Technische Universität München
Institut für Informatik

Multidimensional Indexing and Querying of XML in Digital Libraries and Relational Database Systems

Michael G. Bauer

Institut für Informatik
der Technischen Universität München

Multidimensional Indexing and Querying of XML in Digital Libraries and Relational Database Systems

Michael G. Bauer

Vollständiger Abdruck der von der Fakultät für Informatik der Technischen Universität München zur Erlangung des akademischen Grades eines

Doktors der Naturwissenschaften

genehmigten Dissertation.

Vorsitzender:		Univ.-Prof. Dr. U. Baumgarten
Prüfer der Dissertation:		
	1.	Univ.-Prof. R. Bayer, Ph.D., em.
	2.	Univ.-Prof. Dr. G. Specht, Universität Ulm

Die Dissertation wurde am 24.6.2004 bei der Technischen Universität München eingereicht und durch die Fakultät für Informatik am 6.12.2004 angenommen.

Abstract:

The rise of XML (Extensible Markup Language) as a new universal data format created the demand to store XML with established relational database technology. This became even more important as application fields of relational databases like digital library systems also turned towards a massive use of XML. This work presents an approach to map XML data to a relational schema. The mapping uses an approach based on simplified Multidimensional Hierarchical Clustering (MHC) which was originally designed for data warehousing. On the one hand we analyze the performance of the developed mapping in the context of navigation in XML documents, on the other hand we also examine which indexing technique for the relational schema is especially powerful for typical XML queries. XML and the storage in relational database management systems are techniques which are utilized in the second part of this work which describes the digital library system OMNIS/2. The system allows to access several remote systems via one user interface, to work with the documents of the remote system (annotation and linking of documents), to upload user-defined XML documents and to store them in the system. OMNIS/2 offers this additional functionality without changing the original systems but instead extends the systems in a meta layer.

Zusammenfassung:

Der Aufstieg von XML (Extensible Markup Language) zu einem neuen universellen Datenformat erzeugte eine Nachfrage nach Speicherung von XML mit bestehender relationaler Datenbanktechnologie. Dies wurde auch dadurch verstärkt indem in Anwendungsfeldern relationaler Datenbanken, wie etwa Digitalen Bibliotheken, nach und nach verstärkt XML eingesetzt wurde. In dieser Arbeit wird ein Ansatz vorgestellt mit dem XML-Daten auf ein relationales Schema abgebildet werden können. Es wird einerseits untersucht wie geeignet die entwickelte Abbildung in Bezug auf Navigation in XML-Daten ist, andererseits wird ebenfalls untersucht welche Indexierung des relationalen Schemas bei typischen XML-Anfragen besonders leistungsfähig ist. XML und die Speicherung in relationalen Datenbanksystemen sind Techniken, die im zweiten Teil der Arbeit, dem Bibliothekssystem OMNIS/2, eingesetzt werden. Das System erlaubt es auf mehrere verschiedene entfernte Systeme unter einer Schnittstelle zugreifen zu können, mit den Dokumenten der entfernten Systeme zu arbeiten (z.B. Annotation und Verlinkung von Dokumenten) und benutzerdefinierte XML-Dokumente einzubringen und zu speichern. OMNIS/2 stellt diese zusätzliche Funktionalität bereit ohne die originalen Systeme zu verändern, stattdessen erweitert OMNIS/2 diese in einer Metaschicht.

Acknowledgements

Giving me the chance to write a dissertation was a big honour for me. Many thanks must go to my advisor Prof. Dr. Günther Specht for recruiting me and for giving me the opportunity to work in the interesting field of digital libraries. Many thanks must also go to Prof. Rudolf Bayer, Ph.D. for kindly giving me the chance to stay in München over the years although this was definitely not the apparent option. Throughout all these years I had a great time here! This was also due to my colleagues and ex-colleagues who created a nice environment and often made difficult work easier with pleasant and entertaining situations. They all had their tiny roles in making this possible. Among all colleagues I want to especially thank Dr. Frank Ramsak, who always found time for valuable and long, long discussions and even volunteered to proof-read major parts of this work, Dr. Roland Pieringer, who also read parts of the work and Prof. Dr. Thomas Erlebach, who read the mathematical part.

OMNIS/2 was a very small project, still it gave me the chance to work with skilled students who helped to improve the system and its concepts. With this I was pushed into the role of an advisor for theses and project work of others. It was a work which I enjoyed very much! From the long list of advised students two stand out: Johannes Altaner and Josef Gruber, who initially made me work on the topic of XML storage and XML indexing by insisting that I had to advise their work. OMNIS/2 also enabled me to get introduced into the scientific world. I highly appreciate that I was given the opportunity to visit conferences (mostly on my own) and meet many interesting people.

A dissertation can only reflect scientific work although it was also influenced by non-scientific life. Brothers, parents and friends were not directly participating in the dissertation but were nevertheless very important by giving moral support. Finally, there are people who initiated and supported all of this already in early years. They are no longer able to read this note but they would be very proud if they knew that they are mentioned here.

Contents

Chapter 1

Introduction

1.1 Introduction

The rise of XML (Extensible Markup Language) as a new universal data format created the demand to store XML with established relational database technology. This became even more important as application fields of relational databases like digital library systems also turned towards a massive use of XML. Often XML is seen as just another language to describe or display information and to make it exchangable between different communication partners. The actual target of XML is more complex though. XML can clearly be seen as a representation of semistructured data. The consequences of this are far ranging and bring up questions about the necessity of new data mangement systems. If new systems for the storage of semistructured data are to be developed their success does not only depend on the quality of the systems and the research. It also depends on whether existing systems are no longer capable to handle the new data. This is especially important as widely used applications like digital library systems build on established storage systems. In some application scenarios it is fundamental that either user-defined or harvested documents can be imported into a digital library system. A special group of systems in this context are meta systems for existing digital library systems. Meta systems for existing digital library systems enhance established systems. As they are placed on top of existing systems they allow annotations of existing documents and implement linking concepts which permit adding links to existing documents. They can also offer additional storage and indexing of user-defined multimedia documents and harvested documents (commonly in XML) from external systems.

The thesis is divided into two main parts. The first part presents an approach to store semistructured data in the form of XML in relational database management systems. After an introduction to semistructured data and a motivation of the problem we present a proposal for a classification scheme for XML queries (formulated in XQuery). With knowledge about typical query patterns we give the description of a mapping scheme for XML data to a relational schema, a proposal for fast navigation in stored XML documents and its performance in comparison with another mapping scheme for navigation. In the succeeding chapter we will then discuss multidimensional storage of XML documents based on our approach. We especially discuss different multidimensional indexing methods for our proposed relational mapping and give a detailed performance evaluation.

The second part of the thesis presents the digital libary system OMNIS/2. OMNIS/2 is a meta system for existing digital library system as described above. After an introduction to the architecture of OMNIS/2 and an overview on other systems with similar features we discuss the different components of the system. We describe the document model and how existing XML documents can be enhanced with links. We briefly show how this method can be also applied to PDF documents. One component of OMNIS/2 is an authoring tool which offers users to work with documents which should be processed by OMNIS/2. A final view on the components show how the system was extended towards the OpenArchives Initiative. We continue with a discussion of differences in exploring hypermedia systems and examine methods to ensure link integrity in hypermedia systems like OMNIS/2. In this context we show a similarity between hierarchical documents and ribonucleic acid (RNA). We end the work with a summary.

Although the thesis does contain related work on XML storage it does not give a detailed overview on XML and XML query languages. For detailed material we refer to the standards on XML and XQuery [XMLa, XQu] from the World Wide Web Consortium (W3C).

The thesis also does not contain a detailed discussion of multidimensional index structures. For further information we refer to other published work [Mar99, Ram02] where extensive discussions can be found. Instead we give a detailed discussion of the impacts of multidimensional index structures on our work.

1.2 XML and Relational Database Management Systems

Before talking about the storage of semistructured data in later chapters it is important to precisely understand properties and differences of semistructured data in contrast to commonly available strongly structured data which is commonly stored in Relational Database Management Systems (RDBMS). After a discussion why XML is a special form of semistructured data, we motivate the problem of storing XML in RDBMS in this introductory section.

1.2.1 The Nature of Semistructured Data

Semistructured data in contrast to strongly structured data is often seen to be schemaless (without a schema) or that the schema is frequently changed and highly dynamic. Sometimes semistructured data is also characterized to be self-describing. In addition it is usually considered to be not strictly typed, i.e. there are no (or not many) datatypes in use. These fundamental properties already show that semistructured data can be used very flexible but is not easily handled by machines. It is interesting to note that semistructured data was used already for a long time in computer science although people seem not to have been aware of it. This becomes quite clear when investigating a notation for semistructured data which is based on datatypes in the programming language Lisp.

A typical representation for such a form of data is a simple list where the different components are separated by commas and are grouped by using brackets [ABS99]. The components consist of pairs of labels and values, where the latter can also be label/value pairs. In the following we give two examples to describe instances of scientific papers.

Example 1.1
```
{ Author: {FirstName: "Michael", LastName: "Bauer"},
  Author: {FirstName: "Frank",  LastName: "Ramsak"},
  Title: "Multidimensional Mapping and Indexing of XML,
  Conference: "Business, Technology and Web",
  Date: February-2003}
```

Example 1.2
```
 {Title: Fast Query Processing,
  Author: Jones,
  Author: Smith,
  Date: 1999}
```

Both lists describe a scientific paper but there are several apparent differences. While in example 1.1 the **Author** is separated into two individual fields **FirstName, LastName**, in example 1.2 there is only one non separated **Author** field, the **Date** field contains a single number based value and it completly lacks a **Conference** field. A schema for the above data would have to be very flexible and the schema would have to be evolved when data occurs which does not fit.

The use of brackets already implies a hierarchy of the data so it is quite easy to transform the data and the relationships between the different fields into a graph. The notation is quite simple. Edges (annotated with labels) are connected from the root to the leaves whereas values reside in the leaves.

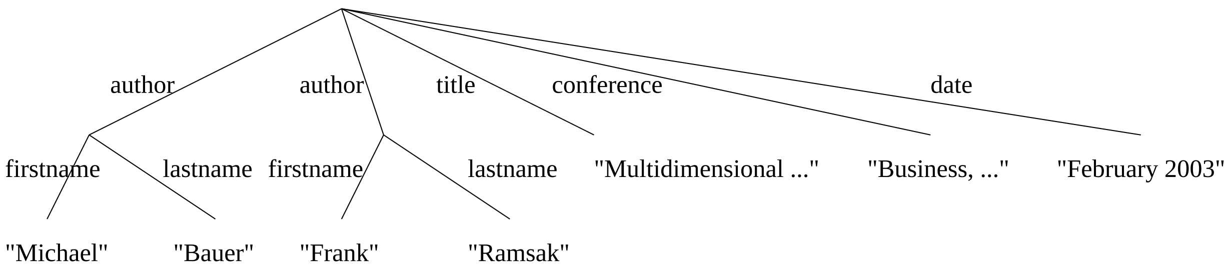


Figure 1.1: Semistructured Data shown as a Graph

Despite intensive research and other proposals (e.g. purely object oriented approaches) relational data is currently still the most popular organization of data in data management. In contrast to semistructured data, relational data is strongly structured, strongly typed and can easily be processed by machines. The basic elements in the relational world are sets and tuples. A variety of datatypes emerged over the years as demands for more complex applications grew.

For relational data the schema is strictly separated from the data. When storing relational data a schema is developed in an initial step which determines the datatypes. The actual data then is considered to be an instance of the schema.

A common way to display the instances of a relational schema (i.e. the data) are rows in a table.

Example 1.3 *relation 1:*

attribute1	*attribute2*	*attribute3*
attribute1a	*attribute2a*	*attribute 3a*
attribute1b	*attribute2b*	*attribute 3b*
attribute1c	*attribute2c*	*attribute 3c*

relation 2:

attribute1	*attribute4*
attribute1a	*attribute4a*
attribute1b	*attribute4b*
attribute1c	*attribute4c*

Interesting interpretations occur, when relational data is described by the list oriented notation we introduced above The relational data can then be rewritten as follows:

Example 1.4

```
{ relation1: {row: attribute1a, attribute2a, attribute 3a},
             {row: attribute1b, attribute2b, attribute 3b},
             {row: attribute1c, attribute2c, attribute 3c}
}

{ relation2: {row: attribute1a, attribute4a},
             {row: attribute1b, attribute4b},
             {row: attribute1c, attribute4c}
}
```

Changes to the data instances which add columns to a row are very difficult as they require a massive change in the schema. Not only the schema has to be changed, consequently also all other rows have to be modified by adding a NULL value for the newly added column. Making the schema very flexible in advance by adding possible columns leads to many NULL values contradicts the relational approach in general and also have to be handled carefully in SQL.

A Data Model for Semistructured Data

As semistructured data became more and more important in research there was also great need for a data model which is flexible and powerful enough to handle the lack of a schema and the reduced (or lack of) typing of data.

Due to its flexiblity which is derived from the lack of a schema and lack of datatypes, semistructured data became popular in the field of data exchange. Self-describing data is important in data exchange when heterogenious systems are involved. In heterogenious systems the used data formats are different in both, syntax and semantics. Especially when there is a strong difference in expressiveness, semistructured data plays an important role as it acts as a flexible container format which can be filled with data by all systems and can be exchanged easily.

Heterogenious data exchange was thoroughly investigated within the Tsimmis project at Stanford University [CGMH$^+$94]. Within the project the Object Exchange Model (OEM) [PGMW95] data model was developed especially for this purpose. In its original proposal an OEM object is defined by a tuple *(label, oid, type, value)*, where *label* is a variable-length character string, *oid* is the object's unique variable-length identifier, and *type* is the data type of the object's values. The *type* is either an *atom* type (e.g. integer, string, real number, etc.) or *complex*. When the type is *complex* then *value* is a set or list of oids. Thus OEM data can be represented as a graph where labels are attached to the nodes and not to the edges. Over time variants of OEM have been developed in several projects where labels are attached to edges or labels are even attached to both, edges and nodes. Due to its wide spread use OEM became the de facto data model for semistructured data.

We will come across OEM again in a later chapter of this work when we will discuss related work on XML mapping and indexing and also in related work on the later presented OMNIS/2 system.

1.2.2 XML as Semistructured Data

XML is widely seen as the lingua franca of the Internet. Originally designed to become the successor of HTML, XML has found its way into many unexpected parts of applications, ranging from simple formats for data exchange to protocols based on XML [SOA]. The roots of XML go back to the markup language SGML (Standard Generalized Markup Language) which has been developed for document processing. SGML though suffers from many flaws making automatic processing very difficult (e.g. ambiguities in grammars, etc.).

