has yet been allocated for it.

The "obvious solution" for this problem would be: Allocate a logical record number for each token, whether it will be written to disc or not. If the token never ends up on disc, that record number just remains unused.

Unfortunately, this does not work too well for Berkeley DB: The database library does not use a B-tree as the index which maps logical record numbers to physical identifiers, but a simple on-disc array. This way, if a record number were allocated by LibdbValue, but then not used, some space in the array would be wasted, and if LibdbValue first allocated the logical record number "1" and next the logical record number "1000", libdb would create 999 "empty" entries inbetween them.

Several workarounds exist for this problem. One would be simply to use a B-tree based database for the token data instead of the "recno" database whose behaviour is described above, at only a small performance penalty. Another one, which is used by LibdbValue, is to introduce a separate set of identifiers for tokens which have not yet been written to disc. The "next" field of a token in the database can then either contain a logical record number or such a "memory object ID", and an additional bit flag indicates which type of identifier is used. Another HashMap similar to the one used above for logical record numbers maps these identifiers to Java references.

Another problem that LibdbValue has to address is that of stale LibdbList objects still being referenced by long-lived LibdbIterator or LibdbIdentifier objects: Both of these classes include private members which reference a LibdbList. If it becomes invalid because the object is dropped from the cache, a flag is set in the LibdbList object. In case the iterator or Identifier later accesses the object again, it notices that it is flagged as invalid, and refreshes its reference to it.

Database Layout The low-level layout of the Berkeley DB database used for storing the data of XL_Value objects is fairly simple. There are two tables in the database. (In libdb terminology, each such table is called a "database", which can be slightly confusing – we will use the term "table" even though the libdb variant is much simpler than a table in a relational DBMS.)

- A B-tree table called "persistent" provides a mapping from the unique identifier string of a value to the logical record number of the list head object for the token stream.

- A "recno" table called "token" allows access to the token data by logical record number.

Both of these are created in a file called `xldata.db`, which is stored inside the directory that the user specified when selecting the libdb storage module.

In "token", both tokens of persistent and non-persistent values are stored – the tokens of non-persistent values are only distinguishable from the persistent ones by the fact that they are not reachable through a series of references originating from one of the logical record numbers in the "persistent" table.

The format of an individual token entry is also straightforward: After an initial fixed header with flags and references e.g. to the "next" and "twin" tokens, the serialized token data follows.

Locking Issues In order to improve performance, the XL runtime system starts several threads inside the Java virtual machine, and optimizes XL applications to run in parallel on more than one thread as much as possible. The individual threads execute XL statements, which can obviously access and modify XL_Values.

From the point of view of the LibdbValue class, the extra parallelism is a problem: Although Berkeley DB allows concurrent access to its databases, within one process the library is single-threaded, i.e. it may only be accessed by one thread of control at a time. To ensure that this is always the case, LibdbValue makes use of the Java synchronized construct. Unfortunately, the object to synchronize on must be a global object (it cannot be just the XL_Value instance being modified), so this step implies that threads will often not run in parallel. In the light of this effect of the synchronization within LibdbValue, it seems questionable whether parallel execution of XL code will still result in any noticeable performance improvement.

A completely different type of locking problem became obvious during the tests of the implementation with the `auction.xl` example application (see section 7.5), which executes three separate XL programs which interact with each other. Originally, LibdbValue was designed to allow this type of operation, but then the following type of deadlock was observed:

- Two instances of XL run in parallel on the same machine (as two separate processes) and use the same database.
- XL process A sends a SOAP message to process B.
- While sending the message, A reads from a token iterator, which causes new tokens to be entered in A's token cache.
- As new tokens enter A's cache, old tokens are written to disc. This results in libdb acquiring a write lock on the database page the tokens are stored in.

- B receives the start of the SOAP message from A and stores it in an XL variable. The tokens of the variable value are to be stored in the database, but the page they are to be written to happens to be locked by process A.

- A waits on B to read the rest of the message, and B waits on A to release the write lock in libdb – deadlock has happened in a way which is not detectable by libdb.

The cause of this deadlock is that Berkeley DB does not support locking with a granularity finer than whole pages – it cannot lock individual records.