In general, XML is a textual representation of data. The basic building block of XML documents are so called *elements* which always have an opening construction (e.g. `<start>`), which is called a start-tag and a closing construction (e.g. `</start>`), which is called the end-tag. Tags are always enclosed in $<$ and $>$. Elements can contain either text or other elements. In cases where the elements do not have any content an element of the form `<start/>` can be used. An important constraint for the elements is that the tags are closed in inverse order to their opening, i.e. equivalent to parenthesis. XML documents which satisfy this property are called *well-formed* (see example 1.5 for an example).

Example 1.5
```
<paper>
 <title>
 XML as semistructured data
 </title>
</paper>
```

Additionally, XML allows the use of attributes. XML attributes are defined as pairs of (name, value) and can only be used in conjunction with tags (and not standalone). Attributes can hereby be seen as “properties of tags”. The values of attributes are always strings and there cannot be any substructure as with elements (example 1.6).

Example 1.6
```
<paper>
 <title style="bold" size="12pt">
 XML as semistructured data
 </title>
</paper>
```

Up to now, we have only discussed XML without a schema. In the XML recommendation from the W3C DTDs (Document Type Definition) are proposed to act as a kind of schema for XML documents. A DTD actually is a grammar for (a set of) XML documents and defines the occurrence and cardinality of elements, in which order they are required in the documents and what attributes are available for the elements. DTDs concentrate on structural properties of XML documents and not on data types. DTDs only distinguish between two data types: CDATA (character data) and PCDATA (parsed character data). When XML documents follow a DTD they are declared to be *valid*.

XML is commonly seen as a representation of semistructured data due to the flexibility of XML, the lack of strict typing and its data model. The above mentioned constraint for well-formed XML documents ensures that the documents can be parsed into a tree. This tree though is different from the tree we have shown for semistructured data. A sample tree is shown in Figure 1.2.

Apparent differences are that label/tag information is now located in the nodes and not on the edges.

People who work with XML quite often do not recognize the importance of order in XML documents. Example 1.7 shows two XML `paper` fragments which contain the same data but in different order. According to the XML recommendation the two fragments have to be treated as different documents.

Example 1.7
```
<paper>
<author>Smith</author>
<title>Fast Query Processing</title>
```

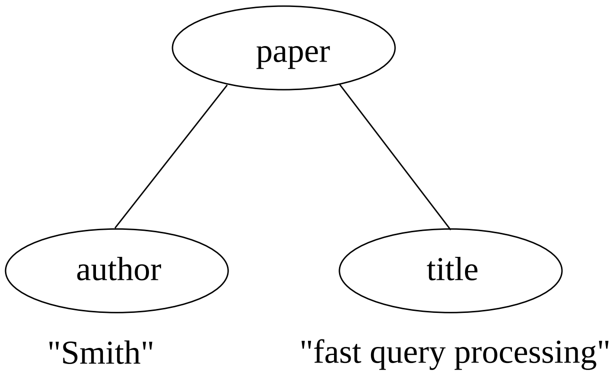


Figure 1.2: Sample tree of the XML document model

```
</paper>

<paper>
<title>Fast Query Processing</title>
<author>Smith</author>
</paper>
```

For attributes the situation is different and the two fragments in example 1.8 are the same from the XML point of view. This difference has its background in the culture of document processing and the SGML tradition within XML.

Example 1.8

```
<paper>
 <title size="12pt" style="bold">
 XML as semistructured data
 </title>
</paper>

<paper>
 <title style="bold" size="12pt">
 XML as semistructured data
 </title>
</paper>
```

XML files are often classified whether they are data-centric (Example 1.9) or document-centric (Example 1.10). While the characteristics are often used quite liberal it still can be said that data-centric documents usually carry their data in the leafs of the XML tree and a more structured, while document-centric XML uses a lot of mixed content (data not in the leaves) and free text. Document-centric files are usually generated in document processing while data-centric files occur in data exchange and can be easily generated from relational data.

Example 1.9
```
<meeting>
<date>2003-08-18</date>
<location>Trondheim, Norway</location>
<purpose>conference</purpose>
</meeting>
```

Example 1.10
```
<meeting>
Please be sure to visit the <purpose>conference</purpose>
in <location>Trondheim, Norway</location> on <date>2003-08-18</date>
<meeting>
```

1.2.3 The Problem: Storing XML in RDBMSs

Storing XML in RDBMS is a challenging problem as loosely structured data has to match with a system which was designed to store strongly structured data. Many approaches have been made to store XML in relational database systems. All approaches use a similar concept though. First an XML document is split into parts of a previously defined granularity. These parts are stored in the RDBMS. Queries on XML documents which are written in an XML query language have to be rewritten to SQL before being processed by the RDBMS. The results of the SQL queries are XML documents (selection phase). The extraction of relevant fragments (elements, subtrees, etc.) of these XML documents is either done via methods like XSLT (Extensible Stylesheet Language for Transformations) or the results are assembled directly from the database. This phase is a projection phase. Our approach to XML storage, which is presented later, takes care of both cases, selection and projection.

Besides storing XML data, querying large amounts of XML data is also a challenging problem. Due to its graph-like nature the classical set oriented query languages in general are not powerful enough. After several proposals for query languages (mainly from the fast evolving field of semistrucutred data) the W3C started a working group to formulate a query language for XML. The current proposal XQuery though is not yet a recommendation of the W3C as of this writing.

We have identified two fundamental parts of a query for XML. Consider the following XQuery statement which should retrieve the value of a tag (**tag1**) in a document containing **tag2** with value ABC:

```
for $b in document ("xmldoc.xml")//document
where $b/tag2 = "ABC"
return <result>
       $b/tag1
       </result>
```

This query can be split up into two steps. The selection part identifies the document(s) which contain(s) the pattern matching a predicate s (in the above case `$b/tag2 = "ABC"`). The projection part in contrast returns only those part(s) of the document(s) which are found after the **return** statement as a set of paths. We are aware that the XQuery language is obviously more powerful but in this work we focus on these two fundamental steps of XQuery processing.

The mentioned problems can be defined more formally:

Definition 1 (selection problem) *Let X be a set of XML documents stored in a database where each document is described by a persistant unique identifier Id. The selection problem is defined as returning Id for those documents where the predicate s from the query is evaluated to true.*

Definition 2 (projection problem) *Let Id be a set of persistant unique identifiers and L be a list of path expressions in a projection list. The projection problem is defined as returning the content of those paths from the documents referenced by Id that fulfill the expressions in L.*

When talking about the projection problem we sometimes use the term "reconstruction of (parts of) the document". We refer to the fact that the desired document is completely or partially assembled from the database into its original state.

In our approach we do not tackle only one of the above mentioned problems (either selection or projection) but both.

One has to be aware of the fact that the projection in the case of XML documents is fundamentally different from the relational world. The operations in the relational algebra deal with tuples and sets as the basic units. Tuples in RDBMSs are typically small compared to the size of an XML document, therefore tuples are normally retrieved as a whole during the selection process and the projection always processes the tuples that result from the selection. For efficiently answering queries in the relational world it is therefore sufficient to only speed-up the selection. For XML documents the scenario is slightly different as the granularity of an XML document can be seen on different levels. A very rough granularity is based on documents (which are identified by a persistent unique identifier). Tags as the building block of XML documents are another level of granularity; a very fine granularity would be based on words or letters. Depending on the storage of XML in the RDBMS the projection works on a completely different position of the XML document than the selection. The consequence is that an index that is suitable for the selection is rarely suitable to speed up projection as well. We will see later that selection and projection can even be orthogonal and require very specialized index support.

1.3 Other Important Approaches for XML Storage

Storing, indexing and retrieving XML in database systems received a lot of attention in research over the last years. The very high number of different approaches omits a detailed discussion of every single contribution. Nevertheless, we give an overview in this chapter which is split into three parts: We will further discuss relational mappings for XML, special indexing methods which were developed for XML data and finally we will take a look on commercial systems which offer storage of XML data.

1.3.1 Relational Mappings

In this section we will present some important approaches which were developed to store XML data in relational database systems. For the sake of readability we will only concentrate on some approaches to show different methods of shredding XML and to explain methods which are used for comparison purposes in the literature. There are numberless other approaches. Among them is the storage of XML as BLOBs (Binary Large Objects) or CLOBs (Character Large Objects). We consider such approaches as trivial and will omit discussing them. Instead we discuss five proposals of which two (Edge Mapping and Pre/Post-Mapping) are used later in our performance evaluations.

1. Edge Mapping

Since XML data can be represented as a graph, typical methods for graph storage are applicable. In such an approach the XML graph is split into basically three components (*parentid*, *childid*, and *label/value*). These components are then stored in an *edge* relation in the database. The *edge* table contains a tuple for every nesting relationship. Non leaf nodes contain the id of the parent node in the *parentid* attribute and the id of the child node in the *childid* attribute. The *label* contains the tag. Tuples for leaf nodes (data elements) carry a NULL value in the *childid* attribute and the data value in the *label* attribute. The approach is especially convenient, when the documents are not provided with a schema or a DTD.

Ensuring the correct document order can be either achieved by adding an additional order attribute (e.g. an integer attribute) or by implicitly storing the order in the *parentid* and *childid* attributes.

When storing multiple documents with Edge mapping it is convenient to store the root node in a separate table *roots(id, label).* The *roots* table contains a tuple for every document with an *id* which identifies the document (document identifier) and *label* for the root tag.

Example 1.11 *The fragment*

```
<paper><title>Fast Query Processing</title></paper>
```

is stored with the tupel (0, paper) in the roots table and with the tuples (0,1, title) and (1, NULL, 'Fast Query Processing') in the edge table.

roots:

id	*label*
0	paper

edge:

parentid	*childid*	*label/value*
0	1	title
1	NULL	Fast Query Processing

This approach was named Edge Mapping and is originally discussed in detail by Florescu and Kossmann [FK99]. The work was very influential and it became a defacto standard for benchmarking other relational approaches for XML storage. The above mentioned variants of the Edge Mapping are further discussed in the work on the Indexfabric [CSF+01] and on order in XML documents [TVB+02].

2. Shared

Another approach to store XML in RDBMS is published by Shanmugasundaram, et.al [STZ+99]. Their approach stores XML documents in relations based on the DTD. Shanmugasundaram, et.al propose different versions of their approach but in general they suggest relations for elements with inlined child elements and separate relations for all elements carrying a Kleene star (including their subelements as inlined). In detail the starting point for their algorithm is always a DTD, which is transformed into a graph which represents the structure of the DTD (DTD graph). Each node in the DTD graph represents an XML element, an XML attribute or a Kleene star (operator). To transform the graph into a relational schema the graph is traversed. At first a relation for the root element is generated. All children of an element are represented inline in the relation of the element except if the child is a Kleene star. In this case the child of the Kleene star node is represented in a separate relation. All created relations have an id field which acts as a primary key. All non root elements also have a parentid field which is a foreign key reference to its parent to link a child element to its parent. To preserve order of an element among its siblings each relation for a non-root element also has an order attribute. Example 1.12 clarifies the method [SSK+01]:

Example 1.12 *A DTD is given as follows:*

```
<!ELEMENT PurchaseOrder (ItemsBought, Payments)>
<!ATTLIST PurchaseOrder BuyerName CDATA #REQUIRED
                        Date CDATA #REQUIRED

<!ELEMENT ItemsBought (Item)*>
```

```
<!ELEMENT Item EMPTY>
<!ATTLIST Item PartId CDATA #REQUIRED
               Cost CDATA #REQUIRED>

<!ELEMENT Payments (Payment)*>

<!ELEMENT Payment EMPTY>
<!ATTLIST Payment CreditCard CDATA #REQUIRED
                  ChargeAmt CDATA #REQUIRED>
```

The resulting relational schema then looks as follows (primary keys are underlined, ParentId is a foreign key to PurchaseOrder.Id):

```
PurchaseOrder (Id, BuyerName, Date)
Item (Id, ParentId, Order, ParID, Cost)
Payment (Id, ParentId, Order, CreditCard, ChargeAmt)
```

3. STORED

In cases where a schema is not known in advance or not considered there are approaches to directly extract a schema from available semistructured data which includes XML as an instance of semistructured data. Such research was performed in the STORED project [DFS99]. The extraction process results in a relational schema and overflow tables. The relational schema is generated with the help of a datamining algorithm for semistructured data. As the used algorithm of Wang and Li [WL98] uses heuristics for the generation of a relational schema it can happen that for some data no schema is available. The same happens if data should be inserted into the relational schema which was not available in the generation stage. In such cases the data is stored in semistructured repository which uses overflow tables. Overflow tables store the graph representation of the semistructured data in an unary relation which carries the root of a graph and a ternary relation which store a source and a target identifier and the attribute name similar to the method of Edge Mapping.

4. Mappings for ordered access

Some proposals for storage of XML data neglect the order of paths in XML documents. The importance of order is discussed in detail in [TVB+02]. The work examines three different mapping approaches and their suitability regarding order. The first proposal is a Global Order approach where every node in the XML document is assigned a number according to its absolute position in the XML document. The second proposal is a Local Order (or Sibling Order) approach where every node is assigned a number which represents its relative position among its siblings. A combination of the number of a node and its ancestor then gives an absolute position but this has to calculated at run-time. Their third proposal is Dewey Order which is a combination of the two other encoding approaches.

Dewey Order is originally used for the classification of library items and is very similar to our MHC technique (Multidimensional Hierarchical Clustering), which we present later. The implementation of their encoding scheme is significantly different from ours as it is based on UTF-8 encoded strings. The work also investigates the processing of some XPath and XQuery statements and also the performance of the mappings for insertion of XML elements. The authors conclude from their work that Dewey Order is the best compromise for queries and updates but that current relational query optimizers do not understand the hierachical structure of XML and choose poor query plans.