In response to this problem, first long-lived transactions (e.g. one transaction for every insert() operation) were broken up into smaller parts, to reduce the number of locks held by individual transactions. Furthermore, dirty reads were enabled in libdb – the way the database is organized (the content of records holding token data never changes), it can be ensured that reading a token does not actually access "dirty" data. This significantly reduced the number of deadlocks – unfortunately, deadlocks can still occur, so it is not recommended to run several XL instances on the same back-end database.

6.3 Buffering Recently Accessed Tokens

The previous section describes the cache of the LibdbValue storage module. As can be seen, the module's cache is integrated into the rest of its implementation. This reflects a change that was made in the original architecture of the persistent storage system. This section explains the original approach, the reasons why it failed and the changed approach that was finally implemented.

First Approach: Global Value Cache When the architecture for the new XL_Value subsystem of XL was first designed, it was with the following thought in mind: Since over time several storage modules will be written (e.g. one for Berkeley DB, one for JDBC, maybe one with a special-purpose XML storage manager), it would be useful to move common functionality of these modules into other parts of the persistent storage system, so that implementing an individual module would require as little duplication of work as possible.

The cache functionality was an obvious candidate: There did not seem to be a reason not to introduce an additional "cache layer" inbetween the XL_Value class and any back-end storage module classes.

Since typical XL variables are either very small (single tokens or small XML documents) or very large ("database variables" of an XL application), it looked promising to take the following approach to caching: Values are always held entirely in memory or are entirely swapped out to disc. If they are in memory, they are stored as Java objects, and if they are on disc, they are only

stored in serialized form in the database, and objects are created on demand. If a value becomes too large, it is written to disc.

During the implementation of a first version of LibdbValue it became obvious that this way of caching would not work, at least not in an efficient way. The central problem turned out to be the process of moving a value that has become too large from memory to disc. The moment this happened, all Identifier objects and all iterators pointing into the value would become invalid – but this is unacceptable because both the XQRL XQuery implementation and XL maintain iterators and Identifiers for a value while e.g. inserting data into it.

When the original approach to caching was designed, some thought went into allowing identifiers and token iterators to be "relocated" so they would point to the new on-disc value after that value had moved out of the cache. In practice, such a relocation operation was not feasible; it would have been too expensive to find the corresponding position in the version of the value that is stored in the database.

A workaround which was pursued for a short time was to prevent values from moving to disc during critical parts of the code, by "locking" them in memory. Apart from the fact that it would have lead to out-of-memory problems if a huge amount of data was inserted into a locked value, this would have required a large number of changes in the XL code that accesses XL_Values – for example, the implementation for the XL **insert** statement would have to be bracketed by calls to "lock" and "unlock" methods. As a further complication, with this complex caching policy in place, it became very difficult to come up with a replacement strategy for the cache which would behave in a reasonable way.

Second Approach: Cache For Each Storage Module At this point, it was necessary to re-evaluate the architecture decision. Since there was no better alternative, the current architecture with its requirement that each storage module implement its own cache was chosen. One should be aware of the following disadvantages of this solution:

- A duplication of effort is necessary for each new storage module.
- The code for storage modules gets significantly more complex. Unfortunately, the final LibdbValue implementation indicates that this is only too true, despite its very simple LRU cache.
- Access to values which are completely buffered in memory is much slower than it could be. In the case of LibdbValue, the "double bookkeeping" of in-memory references and on-disc references, together with the time needed to update the LRU statistics, results in

access speeds which are an order of magnitude slower than comparable BufferedValue accesses (see the benchmark on page 116).

Still, this solution is better than a global value cache – the experience with trying to implement such a global cache has made it clear that buffering data is tied too closely to a particular low-level storage solution to be moved to a separate layer.

6.4 Further Issues

In addition to the work described in the previous sections, a number of other miscellaneous changes were made to the XL code. Many smaller bug fixes and improvements are not mentioned here because they are trivial or not very interesting.

Support For Nested Exceptions With some small changes to the class de.TUM.Exception.XL_RunTimeException, the ability to handle nested exceptions was added. This means that an XL_RunTimeException can contain a reference to another exception object, which is interpreted to represent the cause why the XL_RunTimeException was raised.

When the Java runtime environment prints out a stacktrace for such an exception object, it not only includes the trace to the point where the XL_RunTimeException occurred, but also the error message and stack trace of the exception which caused the XL_RunTimeException. This information is especially useful during debugging, because it is immediately obvious what error (for example, a NullPointerException) is responsible for the XL_RunTimeException.