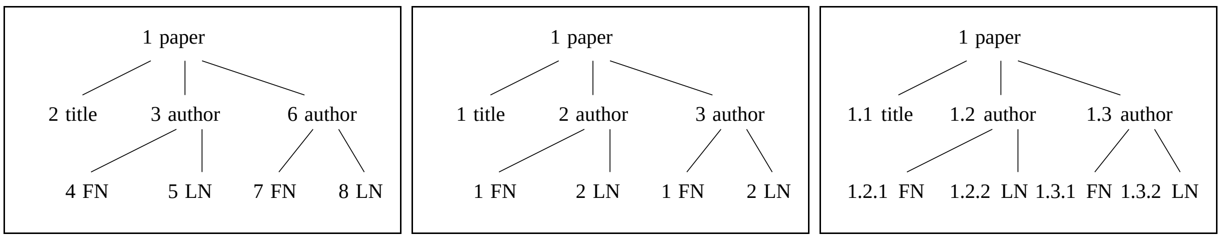


Figure 1.3: Document tree with Global Order, Local Order, and Dewey Order (from left to right)

5. Pre/Post-Mapping for XPath Location Steps

Grust [Gru02] recently proposed a numbering scheme to speed up the processing of XPath location steps. The XPath recommendation [XPa] of the W3C states that the four axes `preceding`, `descendant`, `ancestor`, and `following` partition the document into four disjoint regions. By adding the context node the complete document is covered. This observation is used in the mapping where each node of the XML document is labeled with two values *pre/post*. The *pre* value is hereby calculated by traversing the XML document tree in preorder, i.e. a node is visited and assigned its pre value before its children are recursively traversed from left to right. The *post* value is calculated by traversing the tree in post order, i.e. a node is visited and assigned its post value after all its children are recursively traversed from left to right. Figure 1.4 shows the *pre/post* values for the sample XML document below. For every tree node the left number indicates the pre value and the right number indicates the post value of the node. As the mapping is especially designed for navigation we omit node content at this point. The pre/post values though can be used to store content in a relational table using a foreign key relation.

```
<a>
  <b>
    <c></c> <d></d>
  </b>
  <e></e>
  <f></f>
```

```
</a>
```

This mapping ensures that the above mentioned axes are also mapped to regions in a two dimensional universe as can be seen in Figure 1.5. The four regions (I.-IV.) in Figure 1.5 exactly correspond to the above mentioned four axes. The `ancestor` axis is equivalent to the upper-left region of the context node (I.), `following` is equivalent to the upper-right region (II.), `preceding` to the lower-left region (III.), and `descendant` to the lower-right region (IV.).

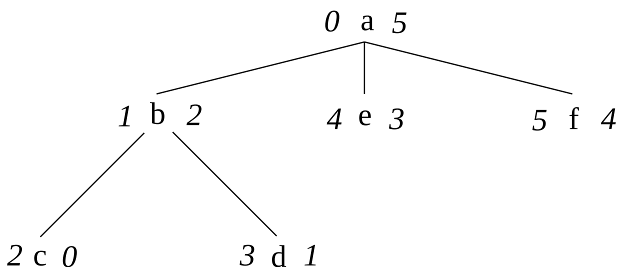


Figure 1.4: Document tree of the sample document with *pre/post* values

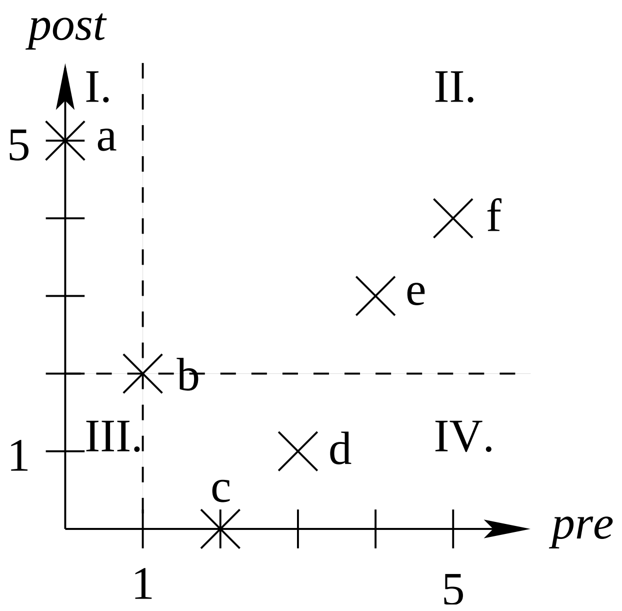


Figure 1.5: Graphical display of *pre/post* values

To be able to handle XML attributes an XML element like the following

```
<a b="" c="">
  <d></d>
</a>
```

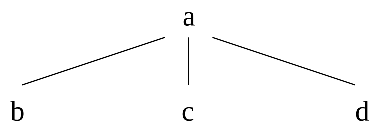


Figure 1.6: Document tree for attributes

is treated like the document tree in Figure 1.6 (*b* and *c* are attributes). This is sound with the XML document order semantics as attributes of a node appear before its children.

When storing the mapping in a relational table each tuple represents a document node. The required information is hereby stored in a table with five attributes. Two attributes are reserved for the *pre/post* values. As some axes require knowledge about the parent node of the context node a third attribute is used to explicitly store the parent of the node. A fourth attribute stores the name of the tag of the node so that name tests can be performed on the node and the fifth attribute indicates whether the node is an element or an attribute. Content in the document can be stored in a separate table which contains either pre or post values as a foreign key.

The mapping scheme is independent of a special underlying index but performance is heavily influenced. In the original paper of Grust a configuration of B-Trees with a separate index on *pre* (clustering index) and on *post*. Later Grust [Gru] proposed a different solution with indexes `(tag, pre, flag, par, post)` and `(tag, post, pre)`. This reveals that nametests on tags are considered to be very selective (which is often the case but depends on the usage of tags in the XML document). Of course restrictions without a nametest are highly problematic as they are not index supported.

The mapping especially targets navigation in XML documents and neglects content of XML documents. For navigation alone the presented benchmarks show that the mapping is superior to Edge Mapping, which in contrast does consider values.

1.3.2 Indexing Methods

Besides storing XML data in relational database systems, there is significant work to develop special index structures to especially support path queries on XML and semistructured data in general. Most of theses index structures for XML and semistructured data have in common that they concentrate on the efficient encoding and indexing of paths. In this section we concentrate on three approaches for indexing semistructured data which are either based on the enhancement of an established idea (Indexfabric) or have a strong theoretical foundation (Dataguides and T-Indexes). A valuable survey on indexing techniques for semistructured documents can be found in [Wei02]. All presented indexing methods are targeted towards graph databases and are normally not suitable for approaches based on relational mappings.

1. The Indexfabric

Cooper, et al.[CSF$^+$01] present a solution for XML indexing based on fast text enconding using patricia tries, which they evolved into a balanced index structure for secondary storage (s.c. *Indexfabric*).

They motivate their work with the fact that conventional B-Trees do not scale well for large keys which have identical long prefixes (as paths in XML documents usually do have). With their proposal of using designators they shorten each path to short strings (i.e. `/book/title` becomes $/\beta/\tau$), which they then store with a modified patricia trie approach. The mapping of designators to paths is stored separately and in contrast to other approaches the Indexfabric also indexes textual content when it is reached by a path. The single characters of the textual content are added to the path string of designators (i.e. `/book/title` *databases* is rewritten to $/\beta/\tau/d/a/t/a/b/a/s/e/s$). The Indexfabric mainly concentrates on paths from the root node to leaves. An extension is proposed though which also indexes arbitrary paths but the paths have to be chosen in advance before creating the index. From the presented benchmarks [CSF$^+$01] the *Indexfabric* is clearly superior to Edge Mapping and STORED. In the available work, order of XML documents is not taken care of although the work claims that a solution is available.

2. DataGuides

Widom, et al. have published significant work on systems storing semistructured data, XML and query languages [MAG$^+$97, AQM$^+$97].

Especially important in this context is the work on Dataguides and the Lore System. Widom, et al. propose DataGuides as dynamic schemas for semistructured data. DataGuides should help for browsing a database structure, formulating queries, storing statistics and sample values, and enable query optimization. DataGuides are based on the OEM model and summarize the structure of it. For every label path of the OEM model (independently how often it is contained in the OEM model) exactly one data path occurs in the DataGuide. A DataGuide also does not contain a path which does not occur in the original OEM model. The creation of a DataGuide for a OEM model (which can be seen as a non-deterministic finite automaton) is equivalent to the conversion of a non-deterministic finite automaton (NFA) to a deterministic finite automaton (DFA). The consequence from this is that a DataGuide might be larger than the original OEM model and might require exponentional space for the number of objects and edges in the source OEM model.

Both, the OEM model and DataGuides are used in the Lore system, a database system for semistructured data [MAG$^+$97].

3. T-Index

T-Indexes (Template Indexes) [MS99] describe a family of path indexes for semistructured data. The idea behind it is to create indexes which use equivalence classes for the nodes of the semistructured data model. The contents of an equivalence class are nodes which are indistinguishable according to a class of paths. The path classes are given by path templates for semistructured data. Path templates can be more complex than simple path expressions for semistructured data as the former also allows the use of regular expressions in the template. Path templates are expressions of the form $T_0x_0 \dots T_kx_k$, where x_i are variables which are bound to document nodes specified by T_i. Simplified T_i is either a constant regular path expression or a path expression with boolean formulas (e.g. label tests). Simple forms of T-Indexes are 1-Indexes and 2-Indexes, which use a very restricted form of path templates. 1-Indexes only index all absolute paths, while 2-Indexes also cover relative paths. When tree-like graphs are indexed by a 1-Index then it is equivalent to a Dataguide. T-Indexes use path templates of an arbitrary but predefined length and are created for a set of privileged paths. Privileged paths are paths for which the index is very suitable. Due to this construction it can happen that some queries cannot be processed with a certain T-Index. In some cases though the query can be rewritten so that a T-Index can still be used.

Due to the relaxation of the equivalence class criteria (which leads to non-determinism), T-Indexes are generalizations of Dataguides and also Patricia Tries.

1.3.3 Commercial Systems

Many commercial database vendors started to build systems for storage and processing of XML. There are mainly two separate approaches. One group of vendors claims to store XML natively in newly designed systems while the other group stores XML in established systems. The claim about storing XML natively is very deceiving as XML is in most systems not stored natively in the original text form but it is always stored with the help of an underlying data model. What is probably meant by the vendors is that XML is not mapped to an existing data model (relational, object-oriented, etc.) but that it is stored in an own format which preserves the special features of XML (hierarchy, semistructured). In the following we only give an overview of some systems. A more complete list about XML and databases can be found in the list of Ronald Bourret [XMLb]. As representatives of systems which claim to store XML natively we list *Tamino* and *eXcelon* and as representatives of (object-)relational systems *Oracle*, *DB2 Universal Database* and *Microsoft SQL Server 2000*. It is important to note that the support of XML might significantly change with future versions. Whenever possible and provided by the vendor the relevant version number is given.

1. Tamino

Tamino [Tam] is a database server especially designed to store XML. It is being developed by the German company Software AG, which claims that Tamino stores XML natively.

Tamino offers two storage methods for XML, a schema based storage approach and a storage approach without a schema. In the schema based approach the DTD or available XSchema is used to generate a storage model for the XML documents which comply with the schema information. When a schema is not available then the XML data is stored using a fulltext index.

Tamino is designed so that all requests are sent via HTTP to the database. This also shows that Tamino is targeted on a heterogenous scenario with multiple data sources and that it is not supposed to replace all existing systems.

While the first versions of Tamino offered XQL (which uses parts of XPath) as the query language on the stored XML data, Tamino is now going towards the XQuery language. As of this writing XQuery is only available as demo for Tamino as XQuery is not yet a recommendation of W3C. The query language currently used is Tamino X-Query, which is also based on XPath but must not be mixed with XQuery from the W3C.

2. eXcelon

Another system which claims to store XML natively is eXcelon [XIS]. The referenced literature deals with version 3.0 of the *Exensible Information Server*. As the system is designed to be an information server it goes beyond storage of XML. The Extensible Information Server instead offers a distributed, scalable solution for managing XML data and content. It also offers transactional access to the content and transformation capabilities. Internally the information server uses a especially designed information model based on XML. XML data is stored in a preparsed format mainly to eliminate any overhead which occurs with parsing on demand (which seems important when documents are queried). There are no further details given on how the information model is implemented. The publication claims though that three different index types are supported: value indexes for string and numeric searches, text indexes for word-based searches, and structural indexes for structure-based searches. The information server offers different access interfaces to the stored data e.g. a DOM interface and a filesystem interface. Queries on one or several XML documents can be formulated in XPath expressions. The server also includes an XSLT processor which allows to transform XML content into HTML, WML or other required formats. Overall eXcelon offers a system which targets at offering a complete solution for the managing of XML.

3. Oracle

Oracle is originally an object-relational database system but starting with version 7 it was decided to also integrate support for XML data into the system [Sch03].

Besides the simple approaches to store XML as a BLOB/CLOB (where indexing is performed with full-text indexes) Oracle offers an object-relational approach to store XML by using s.c. annotated XML Schemas. The standardized XML Schema is hereby enriched with additional information about corresponding datatypes and storage information in the ORDBMS (Object Relational Database Management System) which means that XML is mapped by hand. This information is given in a separate XML namespace so that annotated XML Schemas comply with the standard and are not a proprietary extension.

Oracle offers several ways to access the stored XML documents. It can either be done via protocols like FTP, HTTP or WEBDAV (Web Distributed Authoring and Versioning) or via SQL. The protocol based approach treats the database like a repository and even enables access to parts of the documents by using an adressing scheme based on XPath. The XPath statement is hereby embedded into a URI which used to access the database. To be able to deal with the special requirements of XML it was necessary to enhance SQL by a feature called "piecewise update", which allows to update only parts of the XML document. In addition Oracle supports SQL/XML which introduces new functions and operators. It is planned to integrate XQuery into the Oracle DBMS. Until XQuery becomes a recomendation of the W3C, Oracle offers a prototype implementation of XQuery for download for interested users.

4. DB2 Universal Database

In DB2 Universal Database the handling of XML documents is provided by the toolkit *XML Extender* [CX00]. The XML Extender enables both, the storage of XML documents and the generation of XML from already stored conventional data. Storing XML in DB2 can be done in two different ways. It can be stored either as an XML column or as an XML collection. For storage as an XML column the XML Extender provides three different user-defined data types (UDTs): XMLCLOB, XMLVARCHAR, and XMLFile. XMLCLOB and XMLVARCHAR are the classical methods to store XML data as CLOB or VARCHAR. As XMLFile the XML data resides on a local file system and only a pointer (URL) is stored in DB2. Parts of the XML file can be mapped to tables (side tables) so that some elements and attribute can be indexed. This is determined by a *Data Access Definition (DAD)*, which is an XML file itself and describes mapping and indexing details.

In contrast an XML collection is a set of relational tables which contains the data from the XML file. In this context a DAD is used to define the mapping between a DTD and relational tables and columns. Besides storage (decomposition of XML) an XML collection can also be used to map relational data stored in tables to XML documents (composition of XML).

For both storage and access methods DTDs are stored in a special DTD repository table. Access to the stored XML data is provided by special UDFs (User-defined functions) and stored procedures. When data is decomposed into an XML collection DB2 SQL can be used to select, update, and delete XML fragments. Addressing parts of XML documents is done by supporting a subset of XPath. DB2 does not support SQL/XML (as of this writing).

XML columns and XML collections allow further indexing of the stored data by using DB2 Text Extender which is a set of tools provided for the handling of large text documents in the database.

5. Microsoft SQLServer 2000

Starting with SQL Server 2000 the database product of Microsoft supports XML. XML support is part of a larger effort of Microsoft and shows up in many different products.

The storage of XML in SQL Server is offered by means of the OpenXML rowset provider. OpenXML provides two possible ways to create relational views over XML data. One approach is called *edge table view*, the other one *shredded rowset view*. The edge table view stores the parent-child hierarchy and all other information about each node. This approach is equivalent to the formerly presented Edge Mapping (section 1.3.1). The shredded rowset view uses an XPath expression (s.c. row pattern) to identify nodes in the XML document which will then be mapped to rows in the database. A relative XPath expression (the column pattern) is used to identify the nodes which provide the values for each column. Columns can also contain metadata of the XML document like node ids, parent node ids, etc., which allows broad support for different features of XML.

Generating XML from existing relational data in the database is offered by creating XML views on relational data. XML views are created using annotated XML schemas. The annotated schemas use XML Data Reduced (XDR) schemas which allow to embedd the annotation into the XML document of the schema. In the default mapping every relational column is mapped to an XML attribute of the same name, all other elements map into a row of a table. User-defined mappings use a special name space. This only generates the schema without any data. The data is then retrieved by querying the XML view with a subset of XPath.

XML can also be generated directly from SQL query results without an XML view. A new keyword in SQL maps rows to elements and column values to attributes. SQL queries can also be embedded into templates which allows the creation of arbitrary result XML documents.

Chapter 2

A Classification of XML Queries

2.1 About the XQuery Classification Scheme

Prior to presenting our techniques for storing XML data in RDBMS it is important to discuss the nature of queries on XML. It is common knowledge by now that storage and especially indexing techniques result in a high performance for certain classes of queries while other classes are not supported as well. Identifying these classes of queries in the context of our later presented multidimensional storage approach results in a deeper understanding of queries on XML but is also challenging as XQuery is known to be very powerful. This is already due to the fact that the underlying data model of XML is graph oriented which requires a more sophisticated query language than for example the relational data model.

A classification which represents the different types of queries as it will be developed in this section usually consists of several different classes and enables insights into fundamental issues and properties of the query language. Due to the complexity of XQuery the classification though is not canonical in a way that a query can be assigned to exactly one class. This consequently also leads to the problem that the presented classes are not disjunctive and that a query can strictly speaking belong to several classes. Additionally some classes are further divided into subclasses. Often there are several different types of queries which can be grouped together in a more general class.

The basis for our classification scheme is XQuery 1.0 [XQu]. The term *query* is used in this chapter as a synonym for queries formulated in XQuery according to the above mentioned reference. All other used terms (e.g. *element node*, *text node*, *attribute node*, and *sequence*) conform to their definition from the XQuery draft.

A short overview on our proposed classes can be found in Table 2.1, where each class is listed with a very short description.

To the best of the author's knowledge this is the first approach to classify queries formulated in XQuery. The basic motivation for this chapter came from the book *Database Tuning* [SB02]. The book contains a very rudamentary classification of relational queries in the context of index tuning.

Class	Description
Path Traversal	Navigating and locating structures in XML
Point- and Rangequeries	Querying XML on single nodes or sequences
Predicates - Value, Position, Structure	Querying XML on values, position, and structure
Join	Combining elements from separate XML documents
Text Search	Searching for a string in a XML document
Sorting	Changing the order of a sequence of nodes
Quantification	Two quantifiers test the validity of a predicate
Operations on Sets	Union, difference, and intersection on nodes
Application of Functions	Application of built-in and user-defined functions
Construction	Assembling new documents from existing documents or query results

Table 2.1: Complete List of Classes for XML Queries

The following sections present the developed classes. The examples use two sample documents. The first document is part of the freely available *Xerces XML parser* [Xer] and is the basis for most examples.

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE personnel SYSTEM "personal.dtd">
<personnel>

  <person id="Big.Boss">
    <name><family>Boss</family> <given>Big</given></name>
    <email>chief@foo.com</email>
    <link subordinates="one.worker two.worker three.worker four.worker
                        five.worker"/>
  </person>

  <person id="one.worker">
    <name><family>Worker</family> <given>One</given></name>
    <email>one@foo.com</email>
    <link manager="Big.Boss"/>
  </person>

  <person id="two.worker">
    <name><family>Worker</family> <given>Two</given></name>
    <email>two@foo.com</email>
    <link manager="Big.Boss"/>
  </person>

  <person id="three.worker">
    <name><family>Worker</family> <given>Three</given></name>
    <email>three@foo.com</email>
    <link manager="Big.Boss"/>
  </person>

  <person id="four.worker">
    <name><family>Worker</family> <given>Four</given></name>
    <email>four@foo.com</email>
    <link manager="Big.Boss"/>
  </person>

  <person id="five.worker">
    <name><family>Worker</family> <given>Five</given></name>
    <email>five@foo.com</email>
    <link manager="Big.Boss"/>
  </person>

</personnel>
```

One example in Section 2.2.3 is based on the following document due to its recursive nature. The document is a shortened version from the *XQuery Use Cases* [XQu].

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE personnel SYSTEM "parttree.dtd">
<parttree>
    <part partid="0" name="car">
        <part partid="1" name="engine">
            <part partid="3" name="piston"/>
        </part>
    </part>
</parttree>
```

2.2 Path Traversal

As XML documents are hierarchically structured, traversal of a document by using path expressions is a fundamental building block of almost every XQuery expression. This importance will be again reflected in later chapters when we further discuss navigation. In the following we propose three different subclasses which separately describe a characteristic application of path traversal. The three subclasses are named Traversal, Relation between nodes, and Recursive Elements.

2.2.1 Traversal

Navigation in a document is supported in XQuery by using path expressions. As every document is hierarchically structured this variant of a query is always applicable whenever one or more nodes appear in the document.

Example 2.1 *This example presents a traversal by using a path expression.*

```
document("personal.xml")/personnel/person/name/given
```

The result is a sequence of element nodes:

```
<given>Big</given>
<given>One</given>
<given>Two</given>
<given>Three</given>
<given>Four</given>
<given>Five</given>
```

2.2.2 Relationship between Nodes

In an XML document all elements (except the root element) have an ancestor. The parent-son relationship can not only be accessed in one direction (top-down or bottom-up) but in both directions, which enables random access to nodes which are located "further away" from the context node (seen from a edge oriented point of view, e.g. siblings of the context node). This property is characteristic for hierarchical structures and is supported by XQuery with the help of location steps which can be arbitrarily combined.

Example 2.2 *In this example the query returns all `family` element nodes with a value larger than the value of the `given` element node.*

```
for $x in document("personal.xml")/personnel/person/name/family
where $x > $x/../given
return $x
```

Result:

```
<family>Boss</family>
<family>Worker</family>
<family>Worker</family>
<family>Worker</family>
<family>Worker</family>
<family>Worker</family>
```

2.2.3 Recursive Elements

The third subclass is formed by components which allow the access of recursive elements. This can be done with traversal in combination with an exact localisation of an element node. More formally this is described as an element node which is contained in itself with a depth larger than two. Structural recursion is required to create a document of arbitrary depth.

Example 2.3 *This example describes the search for a recursive element node.*

```
document("parttree.xml")/parttree//part
```

The result are a element node which occur recursivly withing the structure.

```
<part partid = "0" name = "car">
      <part partid = "1" name = "engine" >
              <part partid = "3" name = "piston"/>
      </part>
</part>
<part partid = "1" name = "engine">
      <part partid = "3" name = "piston"/>
</part>
<part partid = "3" name = "piston"/>
```

2.3 Point- and Rangequeries

Another class, which is also important for later, is formed by point- and rangequeries. In the relational world these two kinds of queries are tightly linked with the term "dimension". In XML it is difficult to find a semantically equivalent term so in this context a point query restricts on exactly one value and is one of the fundamental building blocks of XQuery. The value can be either restricted in a text or attribute node. Extending a point query to e.g. several attributes inside an element node results in querying a combination of values.

For a range query a range of nodes (out of a sequence of nodes) is expressed by stating boundaries with values. An extension to several range predicates leads to a combination analog to the case of point queries (here: several range queries).

This leads to classification where point and range queries also classified according to whether they are single or multiple queries.

2.3.1 Simple Point Query (including arithmetic expressions)

A simple point query is restricted to a comparison of a value (according to XQuery 1.0). When using numerical values it is possible to formulate a query using an arithmetic expression. Due to its simplicity this query can be formulated for every XML document which consists of at least one attribute or text node.

Example 2.4 *This example returns all element nodes `name`, which contain the string `Worker` in the child node `family`. In this version no arithmetic expressions are used.*

```
for $x in document("personal.xml")//name
where $x/family = "Worker"
return $x
```

Result:

```
<name>
  <family>Worker</family>
  <given>One</given>
</name>
.
.
.
<name>
  <family>Worker</family>
  <given>Four</given>
</name>
<name>
  <family>Worker</family>
  <given>Five</given>
</name>
```

2.3.2 Point Query with a Combination of Values (using logical expressions)

Extending a point query to several attribute or text nodes within an element node results in a query with a combination of values. The single predicates are combined with the logical operation "`and`" or "`or`".

Example 2.5 *This example returns all `name` element nodes which have two child nodes. One child node should contain the string `Worker`, the second one the string `One`.*

```
for $x in document("personal.xml")//name
where ($x/family = "Worker" and $x/given = "One")
return $x
```

Result:

```
<name>
  <family>Worker</family>
  <given>One</given>
</name>
```

2.3.3 Simple Range Query

A range query restricts a range by two values as known from relational queries. The comparison requires the use of the operators < and > (GeneralComparison according to XQuery 1.0).

Example 2.6 *In this example the query returns all* `name` *element nodes which contain in their child node a string which is lexicographically smaller than the string* `Worker`*.*

```
for $x in document("personal.xml")//name
where $x/family < "Worker"
return $x
```

Result:

```
<name>
   <family>Boss</family>
   <given>Big</given>
</name>
```

2.3.4 Range Query with a Combination of Values (using logical expressions)

When the above mentioned simple range query is extended to several attributes or text nodes the result is a combination of ranges. Combining several of them is also done by using `and` and `or` operations.

Example 2.7 *The query in this example returns all* `name` *element nodes whose* `family` *child node contains a string which is lexicographically larger than* `Worker` *and whose* `give` *child node contains a string which is lexicographically larger than* `One`*.*

```
for $x in document("personal.xml")//name
where ($x/family >= "Worker" and $x/given >= "One")
return $x
```

Result:

```
<name>
   <family>Worker</family>
   <given>One</given>
</name> <name>
   <family>Worker</family>
   <given>Two</given>
</name> <name>
   <family>Worker</family>
   <given>Three</given>
</name>
```

2.4 Predicates - Value, Position, Structure

This class basically describes the same properties as the class in section 2.3 but from a semantic perspective.

Besides a classification based on properties of the query (as in Section 2.3) it is also possible to classify according to properties of an element node. An important property in this context is to query the value of an element node, its position in the XML document and its structure. In the following we will further discuss a classification based on these three aspects.

2.4.1 Value of an Element Node

A query can be described by comparing an exact value with other source values. The position of the context node in the document is not relevant for the value comparison and it is a simple query which can formulated for every document which consists of at least one attribute and text node. The main difference between this class and the class "simple point query" is that in the latter case the definition is based on the way the query is formulated (syntax) while in this case querying for an exact value is important (semantic). This is further clarified by the fact that the more general query operates with a predicate and not with an FWR expression (For, With, Return). In addition, arithmetic expressions can be used in the more general query.

Example 2.8 *In this example element nodes are retrieved which contain the text* `Boss` *in a child element.*

```
document("personal.xml")//name[family ="Boss"]
```

Result:

```
<name>
   <family>Boss</family>
   <given>Big</given>
</name>
```

2.4.2 Position of an Element Node

The position of an element node in an XML document is tightly linked to document order. The preservation of order in the result of a query is a very important feature of the XQuery query language. This enables the comparison of two nodes from different element sequences according to their positions in the XML documents.

Example 2.9 *The query in this example localizes the element node in the middle of a sequence of element nodes, which are siblings. It then outputs every second element node which is located after (in document order) the localized node in the middle of the sequence.*

```
let $y:=
 document("personal.xml")/personnel/person[last() div 2]
return(
for $x in
 document("personal.xml")/personnel/person[position() mod 2 = 1]
where $x >> $y
return $x)
```

Result:

```
<person id="four.worker">
  <name>
    <family>Worker</family>
    <given>Four</given>
  </name>
  <email>four@foo.com</email>
  <link manager="Big.Boss"/>
</person>
```

2.4.3 Structure of an Element Node

When traversing a subtree of arbitrary depth a direct navigation to an element node is important. Basically every property of a node can be used for localization. The number of successors or attributes can also be used to describe the structure of an element node. The result of the query then is the root node of the traversed subtree.

Example 2.10 *In this example the query returns all element nodes (including their content) from the subtree starting at the `name` element node, which itself has two child nodes `given` and `family`*

```
document("personal.xml")/personnel/person/name[given and family]//*
```

Result:

```
<family>Boss</family>
<given>Big</given>
.
.
.
<family>Worker</family>
<given>Five</given>
```

Example 2.11 *The second example in this section describes a special case where all element nodes are returned recursively with their content.*

```
  document("personal.xml")//*
```

Result:

```
<personnel>
  <person id="Big.Boss">
    <name><family>Boss</family> <given>Big</given></name>
    <email>chief@foo.com</email>
    <link subordinates="one.worker two.worker three.worker four.worker
                        five.worker"/>
  </person>
  .
  .
  .
  <person id="five.worker">
    <name><family>Worker</family> <given>Five</given></name>
    <email>five@foo.com</email>
    <link manager="Big.Boss"/>
  </person>
</personnel>

 <person id="Big.Boss">
    <name><family>Boss</family> <given>Big</given></name>
    <email>chief@foo.com</email>
    <link subordinates="one.worker two.worker three.worker four.worker
                        five.worker"/>
  </person>
  .
  .
  .
  <person id="five.worker">
   <name><family>Worker</family> <given>Five</given></name>
    <email>five@foo.com</email>
    <link manager="Big.Boss"/>
  </person>

 <name><family>Boss</family> <given>Big</given></name>
 .
 .
 .
```

2.5 Join

Joins are among the most important features in query languages so that we decided to define a separate class for it even though there are some apparent overlaps with other classes (e.g. traversal, document order). The described XML join is similar to the join technique in relational database systems, which is based on the cross product of two sets of tuples. In XQuery two XML document can be joind on a per element basis. Although the join predicate can be formulated in a very general way as a Θ-join we will concentrate on the equi-join in the example below as this is probably the most often used form of a join. It also shows the logical relationship between the joined documents in a very intuitive way.