Revised XL_IntegerValue and Related Classes Originally, a number of subclasses of XL_Value existed in the de.TUM.RTS.Value package, one for each primitive data type. For example, there was an XL_IntegerValue to create an XL_Value that only contained a single integer token, an XL_StringValue which only contained a single string token, etc.

These subclasses are not a good idea: The resulting new types do not add any new data members or methods to XL_Value, the only extra functionality is in the constructor. Additionally, nothing prevents one to take e.g. an XL_StringValue object, erase it and to insert completely different data.

For this reason, the revised versions of these classes cannot be instantiated by the rest of XL. The classes only have static create() methods which return XL_Value objects. This change implies that throughout the XL code, expressions like

```
new XL_StringValue(str)
```

had to be replaced with

```
XL_StringValue.create(str)
```

"Auction" XL Demo Application The "auction.xl" demonstration application is accompanied by several JSP (Java Server Pages) files which provide a web interface to the features of the XL program.

Apart from the usual work of adapting these files to the changed interface of XL_StringValue and its related classes, `auction.jsp` was extended to return more meaningful error reports if a request for the page fails. In particular, the error includes a stack trace, which helped during the debugging of errors in the new XL_Value implementation which only manifested themselves with the auction example.

The file `print-xl.jsp` is used by `auction.jsp` to pretty-print the source code for the XL programs it executes. A quite serious security problem was fixed in it: By passing the right parameter to the page, an attacker was able to view the contents of arbitrary files on the system, as long as they were readable by the Tomcat JSP application server.

Porting the Persistent Storage Subsystem to a New XL Version During the practical part of this thesis, the XL code underwent modifications to allow it to work with a new version of the XQRL XQuery engine.

Unfortunately, the XQRL API changed significantly, requiring numerous changes in many parts of the XL implementation. During the time that these changes were made by others, testing and extensions of the persistent storage subsystem continued with the last "known to work" version of XL.

Once the new XL version was reasonably stable, the new classes of the de.TUM.RTS.Value package had to be ported to the new branch of XL. Because of the relatively large differences between the two branches, this was not as straightforward as expected.

7 Testing

7.1 General Approach

The general philosophy followed when writing the different components was not to let bugs come into existence in the first place, by following a rigorous testing policy:

- Testing is performed at multiple levels in the code: At the lowest level, assertions in the code catch inconsistencies of data structures or parameter values. This allows the detection of programming mistakes close to the location of the bug, rather than resulting in strange behaviour of other parts of the code.

- At the next level, slightly more expensive tests (in terms of their running time) walk whole data structures and check whether they are valid. In practice, they are also called via assertions, but conceptually, the checks can be regarded as a separate layer. They are not enabled by default because they are too expensive, but were used during the implementation phase to spot mistakes in the code.

- Next, component tests ensure that each component complies with the API specification it offers.

- Finally, entire XL programs and applications are used to test the complete system.

7.2 Low-level Integrity Checks

At the lowest level, a lot of errors in the code can be caught with the consistent and liberal use of assertions, i.e. run-time invariant checks. Assertions have been a part of the Java language for some time and can be a great help when tracking down bugs because they cause the program to fail with an exception close to the part of the program that behaves incorrectly, rather than letting that part execute and cause strange failures later on, possibly in completely different parts of the code.

It is sometimes argued that additional run-time checks slow down the program, but in fact the very slight increase in running time will always be greatly outweighed by the developer time that is saved with assertions, and the fact that “remote debugging” of non-reproducible problems of users becomes much easier. Furthermore, the Java runtime system even allows assertions to be switched off if necessary.

In the various classes of the new XL persistence code, assertions are primarily used for two purposes:

- Checks of parameters passed to methods. The Java API documentation recommends not to use assertions for this purpose and to introduce new types of exceptions instead – however, in the case of the code which was added to XL, introducing new types of exceptions seems overkill because all callers of the methods are part of the XL runtime system. The assertions for argument checking are always accompanied by a comment which explains the invariant that is checked.

- Checks of data structures. If some part of the implementation fails to set up or modify a data structure properly, this usually only leads to problems when the data structure is next accessed. Moreover, subtle problems can go completely unnoticed. An example for this existed in the code of LibdbValue: An error in the code resulted in records being repeatedly fetched from disc rather than being retrieved from the cache.