Example 2.12 *In this sample query the name of a person is joined with its manager via an attribut. The result of the query are newly constructed elements with the content of the* `name` *elements.*

```
for $x in document("personal.xml")/personnel/person
for $y in document("personal.xml")/personnel/person
where          $x/link/@manager = $y/@id
return
<newItem>
       {
       $x/name,
       $y/name
       }
</newItem>
```

Result:

```
<newItem>
    <name>
      <family>Worker</family>
      <given>One</given>
    </name>
    <name>
      <family>Boss</family>
      <given>Big</given>
    </name>
</newItem>
<newItem>
    <name>
      <family>Worker</family>
      <given>Two</given>
    </name>
    <name>
```

```
      <family>Boss</family>
      <given>Big</given>
    </name>
</newItem>
.
.
.
<newItem>
    <name>
      <family>Worker</family>
      <given>Five</given>
    </name>
    <name>
      <family>Boss</family>
      <given>Big</given>
    </name>
</newItem>
```

2.6 Text Search

Besides querying the structure of a document combined with values or positions it is also possible to search the whole text of a document for a string. Text hereby denotes the content of all text nodes, which are searched one after the other. It is different to a simple point query (Section 2.3.1) as for the text search substrings can be matched and that only the content of text nodes is matched (no tag names, etc.). As the text search is based on the built-in function *contains(operand1, operand2)* there is of course a relationship to the class built-in functions (Section 2.10.1) but we have decided to put text search into a separate class as it is very important especially for document-oriented XML documents.

Example 2.13 *In the example the complete document is searched for the string* `One`*. All elements are returned which contain the search string.*

```
for $x in document("personal.xml")//* [text()]
where contains($x, "One")
return $x
```

Result:

```
<given>One</given>
```

2.7 Sorting

The order of a sequence can be changed according to an order expression in the XQuery statement. The resulting sequence contains all items in the given sort order while the identity of the nodes (node identifiers) is preserved.

Example 2.14 *In this example a sequence of element nodes containing text is lexicographically sorted and returned.*

```
for $x in document("personal.xml")//email
return $x
sortby (.)
```

Result:

```
<email>chief@foo.com</email>
<email>five@foo.com</email>
<email>four@foo.com</email>
<email>one@foo.com</email>
<email>three@foo.com</email>
<email>two@foo.com</email>
```

2.8 Quantification

The XQuery languages provides two quantifiers (`some, every`) which implement tests on specified sequences. The tests specify whether an expression is evaluated to true for at least one or all items from the specified sequence.

Example 2.15 *The query in this example tests whether there is a `email` element node which contains the string `@foo.com`.*

```
for $x in document("personal.xml")//person
where some $y in $x/email satisfies contains($y, "@foo.com")
return $x
```

Result:

```
<person id="Big.Boss">
  <name><family>Boss</family> <given>Big</given></name>
  <email>chief@foo.com</email>
  <link subordinates="one.worker two.worker three.worker four.worker
                      five.worker"/>
</person>
```

```
<person id="one.worker">
   <name>
     <family>Worker</family>
     <given>One</given>
   </name>
   <email>one@foo.com</email>
   <link manager="Big.Boss"/>
</person>
   .
   .
   .
<person id="five.worker">
   <name>
     <family>Worker</family>
     <given>Five</given>
   </name>
   <email>five@foo.com</email>
   <link manager="Big.Boss"/>
</person>
```

2.9 Operations on Sets

The XQuery operations union, difference, and intersection are defined on sequences. They all share some features like the elimination of duplicates (based on node identities).

Example 2.16 *The query in this example creates a union of all* `manager` *and* `subordinates` *attributes.*

```
document("personal.xml")//link/@manager
union
document("personal.xml")/personnel/person/link/@subordinates
```

Result: (after the elimination of duplicates)

```
manager="Big.Boss"
subordinates="one.worker two.worker three.worker four.worker five.worker"
```

2.10 Application of Functions

For commonly used operations XQuery offers built-in functions. In addition, the user can define functions manually for special application scenarios leading to a classification into built-in and user-defined functions. Sequences which contain items are an important feature when navigating inside a document tree. Sequences are supported in XQuery by FLWR statements and sequence processing functions. Due to this built-in and user-defined functions are further divided into sequence processing functions and functions which process single values.

2.10.1 Built-In Functions

Processing Sequences

This subclass contains all built-in functions which operate on sequences. Typical representatives of this class are the aggregation functions.

Example 2.17 *The query in this example returns the occurrence of every element node in an XML document.*

```
let $el := document("personal.xml")//*
let $ll := distinct-values(for $e1 in $el
         return local-name($e1))
for $ln in $ll
return
 <element>
 {
  $ln, " ", count(for $e in $el where local-name($e)=$ln return $e)
 }
 </element>
```

Result:

```
<element>personnel 1 </element>
<element>person 6 </element>
<element>name 6 </element>
<element>email 6 </element>
<element>link 6 </element>
<element>family 6 </element>
<element>given 6 </element>
```

Processing Single Values

Similar to the user-defined functions in Section 2.10.2 the function requires a single value as input. An example is the function *substring(operand1, operand2, operand3)* which extracts substrings from input strings.

Example 2.18 *This sample query returns the first letter of each text node in the XML document.*

```
for $t in document("personal.xml")//family/text()
return substring(string($t),1,1)
```

Result:

```
B W W W W W
```

2.10.2 User-Defined Functions

Processing Sequences

This subclass is intuitively defined by the sequence based input of the function.

Example 2.19 *The below listed example recursively traverses the document tree and outputs the content of all text nodes in a separate element node.*

```
define function rek ($p)
{
 <output>
 {
  $p/text(),
  for $r in $p/*
  return rek($r)
 }
 </output>
}
let $p := document("data/xmp-data.xml")/*
return rek($p)
```

Result:

```
<output>

  <output>
    <output><output>Boss</output> <output>Big</output></output>
    <output>chief@foo.com</output>
    <output/>
  </output>

  <output>
    <output><output>Worker</output> <output>One</output></output>
    <output>one@foo.com</output>
    <output/>
  </output>
  .
  .
  .
</output>
```

Processing Single Values

Again it is intuitively clear that the input value is a single value and not a sequence.

Example 2.20 *The user-defined function in this example reverts the content of all text nodes of the `family` node.*

```
define function reverse ($s)
{
 for $i in (string-length($s) to 1)
 return substring($s,$i,1)
}
for $t in document("personal.xml")//family/text()
return reverse(string($t))
```

Result:

```
s s o B r e k r o W r e k r o W r e k r o W r e k r o W r e k r o W
```

2.11 Construction

We call the final class of our classification "Construction". A new document can be assembled from the contents of another document or from the results of queries. The new document can be of arbitrary depth and can contain all contents of the original or queried documents.

Example 2.21 *The content of the element nodes `given` and `family` is inserted into a newly constructed element node `newItem`.*

```
for $x in document("personal.xml")/personnel/person
return
      <newItem>
      {
      $x/name/given/text()," ",
      $x/name/family/text()
      }
      </newItem>
```

Result:

```
<newItem>Big Boss</newItem>
<newItem>One Worker</newItem>
<newItem>Two Worker</newItem>
<newItem>Three Worker</newItem>
<newItem>Four Worker</newItem>
<newItem>Five Worker</newItem>
```

2.12 Summary

In this chapter we have presented a classification for XQuery which elaborated the different features of the query language. The classification has also shown that XPath is a fundamental part of XQuery. XPath is used for addressing positions and navigation within the XML data in several classes. Also the evaluation of predicates which use paths and XML tag contents is a fundamental part of XQuery as can be seen in the classification. Refering to this we will deeper analyse navigation with XPath axes in Chapter 3 and discuss the evaluation of path predicates with values (XML tag contents) in Chapter 4 when we discuss the storage of several XML documents.

Due to the complexity of XQuery we have also seen that it is not possible to define disjunctive classes and to assign a query to exactly one class. The classification usually results in several classes to which the query belongs (or whose features it uses). When the query is not too complex it is often possible though to chose one class which is the main focus of the query.

The application fields of a classification scheme as we have presented in this chapter are manifold. As already described in the introduction to this chapter the classification shows basic features of XQuery which in their combination form the complete query language and helps in understanding the language. The classification can also be used as an introduction to the language for interested parties who do not want to read the standard.

A very interesting application of the classification besides XML storage is to use the classes as a source for test cases for XQuery processors in development. Especially processors for query languages require test cases which cover all aspects of a query language. It is difficult to ad-hoc find sets of queries which use a broad range of features of the query language. This can be solved by choosing queries from the above presented classes to help the developers of the query processor to provoke errors which would otherwise remain undiscovered. Simple queries can be chosen at first and more complex queries can be built from them. This results in queries which use much more features of the query language than an ad-hoc approach thus increasing software quality.

Chapter 3

A Numbering Scheme for XML Paths

3.1 Introduction

Previous chapters described the nature of XML as a data format and also of queries on XML data. We have seen that XML is hierarchical data and that it features structure and content. While content is further discussed when we present our multidimensional model to store XML in a relational schema, we concentrate in this chapter on navigation in the relational representation of XML documents.

As seen before, XML documents are usually modeled in a tree representation. Navigation in trees is one of the fundamentals of computer science and is based on only a few operations. When talking about navigation this is highly correlated to a change of position. Thus it is also important to discuss positions in trees when talking about navigation. For XML documents the W3C defined the XPath language to provide a standardized way to address positions and navigation in XML documents. As we are going towards storage of XML in relational database systems we discuss navigation with XPath in a relational representation of the hierarchical XML document first. This requires a s.c. numbering scheme of the XML graph which preserves the hierachical structure and document order but is still simple enough to allow efficient navigation as required for the processing of XPath. The same numbering scheme is then used later for the storage of XML with our multidimensional approach.

In the remainder of this chapter we introduce a numbering scheme (based on Multidimensional Hierarchical Clustering), we show that we can process XPath on this representation and compare it to the *pre/post* numbering scheme for XML documents from Section 1.3.1.

3.2 Multidimensional Hierarchical Clustering (MHC)

Multidimensional Hierarchical Clustering (MHC) is a technique that was originally developed for data warehousing applications [MRB99]. In data warehousing MHC is used to cluster data with respect to multiple hierarchical dimensions. In our context we use a variant of MHC which does not consider hierarchies in several dimensions but restricts the application to one dimension which is a special case of MHC. Consequently, we have decided to use the term MHC throughout this work to show the strong similarity. We briefly describe the technique in an overview, show its application in our context of XML indexing, how we use MHC to preserve order and how to store paths in a compact form.

It is well known by now that XML documents form a hierarchy with nodes and edges. We assume that each XML document has a maximal hierarchy of depth h leading to an overall of $h + 1$ levels in each document. We use a tree representation which is slightly different from the commonly used one. Identical paths which occur several times within a subtree are rewritten by using repetition numbers to preserve uniqueness and order. The paths from example 3.1 are rewritten to **<author>[1]<FN>[1], <author>[1]<LN>[1], <author>[2]<FN>[1], <author>[2]<LN>[1]** .

Example 3.1
```
<author><FN>Michael</FN><LN>Bauer</LN></author>
<author><FN>Rudolf</FN><LN>Bayer</LN></author>
```

In the following we treat the repetition numbers as a separate hierarchy level. The set of levels is ordered according to the document structure. Each hierarchy level i is a set of sets L ($L = \bigcup_{i=0}^{n} L_i$), where each set L_i consists of nodes m_k^i. L_0 is defined to be the root level. Due to the hierarchical nature every member m_k^i has a (varying) number of children. A function ord_m defines a numbering scheme for the children of m_k^i and assigns each child of m_k^i a number between 1 and the total number of children of m_k^i, i.e. $ord_m : children(m) \longrightarrow \{1, \ldots, |children(m)|\}$. In contrast to the classical MHC definition the function ord_m needs to be order preservering so that nodes are sequentially numbered according to document order. We call each element of $\{1, \ldots, |children(m)|\}$ a surrogate of a node. To encode paths through the hierarchy we introduce the concept of compound surrogates. A compound surrogate is formed by recursively concatenating the compound surrogate of the father node with the surrogate of the current node. More formally the compound surrogate for a node is defined as follows:

$$cs(m^i) = \begin{cases} ord_{father(m^i)}(m^i), & \text{if } i = 1 \\ cs(father(m^i)) \circ ord_{father(m^i)}(m^i), & otherwise \end{cases}$$

Example 3.2 *Using figure 3.2 the path* `<paper>[1]<author>[1]<FN>[1]` *is transformed into the compound surrogate 1.1.2.1.1.1*

Compound surrogates can be stored very efficiently in a compact binary representation. The upper limit of reserved bits for each surrogate can be calculated by the formula

$$surrogate(i) = max\{|(children(m)|) \text{ where } m \in level(i-1)\}$$

The length of the surrogates must be chosen sufficiently large in advance and although the fixed length leads to static boundaries of the surrogates this can be neglected as extending every surrogate by one bit doubles the number of children that can be addressed at every level.

MHC enables us to find a compact and order preserving representation for paths in XML documents. It also fixes the problem of long equal prefixes which has already been addressed in work on special index structures for XML [CSF$^+$01]. It is also possible to evaluate path expression on compound surrogates. Evaluating simple path expressions (i.e. full qualified paths) in MHC is equivalent to point queries on the compound surrogates while other path expressions are equivalent to range queries or combinations of point and range queries. We are going to further discuss this in the next section.

```
<paper>
<title> Sample Title </title>
<author> <FN> John </FN> <LN> Smithe </LN>
         <affiliation> University</affiliation> </author>
<author> <FN> Jack </FN> <LN> Brown </LN>
         <affiliation> Industry </affiliation> </author>
<keyword> Data structures </keyword>
<keyword> Database Management Systems </keyword>
<abstract> Sample Abstract </abstract>
<bibrec> <author> <FN> Jack </FN> <LN> Brown </LN>
           <affiliation> Industry </affiliation>
         </author>
         <author> <FN> John </FN> <LN> Smith </LN>
           <affiliation> Another University </affiliation>
         </author>
         <title> News about DBMS </title>
</bibrec>
<bibrec> <author> <FN> Jack </FN> <LN> Brown </LN>
           <affiliation> Industry </affiliation>
         </author>
         <author> <FN> John </FN> <LN> Smith </LN>
           <affiliation> Another University </affiliation>
         </author>
         <title> A system for fast query processing </title>
</bibrec>
</paper>
```

Figure 3.1: Sample XML document (used in Figure 3.2)

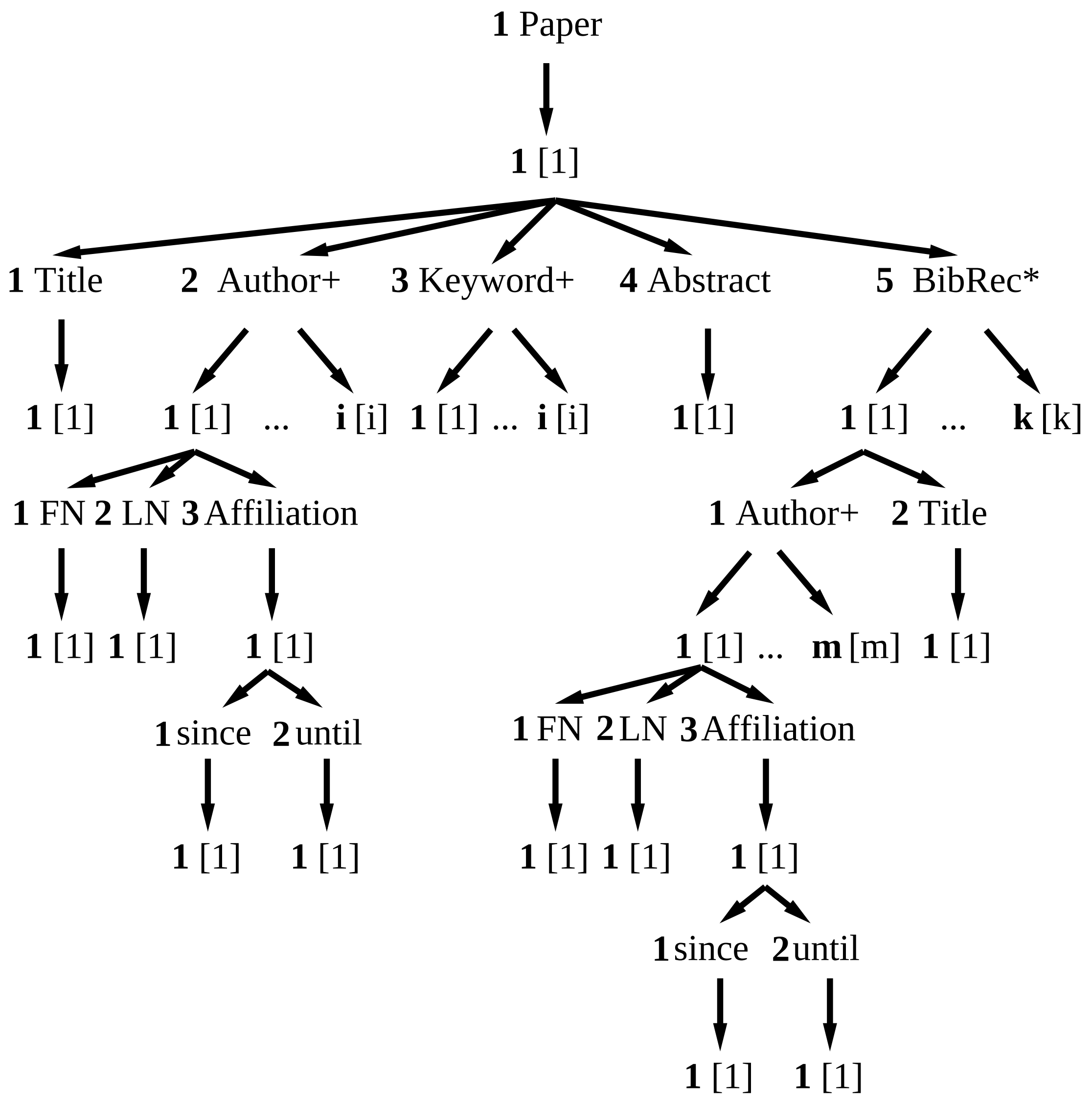


Figure 3.2: XML document hierarchy and MHC surrogates

MHC Surrogate	rewritten Path
1.1.1.1	paper[1]/title[1]
1.1.2.1.1.1	paper[1]/author[1]/FN[1]
1.1.2.1.2.1	paper[1]/author[1]/LN[1]
1.1.2.1.3.1	paper[1]/author[1]/affiliation[1]
1.1.2.2.1.1	paper[1]/author[2]/FN[1]
1.1.2.2.2.1	paper[1]/author[2]/LN[1]
1.1.2.2.3.1	paper[1]/author[2]/affiliation[1]
1.1.3.1	paper[1]/keyword[1]
1.1.3.2	paper[1]/keyword[2]

Table 3.1: Paths and corresponding MHC surrogates (according to Figure 3.2)

3.3 Mapping XPath Location Steps to MHC

3.3.1 XPath Location Steps

XPath defines several *location steps* to navigate in XML documents. A total number of 13 location steps are defined in the standard. All location steps select a set of nodes relative to a context node. A sequence of location steps form a *location path*. One can distinguish absolute location paths (paths which start from the root) and relative location paths (paths which start at a node other than the root) but this separation is not of importance for our work. A location step consists of three parts: an *axis* to specify the tree relationship between the context node and the node selected by the location step, a *node test* to specify the node type and name, and zero or more *predicates* to refine the set of result nodes selected by the location step.

In the further work we concentrate on axes and processing of axes. A complete list of the 13 axes defined in the standard is shown in Table 3.2.

3.3.2 Processing XPath Axes with MHC

To process the axes from Table 3.2 on MHC compound surrogates we use a different notation for the compound surrogates than in Section 3.2.

Definition 3 *The current node in an XML document is addressed by the compound surrogate* $c = X_1.Y_1.\cdots.X_k.Y_k.\cdots.X_n.Y_n$.

The variable X_i hereby denotes nodes while Y_i denotes repetition numbers. The set of compound surrogates is denoted by C in the remainder of this work. Compound surrogates are tightly linked to positions in XML documents and document order. When talking about compound surrogates we therefore need an order to be able to tell whether one compound surrogate is smaller or larger than another compound surrogate. This order defines whether positions in the XML documents are before or after a context node. In the following we

Axis	Description
child	children of the context node
descendant	recursive *child* operation
descendant-or-self	like *descendant* plus context node
parent	parent node of the context node
ancestor	recursive *parent* operation
ancestor-or-self	like *ancestor* plus context node
following	nodes after the context node in document order
preceding	nodes before the context node in document order
following-sibling	same as *following*, but same parent as context node
preceding-sibling	same as *preceding*, but same parent as context node
attribute	attributes of the context node
self	the context node
namespace	namespace nodes of the context node

Table 3.2: List of XPath axes and their descriptions

define a relation on compound surrogates and then show that it is a total order, i.e. that the relation is reflexive, transitive, antisymmetric and linear.

Definition 4 *Given two compound surrogates* $c_1 = X_1.Y_1.\cdots.X_k.Y_k.\cdots.X_n.Y_n$
and $c_2 = M_1.N_1.\cdots.M_l.N_l.\cdots.M_p.N_p, (X, Y, M, N \in \mathbf{N})$
The relation $R \subset C \times C$ *is defined by* $(c_1, c_2) \in R \Longleftrightarrow$
case 1: c_1 *is a prefix of* c_2*, i.e.* $\forall i \leq n : X_i = M_i \wedge Y_i = N_i \wedge n \leq p$*. (prefix condition)*
case 2: $\exists i, 1 \leq i \leq \min\{n, p\} : (\forall j, 1 \leq j < i : X_j = M_j \wedge Y_j = N_j) \wedge$
$(X_i < M_i \vee (X_i = M_i \wedge Y_i < N_i))$

Theorem 3.1 *The relation R, which we in the following also call* $\leq$*, is a total order.*

Proof: In the following, we sketch the idea of the proof, in which we show that the above stated relation is a total order, i.e. that it is reflexive, transitive, antisymmetric, and linear.
reflexive: $c_1 \leq c_1$. This is satisfied as every compound surrogate is a prefix of itself.
transitive: $c_1 \leq c_2 \wedge c_2 \leq c_3 \Rightarrow c_1 \leq c_3$. For each step we have to distinguish the above stated two cases leading to an overall of four cases.
case 1.1: In this case c_1 is a prefix of c_2 and c_2 is a prefix of c_3, which leads to the fact that c_1 is a prefix of c_3. Since the prefix relation is a partial order, the prefix relation is transitive.
case 1.2: In this case we also assume that c_1 is a prefix of c_2 but assume that case 2 (from the theorem) holds for c_2 and c_3. This leads to the case where $c_1 \leq c_3$ due to case 1 or case 2 from the definition. Transitivity is satisfied as $\leq$ is a total order in $\mathbf{N}$ and consequently for all surrogates X and Y in C.

case 2.1: Here $c_1 \leq c_2$ due to case 2 and $c_2 \leq c_3$ due to case 1. The proof is analogous to case 1.2 as the same conditions hold with the exception that $c_1 \leq c_3$ because of case 2 (from Definition 4).
case 2.2: For this case we assume that both $c_1 \leq c_2$ and $c_2 \leq c_3$ due to case 2. This case is also similar to case 1.2 as it can be seen as a generalization. Again transitivity is satisfied as $\leq$ is a total order in $\mathbf{N}$.
antisymmetric: $c_1 \leq c_2 \wedge c_2 \leq c_1 \Rightarrow c_1 = c_2$ Here we would have to distiguish four cases analogously to transitivity. Only the case $c_1 \leq c_2$ because of case 1 and $c_2 \leq c_1$ because of case 1 can actually occur. Every involvement of case 2 leads to a contradiction. Following this we can argue that if the prefix condition holds, then the relation is antisymmetric because of the equality of the "prefixes".
linear: Linearity is defined as: $\forall c_1, c_2 : c_1 \leq c_2 \vee c_2 \leq c_1$
We can prove this by examining two cases.
case 1: One compound surrogate is a prefix of the other. Consequently this compound surrogate is $\leq$ than the other according to case 1 (from the definition).
case 2: If the two compound surrogates are not prefixes of each other then there exists a position i where the two compound surrogates differ. In this case one of the two compound surrogates is defined as being $\leq$ to the other one.

□

In addition to the total order we have just defined there are other order relations which are also quite intuitive ($<$ and the apparent counterparts $>$ and $\geq$).

As Grust has shown in [Gru02] it is sufficient to talk only about the four axes *preceding, following, ancestor* and *descendant* as the union of the results (plus the context node) contains all nodes of the document. All other axes are therefore only special cases of these four basic axes. For the sake of completeness we nevertheless show how to process all navigation axes (i.e. all axes except attribute, self, namespace) with MHC.

We define the compound surrogate of a context node as $c = A_1.B_1.\cdots.A_k.B_k$. A_i hereby denotes tag surrogates, while B_i denotes the repetition number. In the following we give an explanation of the XPath axes based on this notation in a set based manner. The different axes are always formulated relatively to context node c and use different manipulations on the compound surrogate of the context node. The compound surrogate is either extended, shortened or other compound surrogates are identified with properties relatively to the context node.

1. child: $(c) = \{A_1.B_1.\cdots.A_k.B_k.A_{k+1}.B_{k+1,i}\}$
   The variable A_{k+1} represents the child tag surrogates and $B_{k+1,i}$ the repetition number i for the tag surrogate A_{k+1}. In other words: The compound surrogate is extended by exactly one step (consisting of two surrogates).

2. descendant: $(c) = \{A_1.B_1.\cdots.A_k.B_k.\cdots.A_{k+l}.B_{k+l,i}\}$
   As the descendant axis is a recursive application of the child axis this axis defines a subtree with the context node being the root. The expression above considers a

subtree upto a height of l. The variable i indicates that every tag surrogate can be followed by repetition numbers.

3. descendant-or-self: $(c) = \{A_1.B_1.\cdots.A_k.B_k.A_{k+l}.B_{k+l,i}\} \cup \{A_1.B_1.\cdots.A_k.B_k\}$
   This axis adds the context node to the result of the descendant axis.

4. parent: $(c) = \{A_1.B_1.\cdots.A_{k-1}.B_{k-1}\}$
   Starting from a context node the parent axis can be generated by calculating the prefix of the compound surrogate of the context node (i.e. shortening the compound surrogate by two surrogates). This is expressed by the index $k-1$.

5. ancestor: $(c) = \{A_1.B_1.\cdots.A_{k-l}.B_{k-l,i} \wedge 1 \leq l \leq k-1\}$
   The ancestor axis is a recursive application of the parent axis. It can be calculated by recursively shortening the compound surrogate. This is expressed in the above statement by the variable l.

6. ancestor-or-self: $(c) = \{A_1.B_1.\cdots.A_{k-l}.B_{k-l,i}\} \cup \{A_1.B_1.\cdots.A_k.B_k\}$
   This axis simply adds the context node to the result of the ancestor node.

7. following: $(c) = \{c_n | A_1.B_1.\cdots.A_k.(B_k+1) \leq c_n\}$
   It is important to note that this axis can not be expressed by returning all compound surrogates larger than the context node as this would also include the descendant axis. Therefore this axis has to be expressed with the sibling of the context node. Hereby c_n denotes compound surrogates. It can happen that the value $B_k + 1$ does not exist. The expression is still valid though because of the definition of $\leq$.

8. preceding: $(c) = \{c_n | (c_n < A_1.B_1.\cdots.A_k.B_k)\} \backslash \text{ancestor}$
   Processing the preceding axis requires a combination of two steps. At first all compound surrogates have to be retrieved which are smaller than the compound surrogate of the context node. In a postfiltering stage the compound surrogates from the ancestor node have to be eliminated. Hereby c_n again denotes compound surrogates.

9. following-sibling: We have to distinguish two cases:
   $(c) = \{A_1.B_1.\cdots.A_k.B_{k,i} \wedge B_{k,i} > B_k\} \cup \{A_1.B_1.\cdots.A_{k,i}.B_{k,i} \wedge A_{k,i} > A_k\}$

   The first case specifies the compound surrogates of siblings of the same tag name (which are of the same tag element and only differ in the repetition number) while the second case specifies the following siblings with different tag names.

10. preceding-sibling: We have to distinguish two cases:
   $(c) = \{A_1.B_1.\cdots.A_k.B_{k,i} \wedge B_{k,i} < B_k\} \cup \{A_1.B_1.\cdots.A_{k,i}.B_{k,i} \wedge A_{k,i} < A_k\}$

   This axis is equivalent to the above mentioned following sibling axis but of course returns the compound surrogates which are smaller than the context node.