The latter of the above checks can be subdivided further, into simple "local" checks which only take a constant amount of time, and more complex and expensive ones:

The implementation of the BufferedValue and LibdbValue classes includes code which performs a complete check of a value stored in the respective class. In the case of LibdbValue, this also includes checking whether the cache of recently accessed tokens is consistent with the information stored in the value. Since such a check is too expensive to be called under normal circumstances, it is not executed by default and must be called explicitly via XL_Value.assertValid() if required.

7.3 Component Tests

The XL storage system only includes one component test: The class de.TUM.RTS.Value.TestValueAPI creates a number of XL_Value objects and performs the various supported actions on them, such as reading the value with iterators, making insertions and deletions, and clearing the value. After each step, the value is compared to the expected result and an error message is printed if the two differ.

TestValueAPI was used with the BufferedValue and LibdbValue storage back-ends to test for a correct implementation of the basic operations of XL_Value. However, it is a relatively simple and undemanding test – for example, it does not test concurrent accesses to XL_Value objects.

7.4 XL Test Programs

A number of smaller XL test programs were written to test the features of the new XL_Value class, such as persistent storage and efficient insertion into values.

The `stack-server.xl` and `stack-client.xl` programs are simple example web services: The server program maintains a persistent variable with a list of entries and offers access to it via the standard operations of a stack. These operations are executed by sending messages containing "push", "pop" and "print" commands to the web service.

`stack-client.xl` is a simple XL program which allows the different commands to be sent to the server.

The two XL programs were useful for testing the new persistence features of XL – despite being small, they use most of the new features of the language.

Several other XL programs were written for a slightly different purpose: As mentioned before, the different XL document modification statements, **insert**, **delete**, **replace**, **rename** and **move**, need to work in a variety of different cases. Since there is a risk that changes to the code make it fail in some of these cases, five XL programs test the different possibilities for each type of statement, and corresponding entries in the `tests.xml` file compare the programs' output to the expected output. By executing the XL test suite with "`ant -f test.xml`", there is a convenient way to run tests on the document modification statements as well as other parts of the XL runtime system.

7.5 Complex Test Application: auction.xl

`auction.xl` was the final part of the testing effort which was undertaken to ensure that the new XL_Value implementation is correct. This demonstration application was already present before work on the new persistent value support began, and is special due to the fact that it features three instances of XL which run as separate processes. The processes interact with each other by sending SOAP messages. Quite often, they run in parallel.

As it turned out, the concurrent accesses to XL_Value objects uncovered a number of problems in the implementation of the LibdbValue class. The first problem was that sometimes several threads would attempt to access the same XL_Value object at the same time. By using the synchronization features built into Java, access to the object was limited to one thread at a time.

Another problem has already been mentioned in section 6.2.4: Concurrent access failed if several instances of XL used the same database to store their persistent variables, due to deadlocks. Unfortunately, the modifications made in response to the problem do not eliminate the deadlocks entirely. Since this type of usage will be quite uncommon in practice (different instances of XL web services typically run on separate machines) and it is no problem to simply store different XL instances' data in separate directories on disc, it is believed that the continuing locking problems do not seriously reduce the usefulness of the persistent storage system.

8 Performance Tests

The final version of the new XL_Value implementation was also tested with regard to its performance. All tests took place on an IBM compatible PC with an AMD XP 2100+ (1733 MHz) running Linux 2.4.19, with 512 MB of memory and using Sun JDK 1.4.0. All accessed files were already buffered by the operating system, and all involved Java classes were already loaded before timing for a test case began.

8.1 Small Documents, Modifications

The tests described in this section are performed and timed by the class de.TUM.RTS.Value.TestAccessSpeed. It concentrates on comparing the access times for different types of value storage, with small values.

The different storage implementations used have the following properties:

BufferedValue Stores all tokens in memory, no support for persistent values – requests to make values persistent are ignored.

LibdbValue, buffered, non-persistent The storage module has persistence support, but this value is never written to disc because the cache is big enough and the value is not declared persistent.

LibdbValue, buffered, persistent As above, but the value is declared persistent. For Libdb-Value, this means that all modifications are write-through.

LibdbValue, unbuffered The implementation's in-memory cache is switched off completely, every token access involves a database access and (de)serialization. Behaviour is identical for persistent and non-persistent values.