3.3.3 Observations on One- and Two-Dimensional Mappings

The previous sections have shown our variant of MHC, which is a one-dimensional mapping, i.e. the hierarchical structure is mapped to values in one dimension. This is in contrast to the *pre/post* mapping proposed by Grust which uses two dimensions for each node (actually even three dimensions as we will see). While this difference does not appear to be very important at a first glance the consequences are surprising. For the following discussion we have to recall the fact that four axes partition a document into four parts (left, right, above, and below of the context node). In a two-dimensional mapping the universe can be easily partitioned into four parts for a given point in the universe (coordinates: x, y). A possible partitioning are four areas:

1. $< x \wedge > y$ (upper left area, corresponds to *descendant* axis)
2. $> x \wedge > y$ (upper right area, corresponds to *following* axis)
3. $> x \wedge < y$ (lower right area, corresponds to *ancestors* axis)
4. $< x \wedge < y$ (lower left area, corresponds to *preceding* axis)

This partitioning is identical to the partitioning proposed in the work of Grust.

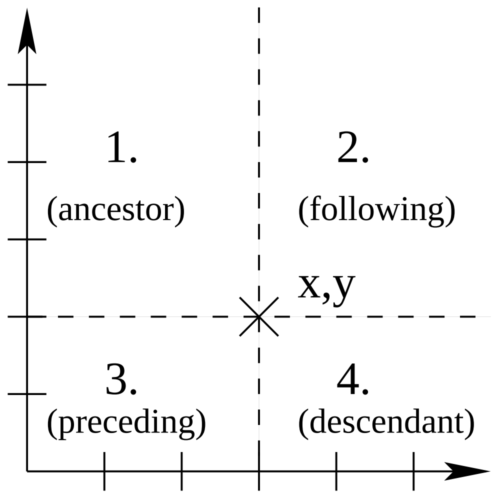


Figure 3.3: Two dimensional partitioning into four areas

In a one-dimensional mapping the situation is different. A simple partitioning using $<$ as the order relation for a point (coordinate: z) in a one-dimensional mapping partitions the universe into two areas.

1. $< z$ (left area, corresponds to *ancestor* and *preceding* axis)
2. $> z$ (right area, corresponds to *descendant* and *following* axis)

If a mapping resembles document order then the two areas correspond to the ancestor axis plus the preceding axis and the descendant axis plus the following axis. The corresponding XPath axes are special cases of these two constructs and require special query patterns in MHC, as shown above.

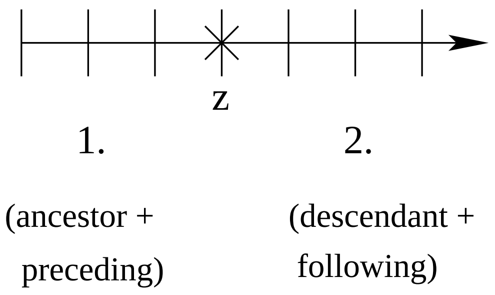


Figure 3.4: One dimensional partitioning into two areas

3.3.4 Implementation

Preliminaries

For the implementation of the proposed method we have decided to map the surrogates to bit strings. This representation offers two advantages. The size of the bit strings can be chosen sufficiently large in advance as adding one digit doubles the value range of the surrogate and compound surrogates can be generated by simply concatenating the bit strings of surrogates. The length of bit strings for compound surrogates is also fixed and if a longer path for a compound surrogate does not exist the remaining bits are filled up with 0.

Example 3.3 *The compound surrogate 3.1.2.1 is transfered into the bit string 0b 011 001 010 001 000 000. A bit string is hereby denoted by 0b and three bits where reserved for each surrogate resulting in 18 bits for the compound surrogate.*

As we are using bit strings the left most bit is the first bit and consequently is at position 1.

Example 3.4 *In the bit string 0b010 001, the bits 2 and 6 are set.*

For the further discussion of the implementation we take a closer look on the XPath axes. Some of the axes are special among the others as they require the parent relationship of the context node to be properly processed. In the following we have listed which axes do not require the knowledge of the parent relationships:

- child, descendant

- following,
- preceding (as it is preceding without ancestors).

In contrast the following axes do require the knowledge of the parent relationships:

- parent, ancestor
- following-sibling, preceding-sibling

Two approaches to take care of the parent relation in our implementation are possible. Either the parent for each node is stored explicitly or the parent is calculated from the context node.

When using MHC and an implementation using bit strings an efficient calculation of the compound surrogate of the parent node is possible by generating the prefix of the compound surrogate of the context node. The calculation differs whether the number of reserved bits is fixed for all surrogates or if it differs from surrogate to surrogate. Tag surrogates and repetition numbers can be treated as one long surrogate as the parent axes always returns a node which consists of a tag surrogate and a repetition number.

Calculating the prefix for a fixed number of bits

It is important to again note that for each surrogate counting starts at one and the remaining bits in the bit string are set to 0. The calculation is then done in two steps with the knowledge of the number of bits per surrogate `bps` and the position of the right-most set bit of the compound surrogate (calculated by a function `findbit()`). As a consequence from starting the surrogates from 1 at least one bit is always set for a surrogate. It is now easy to calculate the hierarchy level of a compound surrogate by calculating $h = (\texttt{findbit}(\texttt{cs}) - 1) \div \texttt{bfs}$. From this formula the number of relevant bits of the parent can be calculated by multiplying the hierarchy level with the number of bits per surrogate. The compound surrogate of the parent node is then calculated by clearing the remaining bits after the surrogate. An exemplary calculation is performed in Example 3.5.

Example 3.5 *The following compound surrogate c consists of eight surrogates and four bits per surrogate:* $c =$ 0b0010 0001 0001 0001 0000 0000 0000 0000*. The number of bits which are the prefix of the parent can be calculated with the following formula: ((findbit(c)-1)÷4)*4. It is then sufficient to fill the remaining bits of the compound surrogate after the calculated position with 0. The findbit function is hereby defined as returning the position of the right-most set bit and multiplication and division return integer values. The calculation results in the following compound surrogate:* 0b0010 0001 0001 0000 0000 0000 0000 0000

Calculating the prefix for a variable number of bits

A different number of bits for each surrogate results in a more complex calculation. In the following we give two functions which perform the required calculations. Both functions expect an array $a[n]$ which contains the bit positions of the right most bit of each surrogate (Example 3.6).

Example 3.6 *For four surrogates ($n = 4$) with a bit length of 3, 6, 4, and 5 bits, the array contains $a[0] = 3$, $a[1] = 9$, $a[2] = 13$, $a[3] = 18$.*

The first function `generateParent()` calculates the parent for a given context node in the compound surrogate representation while the second function `generateAncestor()` calculates the ancestor. Both functions are given in pseudo code similar to the programming language C.

```
compoundsurr generateParent(compoundsurr contextnode){
 int i,k=0;

 i = findbit(contextnode);
 while (a[k] < i)
   k = k + 1;

 parent = set0(contextnode, a[k-1], a[n]));
 return parent;
}
```

The function `generateParent()` calculates the parent by clearing the last two surrogates (tag number and repetition number). The variable i is set to the bit position of the right-most bit by the function `findbit` which has already been defined above. The `while` loop sets k to the surrogate step which contains the right-most bit (the right-most bit is apparently always a repetition number). The `set0()` function sets the bits between `a[k-1]` and `a[n]` to zero.

```
ancestor generateAncestor (compoundsurr contextnode){

 setofcompoundsurr ancestor = emptyset;

 while (findbit(contextsurr) > a[1]){
   parent = generateParent(contextnode);
   addToAncestor(parent);
   contextsurr = parent;
 }
 return ancestor;

}
```

The algorithm for `generateAncestor()` calculates the ancestors of a context node by repeatingly calculating the parent of a node. The body of the `while` loop is executed until the rightmost bit is in the second repetiton number `a[1]`. The `while` loop terminates when the last parent generated is the root node. A parent node is calculated by the above explained `generateParent()` algorithm. The ancestors are returned in a data structure which represents a set of compound surrogates (`setofcompoundsurr`). This set is initialized with an empty set in advance. A compound surrogate is added to the set by `addToAncestor()`.

Due to their complexity the functions `generateParent()` and `generateAncestor()` either have to be used by an application on top of a database system or the database system has to support *User Defined Functions*. For certain applications (e.g. the ancestor-or-self axis) `generateAncestors()` has to be extended so that the compound surrogate of the context node is also added to the result set.

In cases where it is not possible to calculate the parent node from the context node it is necessary to explicitly store the parent node with each node (as it is done e.g. with the mapping proposed by Grust)

Relational Schema

According to the above discussion two different relational schemas can be chosen to store the information which is required for navigation. The first version stores the `compound surrogate`, the `tag name` and a `flag`, which distinguishes elements and attributes. This single table is already sufficient to perform XPath queries, if the relational system offers methods to calculate parents and ancestors of context nodes. The content of the tags (the actual content of the XML document) is then stored separately in a separate table which uses a foreign key relationship with the compound surrogate (as in the pre/post mapping).

The second version of the schema is equivalent to the first with the only difference that the compound surrogate of the parent node is explicitly stored in the table. This table thus contains the attributes `compound surrogate`, `parent compound surrogate`, `tag name`, and an indicator `flag`. As mentioned above the actual XML content is again stored in a separate table. Throughout this chapter we refer to tables folowing one of the schemas as *typedim*. Whether the parent is explicitly stored in typedim is visible from the context.

Example 3.7 *In this example elements are marked with the value 'E' (attributes would be marked with an 'A'), other equivalent boolean values would also be appropriate. The compound surrogates are stored with 3 bits per surrogate and an overall length of 18 bits. The first table shows the relational schema with sample data for the first variant. (see Figure 3.1 on page 51 for the document).*

surr	*tag*	*flag*
0b001 001 000 000 000 000	*paper*	*E*
0b001 001 001 001 000 000	*title*	*E*
0b001 001 010 001 000 000	*author*	*E*
0b001 001 010 001 001 001	*FN*	*E*
0b001 001 010 001 010 001	*LN*	*E*
0b001 001 010 001 011 001	*affiliation*	*E*
0b001 001 010 010 000 000	*author*	*E*
0b001 001 010 010 001 001	*FN*	*E*
0b001 001 010 010 010 001	*LN*	*E*
0b001 001 010 010 011 001	*affiliation*	*E*
0b001 001 011 001 000 000	*keyword*	*E*
0b001 001 011 010 000 000	*keyword*	*E*

The second variant explicitly stores the compound surrogate of the parent node For the root node the parent is stored with a NULL value

surr	*parent*	*tag*	*flag*
0b001 001 000 000 000 000	*NULL*	*paper*	*E*
0b001 001 001 001 000 000	*0b001 001 000 000 000 000*	*title*	*E*
0b001 001 010 001 000 000	*0b001 001 000 000 000 000*	*author*	*E*
0b001 001 010 001 001 001	*0b001 001 010 001 000 000*	*FN*	*E*
0b001 001 010 001 010 001	*0b001 001 010 001 000 000*	*LN*	*E*
0b001 001 010 001 011 001	*0b001 001 010 001 000 000*	*affiliation*	*E*
0b001 001 010 010 000 000	*0b001 001 000 000 000 000*	*author*	*E*
0b001 001 010 010 001 001	*0b001 001 010 010 000 000*	*FN*	*E*
0b001 001 010 010 010 001	*0b001 001 010 010 000 000*	*LN*	*E*
0b001 001 010 010 011 001	*0b001 001 010 010 000 000*	*affiliation*	*E*
0b001 001 011 001 000 000	*0b001 001 000 000 000 000*	*keyword*	*E*
0b001 001 011 010 000 000	*0b001 001 000 000 000 000*	*keyword*	*E*

As mentioned already in [TVB$^+$02] the storage of tag names can be modified to store complete paths. This especially supports queries with a completely specified path as the number of self-joins in SQL queries is greatly reduced. In the case of processing XPath axes we neglect this feature as we want to examine the case of explicitly processing the axes one after the other. In addition we want to evaluate our mapping against Grust's mapping [Gru02] which also uses single tag names.

If complete paths are stored then the mapping can be further optimized by leaving out the root element as it is always identical for documents conforming to a DTD. This shortens the compound surrogates by one tag surrogate and one repetition number and has been performed in Chapter 4.

XPath Axes in SQL

By transforming XPath axes to an implementation of our mapping we can give SQL statements for each axes. Due to the above considerations about the parent relationship we give different statements for each axis which require the parent relationship. Due to the nature of SQL it is possible to formulate many semantically equivalent query statements. Therefore we concentrate on mainly three different versions. Two versions explicitly calculate the parent from the context node (one variant for a static length of surrogates and one for flexible length of surrogate) while the other uses the parent node stored in the relational schema for each context node. Axes which do not require the parent relationship use the same statement for all cases. In the SQL statements we assume a length of 24 bits for a tag surrogate plus a repetition number and an overall length of 288 bits (this length is later used for the performance evaluation). In the following list it is also important to note that we omited nametests, which are required for a location step but not an axis. We also do not include predicates which test for elements and attributes.

The examples are formulated in TB/SQL which is the implementation of the database system Transbase which we will later use for performance evaluation. TB/SQL is an implementation compatible to the standard ISO/DIS 9075. TB/SQL realizes most of SQL2 (Intermediate LEVEL). It also adds some functional extensions. In the following we use the three extensions `SUBRANGE()`, `FINDBIT()`, and `MAKEBIT()`.

- `SUBRANGE(expr1, expr2)` creates a bit string which consists of the bits from position *expr1* to position *expr2*.
- `FINDBIT(bexpr, 0)` returns the position of the last 1-bit in *bexpr*.
- `MAKEBIT(iexpr1)` constructs a bit string which has 1-bits on those positions described by *iexpr1*.

In the following the starting point for all queries is a context node given by its *contextsurr* which denotes its compound surrogate.

1. child: The child axis is implemented by testing a prefix of the compound surrogate of the context node. The length of the prefix is defined by the level of the context node in the XML hierarchy. The next surrogate step (tag number and repetition number) is variable and not restricted in the query. The remaining postfix following the possible child nodes has to be set to zero (0b0 as a bit string). In our example we assume that the context node is on the third level in the XML hierarchy resulting in a prefix of 72 bits. For this query it is required that the level of the context node in the XML hierarchy is known otherwise it has to be calculated with the same techniques as parent nodes.

   ```
   SELECT surr FROM typedim
   WHERE typedim.surr SUBRANGE(1,72) = contextsurr SUBRANGE (1,72)
   AND typedim.surr SUBRANGE (97, 288) = 0b0
   ```

2. descendant: As the descendant axis is a recursive application of the child axis it is sufficient to only test a prefix of the compound surrogate of the context node. The length of the prefix has to be chosen equivalently to the child axis. We again assume that the context node is on the third level of the XML hierarchy. As the subrange would also return the context node itself we have to ensure that the returned surrogates are not equal to the compound surrogates of the context node.