The following different test cases were executed many times, then the average execution time was calculated:

Read 23 tokens Use an iterator to read the complete contents of a value with the content "<foo a="b" c="d" e="f" g="h" i="j"/>"

Clear & write 23 tokens Clear the value, then insert the tokens that the empty-element tag "<foo a="b" c="d" e="f" g="h" i="j"/>" consists of.

500 inserts Start off with a value of "<database/>", then insert "<entry>Text</entry>" 500 times as the new last child of "<database>".

500 reads & inserts Start off with a value of "<database/>", then repeat 500 times: Read whole value content, insert "<entry>Text</entry>" as the new last child of "<database>".

500 reads & inserts in XL Time needed for the execution of the following XL operation (excludes the startup time of the XL virtual machine):

```
service http://localhost:3328/

    let $x := <entry>Text</entry>;
    let $database := <database/>;

    operation x
        body
            let $i := 0;
            while ($i < 500) do
                insert $x into $database;
                insert $x into $database;    !! insert statement repeated 100×
                ...
                let $i := $i + 100;
            endwhile;
        endbody
    endoperation

endservice
```

For the "non-persistent" test case, the declarations of $x and $database were moved inside the body of the operation. Note that the XQuery implementation in XL reads the entire value during every insert, even though this is not actually necessary to perform the insertion.

The results when repeatedly reading and writing 23 tokens are as follows:

	Read 23 tokens	**Clear & write 23 tokens**
BufferedValue	0.00072 ms (1400000/s)	0.00441 ms (227000/s)
LibdbValue (buf., non-pers.)	0.00302 ms (331000/s)	0.02005 ms (49880/s)
LibdbValue (buf., persistent)	0.00279 ms (358000/s)	9.65 ms (103/s)
LibdbValue (unbuffered)	3.57 ms (280/s)	4.65 ms (215/s)

As expected, BufferedValue performs a lot better than even a buffered LibdbValue, reading and writing the tokens almost an order of magnitude faster. This is due to the fact that with

BufferedValue, reading a token only requires following a single in-memory reference, and similarly, writing tokens only involves changing a few references in the linked list implementation.

In contrast, when accessing a fully buffered LibdbValue, the implementation needs to acquire a lock via the Java **synchronized** statement, then check whether a buffered token is still valid (i.e. has not been marked invalid after being pushed out of the cache), and finally update the LRU cache statistics for the token before returning it.

Comparing the accesses for (non-)persistent buffered LibdbValues, read access happens with roughly the same speed for persistent and non-persistent values. On the other hand, write access to persistent variables is slower by a factor of 500 compared to non-persistent variables, due to the fact that modifications to persistent variables are write-through.

The numbers for unbuffered LibdbValues are slightly unrealistic – in practice, nobody will want to switch off the class's token cache completely. It is not entirely clear why clearing and writing the tokens is twice as fast as for buffered, non-persistent values – one possible explanation is that clearing the value can be performed more quickly if none of the tokens in the value are in memory: The token cache does not need to be kept updated, and the tokens do not have to be deleted from an in-memory doubly-linked list as well as in the database.

The results for creating an XML document with 500 entries using repeated insert operations are:

	500 inserts	**500 reads & inserts**	**500 r. & i. in XL**
BufferedValue	1 ms	21 ms	2913 ms
LibdbValue (buf., non-pers.)	11 ms	165 ms	2512 ms
LibdbValue (buf., persistent)	1227 ms	1647 ms	4029 ms
LibdbValue (unbuffered)	483 ms	4310 ms	108000 ms

The figures in the left and middle columns show the results for direct accesses to XL_Value objects. As before, BufferedValue performs best, followed first by non-persistent and then by persistent LibdbValue accesses.

The fact that in the case of the first column, accesses to completely unbuffered LibdbValue objects are faster than accesses to buffered persistent objects can probably again be attributed to the LibdbValue token cache. However, it is unlikely that the cache management alone would account for such a large difference (a factor of 2.5) – the exact cause remains unclear.

To compare the overheads in the XL_Value subsystem to those of the rest of XL, the right column shows how long it takes if an XL program performs exactly the same operations on the XL_Value object as a special-purpose Java program does for the middle column. Even though the measured running time does *not* include the time taken by XL to start up and parse the program, the XL program still does not compare favourably at all. A closer look at the data in the top three rows of the middle and right columns shows that the XL runtime system appears to add a

overhead of about 2500 milliseconds irrespective of the type of storage back-end selected. (As before, the results for unbuffered LibdbValues should be disregarded, they are not relevant in practice.)

It may seem puzzling that buffered, non-persistent LibdbValues appear to perform better than BufferedValues with the XL program. However, the numbers are probably slightly skewed, and in reality, the BufferedValue implementation is faster: An effect which was observed during testing, which might also be responsible for the strange numbers above is that the Java garbage collection may have run one time more often during the time the BufferedValue performance was measured.

8.2 Large Documents, Reads

Whereas the purpose of the performance tests in the previous section was to compare the different XML storage modules, the reason behind the following tests is to measure the maximum throughput (in terms of the number of tokens that can be read per second) and to identify any bottlenecks in the system.

Sequential Reads and Random Access The class de.TUM.RTS.Value.TestSeqRead contains code which generates an artificial, large XML document which is declared persistent. Essentially, the document reads

```
<document>
    <entry>This is an entry in the document</entry>
</document>
```

and the <entry> tag is repeated over and over again. The exact number of tokens in the final document is configurable. Having created the value, TestSeqRead attempts to measure the speed in tokens per second once for a sequential read of the entire data and once for random accesses to tokens in the stored stream.

At first, the speed was measured with a relatively small value of 1.500.000 tokens. The test program gives the following results for this case:

Sequential read of 1.500.000 tokens:	203 seconds (7358 tokens/second)
Random access to 1.500.000 tokens:	241 seconds (6218 tokens/second)

The size of the corresponding XML document in ASCII text form is 9.6 MB (6.7 bytes per token), and the size of the Berkeley DB database which holds the value amounts to 56 MB (39.5 bytes per token).

The results show that sequential reads of the data can be performed faster than random accesses. The difference is not as pronounced as one might expect, for the following reason: For normal database applications, the expensive aspect of random accesses is the amount of disc seeks, but in this case, as the 56 MB of the Berkeley DB database did not exceed the amount of available memory (512 MB), all data was cached, so no disc accesses were necessary.

In addition to providing figures for a comparison of sequential vs. random access to a document, the TestSeqRead uncovered the following problem: Reading a token stream from the cached pages of the database into memory and turning it into a stream of Java token objects is not an I/O-bound operation, but very much a CPU-bound operation.

Originally, a second test was planned with a document whose size in the database exceeded the amount of available memory. However, this was abandoned because it would have taken too long to run and because the results would probably not have differed very much from those of the smaller test, due to the fact that disc accesses would only have accounted for a minuscule portion of the total time the test program would have taken to run.

Bottlenecks in the LibdbValue Storage System Instead of measuring the throughput when doing a sequential read of the data, work now concentrated on identifying which parts of the test program took so long to execute. No Java profiler was available to accumulate exact information about which parts of the system take up how much time, but by temporarily uncommenting important sections of the code which is invoked by TestSeqRead and measuring the resulting execution times, the following *approximate* figures can be given. They show how much time each part of the implementation takes up:

Test program (loop, selecting random token positions)	22%
Libdb transaction to fetch token data from database	51%
Unserialize data, create token objects	10%
LibdbValue cache management	16%

Again, the Java garbage collection makes exact measurements difficult: As long as large parts of the code are commented out, far fewer objects are created and as a result, the garbage collection runs less often and takes a shorter time to complete.

The result of this test is slightly disappointing: Most of the time is spent in libdb to retrieve token data. Since it was suspected that much of this time was due to the fact that full logging and transaction support was turned on for libdb, as an additional experiment the test was repeated with transactions turned off. The results showed that transactions indeed slow down access to the data; without transactions, accesses to the token data were almost twice as fast as with them.

9 Conclusion

In section 1, the nature of the problem that was to be solved in this diploma thesis was outlined: XML documents (in the form of variables of the XL programming language) must be stored on disc. In section 2, we took a closer look at the technologies one should be aware of when implementing such a solution for the storage of XML documents, including the XML standard and related standards, the area of web services, and the XL programming language.

Section 3 proceeded to introduce the different possibilities of storing XML data on disc: Flat files can hold the document in its standard format, but are not very flexible. When turning the document into tokens, it is possible to store the data as a stream of tokens, or to add information which allows movement in the document tree. For either case, it needs to be decided whether to store each token individually or whether to create chunks of token data. Furthermore, there is a choice regarding the storage of these tokens or token data: Special low-level data structures exist which ensure fast sequential access as well as efficient modifications, but it is also possible to use a relational database system for the on-disc storage. Finally, a number of different types of index structures increase the performance for certain document queries, at the cost of additional work to create the index and to keep it up to date.

The practical part of the diploma thesis was described from section 4 onwards. That section discussed the requirements for the XL persistent storage system which had to be implemented, notably support for efficient modification of XML data, the ability to store XML documents persistently, and a modular architecture which allows other persistent storage solutions to be added over time.

In section 5, the tasks to be performed by each part of the implementation were specified, together with a corresponding API. Section 6 then concerned itself with the implementation of the different parts, such as the new semantics for XL variable initialization, classes for persistent and non-persistent storage of XL variables, and the choice of Berkeley DB as the database "back-end" for the persistence code.

Finally, in sections 7 and 8, the new implementation was tested to eliminate as many errors in the code as possible and measure its performance for different types of accesses to the XML data.

Due to the maturity of the Java platform, the available compiler and other tools which were used during the practical part of this thesis were well-suited for the given task, and no problems were encountered using them. Instead, a different and unexpected problem turned out to be the existing XL implementation: Because of the organization of the source files and the lack of comments, it took much longer than expected to get to the point where it was possible to start making modifications to the code.

Some interesting conclusions can be drawn from the various performance tests that were conducted: The tests show that the code which handles tokens on disc is rather slow, causing the speed of access to the data to be limited by the available CPU power and not disc throughput.

However, the tests also make it clear that when an XL program executes, accesses to XML variables only account for a quite small percentage of the total execution time. Without attempting to turn this into a justification for the suboptimal performance of the persistent storage subsystem, it must be said that the performance problems in other parts of XL should be tackled first to improve the overall speed of XL programs.

More work would be necessary to identify the parts of XL where optimizations are most needed, but the following two aspects of the current XL version appear to be problematic: First, the XQRL XQuery optimizer is called too often to (re-)compile expressions, and second, the XQRL code currently keeps reading XML documents from the start, making e.g. insertions into the XML data inefficient despite the revised XL_Value class: Even though the insertion itself is now fast, it takes too long to find the point of insertion.

There is room for future improvements in a number of further areas: First, it is possible to create other persistent storage modules which store the XML data in different ways, and to compare them to the LibdbValue implementation with regard to speed and disc space efficiency. For example, it might be worthwhile to implement a storage manager similar to the EOS or Natix ones from sections 3.3.4 and 3.4.2. Furthermore, it would be very interesting to to evaluate different XML index structures by adding support for them to XL.

References

[Biliris92] Alexandros Biliris: The Performance of Three Database Storage Structures for Managing Large Objects, *Proceedings ACM SIGMOD International Conference on Management of Data*, 1992, pages 276–285.

[Cerami02] Ethan Cerami: Top Ten FAQs for Web Services, 2002, O'Reilly & Associates, Inc.; accessed 20 October 2002
`http://www.oreillynet.com/lpt/a//webservices/2002/02/12/webservicefaqs.html`

[Cooper+01] Brian F. Cooper, Neal Sample, Michael J. Franklin, Gísli R. Hjaltason, Moshe Shadmon: A Fast Index for Semistructured Data, *Proceedings of the 27th VLDB Conference*, Roma, Italy, 2001.

[FlorKoss99] Daniela Florescu, Donald Kossmann: A Performance Evaluation of Alternative Mapping Schemes for Storing XML Data in a Relational Database. Rapport de Recherche No. 3680, INRIA, Rocquencourt, France, May 1999.

[GoldmWid97] Roy Goldman, Jennifer Widom: DataGuides: Enabling Query Formulation and Optimization in Semistructured Databases, *Proceedings of the 23rd VLDB Conference*, Athens, Greece, 1997.

[GoldmWid99] Roy Goldman, Jennifer Widom: Approximate DataGuides, 1999.

[Graves02] Mark Graves: Designing XML Databases, Prentice Hall, Inc., Upper Saddle River, New Jersey, USA, 2002.

[Grust02] Torsten Grust: Accelerating XPath Location Steps, *Proceedings of the 21st ACM SIGMOD International Conference on Management of Data* (SIGMOD 2002), pages 109–120, ACM Press, Madison, Wisconsin, USA, June 2002.

[Heun00] Volker Heun: Grundlegende Algorithmen. Vieweg-Verlag, Braunschweig/ Wiesbaden, 2000.

[ISO10646] ISO (International Organization for Standardization). ISO/IEC 10646–2000. Information technology – Universal Multiple-Octet Coded Character Set (UCS). [Geneva]: International Organization for Standardization, 2000.

[JSP] java.sun.com: JavaServer Pages; accessed 3 October 2002
`http://java.sun.com/products/jsp/`

[KanneMoer99] Carl-Christian Kanne, Guido Moerkotte: Efficient Storage of XML Data, 1999.

[Knuth73] Donald Knuth, The Art of Computer Programming, Addison-Wesley, 1973.

[LiMoon01] Quanzhong Li, Bongki Moon: Indexing and Querying XML Data for Regular Path Expressions, *Proceedings of the 27th VLDB Conference*, pages 361–370, Roma, Italy, 2001.

[MiloSuciu99] Tova Milo and Dan Suciu: Index Structures for Path Expressions, 1999.

[Sleepycat] Sleepycat Software, Inc.: Documentation for Berkeley DB; accessed 29 November 2002
`http://www.sleepycat.com/docs/index.html`

[SGML] ISO (International Organization for Standardization). ISO 8879. Information Processing – Text and Office Systems – Standard Generalized Markup Language (SGML): International Organization for Standardization, 1986

[Vasudevan01] Venu Vasudevan: A Web Services Primer, 2001, O'Reilly & Associates, Inc.; accessed 23 October 2002
`http://www.xml.com/pub/a/2001/04/04/webservices/`

[W3C-WS] Word Wide Web Consortium: Web Services; accessed 18 October 2002
`http://www.w3.org/2002/ws/`

[XL01] Daniela Florescu, Andreas Grünhagen, Donald Kossmann. XL: An XML Programming Language. Technical Report, December 2001.

[XL02] Daniela Florescu, Andreas Grünhagen, Donald Kossmann. XL: An XML Programming Language for Web Service Specification and Composition. *WWW2002, International World Wide Web Conference*, Honolulu, HI, USA, May 7–11, 2002.

[XLink] World Wide Web Consortium: XML Linking Language (XLink) Version 1.0. W3C Recommendation 27 June 2001; accessed 16 October 2002
`http://www.w3.org/TR/xlink/`

[XML] World Wide Web Consortium: Extensible Markup Language (XML) 1.0 (Second Edition). W3C Recommendation 6 October 2000; accessed 1 October 2002
`http://www.w3.org/TR/REC-xml`

[XMLNS] World Wide Web Consortium: Namespaces in XML. W3C Recommendation 14 January 1999; accessed 13 October 2002
http://www.w3.org/TR/REC-xml-names/

[XMLSchema0] World Wide Web Consortium: XML Schema Part 0: Primer. W3C Recommendation, 2 May 2001; accessed 13 October 2002
http://www.w3.org/TR/xmlschema-0/

[XMLSchema1] World Wide Web Consortium: XML Schema Part 1: Structures. W3C Recommendation, 2 May 2001; accessed 13 October 2002
http://www.w3.org/TR/xmlschema-1/

[XMLSchema2] World Wide Web Consortium: XML Schema Part 2: Datatypes. W3C Recommendation, 2 May 2001; accessed 13 October 2002
http://www.w3.org/TR/xmlschema-2/

[XQuery] World Wide Web Consortium: XQuery 1.0: An XML Query Language. W3C Working Draft, 16 August 2002; accessed 18 October 2002
http://www.w3.org/TR/xquery/

[XSL] World Wide Web Consortium: Extensible Stylesheet Language (XSL) Version 1.0. W3C Recommendation 15 October 2001; accessed 16 October 2002
http://www.w3.org/TR/xsl/

[XSLT] World Wide Web Consortium: XSL Transformations (XSLT) Version 1.0. W3C Recommendation 16 November 1999; accessed 16 October 2002
http://www.w3.org/TR/xslt

[XPointer] World Wide Web Consortium: XML Pointer Language (XPointer). W3C Working Draft 16 August 2002; accessed 5 October 2002
http://www.w3.org/TR/xptr/