   ```
   SELECT surr FROM typedim
   WHERE typedim.surr SUBRANGE (1,72) = contextsurr SUBRANGE (1,72)
   AND typedim.surr <> contextsurr
   ```

3. descendant-or-self: The descendant-or-self axis is equivalent to the descendant axis and can be formulated even simpler as the compound surrogate of the context node does not have to be explicitly excluded. The length of the prefix again has to be chosen equivalently to the child axis. We again assume level 3 in the XML hierarchy.

   ```
   SELECT surr FROM typedim
   WHERE typedim.surr SUBRANGE (1,72) = contextsurr SUBRANGE (1,72)
   ```

4. parent: Depending on whether the parent node is explicitly stored in the queried relation or has to be calculated, the SQL statement differs fundamentally. In the following, three different versions are given to query for the parent node of a context node. The first version of the SQL statement explicitly calculates the parent node under the assumption that the length of the surrogates is static over the bit string. The calculation identifies the right most bit according to the method stated above and tests the prefix of the context node followed by a zeroed bit string.

   ```
   SELECT surr FROM typedim
   WHERE
     typedim.surr SUBRANGE(1, ((FINDBIT(contextsurr, 0)−1)/24)*24) =
     contextsurr SUBRANGE(1, ((FINDBIT(contextsurr, 0)−1)/24)*24)
   AND
     typedim.surr SUBRANGE( ((FINDBIT(contextsurr, 0)−1)/24)*24+1, 288)
       = 0b0
   ```

   The second version uses a User Defined Function as already described above to calculate the parent.

   ```
   SELECT parent FROM generateParent(contextsurr)
   ```

   The third version uses a schema for typedim which explicitly contains the parent node for every context node. Thus the parent can be retrieved very easily.

   ```
   SELECT typedim.parent FROM typedim
   WHERE typedim.surr = contextsurr
   ```

5. ancestor: The ancestor axis is a recursive application of the parent axis. It can be implemented with a User Defined Function.

   ```
   SELECT surr FROM generateAncestor(contextsurr)
   ```

6. ancestor-or-self: To include the context node to the result of the ancestor axis a variant of `generateAncestor` can be used.

```
SELECT surr FROM generateAncestorSelf(contextsurr)
```

7. following: The following axis returns the nodes following the context node in document order but excluding descendants. Excluding descendants can be achieved by calculating the following sibling (which is a node and not an axis) by an addition of one to the corresponding surrogate step (i.e. to the repetition number). Addition on the bit strings might lead to an overflow in the surrogate. The overflow can be propagated to the other surrogates as is expected from addition on bits. As zero values do not exist for surrogates this does not affect query results. It has to be taken care of though that there is no overflow for a context node with all bits set for the compound surrogate. This would result in `0b0` after the addition and the following axis would return all nodes in the database. This can be omited though by preventing the generation of a compound surrogate with all bits set when the XML document is inserted into the database. The following SQL statement uses addition on bit strings. Addition on bitstrings is not available in the version of the database system Transbase which is later used for the performance evaluation. We give the statement nevertheless because it is useful on systems which enable bitwise addition (among other required functions) and might work in later versions of Transbase.

```
SELECT surr FROM typedim
WHERE
  typedim.surr >=
  contextsurr + MAKEBIT(((FINDBIT(contextsurr, 0)-1)/24)*24+1)
```

An alternative can also be formulated by returning the compound surrogates larger than the context node but removing the descendants. This variant requires the same knowledge about the depth of the context node in the XML hierarchy as the child axis.

```
SELECT surr FROM typedim
WHERE typedim.surr > contextsurr AND typedim.surr NOT IN
(SELECT surr FROM typedim
WHERE typedim.surr SUBRANGE (1,72) = contextsurr SUBRANGE (1,72))
```

8. preceding: As the preceding axis returns the nodes before the context node in document order excluding descendants, the SQL statements have to explicitly remove descendant nodes which can be done in many different variants. As an example we give an SQL statement with a `NOT IN` clause.

```
SELECT surr FROM typedim
WHERE typedim.surr < contextsurr AND typedim.surr NOT IN
(SELECT surr FROM generateAncestors(contextsurr) )
```

9. following-sibling: The following-sibling axis can be formulated in two versions. Both versions use the fact that the following-sibling has the same parent as the context

node. The first version uses the calculation of the parent with a static length of the surrogates. The query tests the prefix of the context node to ensure that the retrieved compound surrogates have the same parent. For following-siblings it is consequently also required that the compound surrogates are larger than the context node. As the following-siblings all have the same parent the suffix of the compound surrogates equal `0b0`.

```
SELECT surr FROM typedim
WHERE
  typedim.surr SUBRANGE(1, ((FINDBIT(contextsurr, 0)-1)/24)*24) =
  contextsurr SUBRANGE(1, ((FINDBIT(contextsurr, 0)-1)/24)*24)
AND
  typedim.surr SUBRANGE( ((FINDBIT(contextsurr, 0)-1)/24)*24+1,
                        (((FINDBIT(contextsurr, 0)-1)/24)+1)*24) >
  contextsurr SUBRANGE( ((FINDBIT(contextsurr, 0)-1)/24)*24+1,
                        (((FINDBIT(contextsurr, 0)-1)/24)+1)*24)
AND
  typedim.surr SUBRANGE( (((FINDBIT(contextsurr, 0)-1)/24)+1)*24+1,
                        288 ) = 0b0
```

The second version uses the stored parent in the relation.

```
SELECT surr FROM typedim t1, typedim t2
WHERE t1.surr = contextsurr AND t1.parent = t2.parent
AND t2.surr > contextsurr
```

10. preceding-sibling: The preceding-sibling axis is almost identical to the following-sibling axis with the exception that the nodes before the context node have to be returned. Thus the only difference is that the bit string in the `SUBRANGE` is smaller than the `SUBRANGE` of the context node.

```
SELECT surr FROM typedim
WHERE
  typedim.surr SUBRANGE(1, ((FINDBIT(contextsurr, 0)-1)/24)*24) =
  contextsurr SUBRANGE(1, ((FINDBIT(contextsurr, 0)-1)/24)*24)
AND
  typedim.surr SUBRANGE( ((FINDBIT(contextsurr, 0)-1)/24)*24+1,
                        (((FINDBIT(contextsurr, 0)-1)/24)+1)*24) <
  contextsurr SUBRANGE( ((FINDBIT(contextsurr, 0)-1)/24)*24+1,
                        (((FINDBIT(contextsurr, 0)-1)/24)+1)*24)
AND
  typedim.surr SUBRANGE( (((FINDBIT(contextsurr, 0)-1)/24)+1)*24+1,
                        288 ) = 0b0
```

The version which uses the stored parent for every context node is also similar to the statement for the following-sibling and again more compact.

```
SELECT surr FROM typedim t1, typedim t2
WHERE t1.surr = contextsurr AND t1.parent = t2.parent
```

```
    AND t2.surr < contextsurr
```

A combination of axes as it is common for navigation in XML documents consequently leads to a series of self joins on the relation which contains the compound surrogates.

The restriction on prefixes which some axis use either for the child, descendant axis or for the parent relationship can be reformulated with ranges. When only one context node is available the range can be easily formulated with the prefix followed by 0 bits as the lower limit and the prefix followed by 1 bits as the upper limit of the range. When self joins are processed and sets of new context nodes are retrieved this leads to several ranges but omits a static formulation of ranges in the query statement before query execution. In these cases a `SUBRANGE` statement with the calculation of the right most bit is more flexible. In addition an optimizer can exploit this property for a more efficient processing of a `SUBRANGE` statement.

Up to now we have neglected the attribute axis. The tree transformation which is shown for the pre/post mapping in Section 1.3.1 is also valid for MHC. This allows to process the attribute axis for a context node as a combination of the child axis for a context node and an additional filtering step which extracts the attributes (attributes are marked with a flag in the relational schema). As the other navigational axis only return elements it is required that they have an equivalent post filtering step which returns elements only. This fact has been taken care of in the performance evaluation.

3.3.5 Performance Evaluation

Benchmarking Environment

For comparing our proposed MHC mapping and the *pre/post* mapping we perform several measurements with data in different sizes and with different queries.

The data for the measurements is taken from the Xmark benchmark project [SWK+01]. The project allows easy access to data of (theoretically) unrestricted size by providing a generator tool for XML data (XMLgen). The tool allows generation of data based on a given DTD. In our case we have chosen the accompaning DTD of XMLgen which models an Internet auction site. The simplified hierarchical structure of the DTD is shown in Figure 3.5. The complete structure is deeper nested at some nodes and carries optional tags thus making it difficult to display on one page. Table 3.3 shows the used XMLgen factors and the sizes of the resulting XML files. We have generated four different XML documents which differ in size by a factor of ten.

In the following we perform measurements with the following queries, which were also measured in [Gru02]. The exact semantics of the queries are explained in the following paragraphs. The first query navigates from a context node through subtrees to the leaves of the XML tree, while the second query navigates through subtrees to leaves and then considers following-siblings.

```
//open_auction//description
```

XMLgen factor	Nr. of Nodes	Document size (MB)
0.001	2086	0.11
0.01	21051	1.16
0.1	206130	11.6
1.0	2048193	116

Table 3.3: XMLgen factors and resulting document sizes

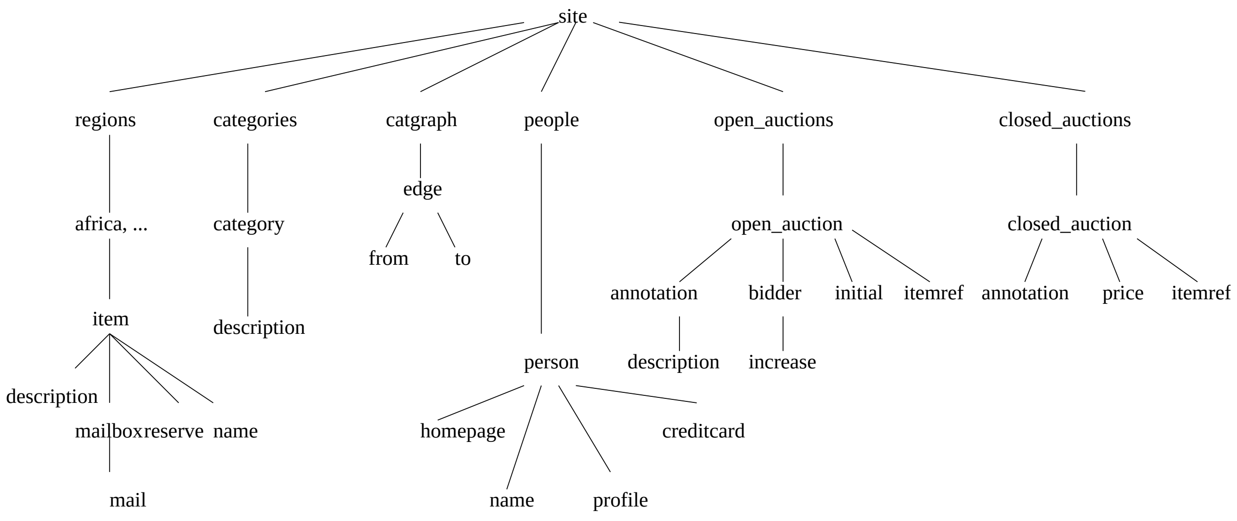


Figure 3.5: Structure of the auction DTD (simplified)

```
//name/following-sibling::*
```

All measurements were performed with the relational database system Transbase[1]. We chose a page size of 2KB and limited the database cache to 128 KB to minimize caching effects. The database system was installed on a Sun Ultra 10 (400MHz, 512MB main memory) and the measurements were performed on a Seagate ST39111A (73.4GB) U160 hard disk.

In the compound surrogate we reserved 6 bits for the surrogate of the tag number and 18 bits for the surrogate of the repetition numbers. The documents generated by XMLgen are all of depth 12 requiring an overall of 288 bits for the compound surrogate.

In total we benchmarked three mappings. Two variants are based on MHC. One variant uses the explicit storage of the parent relationship as described above. This variant is referred to in the following as *MHC_parent*. The compact variant without the explicitly stored parent relationship is simply called *MHC*. The third mapping is the mapping proposed by Grust and is referred to as *pre/post*. Due to the similarity of the *MHC_parent* approach to the *pre/post* approach we have chosen a similiar index support as originally

[1]http://www.transaction.de

	pre/post	MHC_parent	MHC
0.001	140	262	154
0.01	1347	2593	1518
0.1	13060	25324	14820
1.0	129666	251563	147203

Table 3.4: Overall database sizes of different mappings in 2KB pages (including indexes)

proposed by Grust. We have created a primary index on the compound surrogate `surr` and a secondary index on `tag, surr, flag, parent`, as we also consider a restriction on the path attribute `tag` to be very selective (as proposed by Grust). The MHC mapping used a primary index on the compound surrogate and a secondary index on `tag, surr, flag`. The space requirements for the databases can be seen in Table 3.4 which lists the different sizes of the databases in 2KB pages.

Query 1: `//open_auction//description`

The query checks the subtree starting from the root node (in other words the complete document) for an `open_auction` tag and checks for a description tag in every subtree specified by the `open_auction` tag. The number of returned tags varies from 12 tuples for a document size of 0.11 MB to 120000 for a document size of 1.1 GB.

For this query we measured the running time of the query and the number of pages which had to be retrieved from the database to answer the query. Benchmarking running times of queries in general is highly problematic as it is influenced by modern computer architecture like prefetching and caching over several component hierarchies like hard disks, operating systems (including file systems) and applications (database caches). The results in Table 3.5 show an interesting behaviour.

	pre/post	MHC_parent	MHC
0.001	0.050 sec	0.036 sec	0.044 sec
0.01	2.096 sec	0.297 sec	0.124 sec
0.1	221.169 sec	1.218 sec	0.982 sec
1.0	22204.609 sec	12.367 sec	8.826 sec

Table 3.5: Running times of Query 1 for different mappings and database sizes

Apparently the running times of the query for the pre/post mapping suffer from a quadratic factor when processing the query, while the MHC mappings do not as they show a linear behaviour.

The result for the number of retrieved pages is displayed in Figure 3.6, which shows the number of physical page accesses, while Figure 3.7 shows the number of logical page accesses. Both diagrams use a logarithmic scale on both axes and contain the values for all three benchmarked mappings. Physical page accesses occur whenever the database system requests a page from secondary storage, i.e. the page is not available in the database cache. Logical page accesses occur whenever a page is accessed by the database system. One physical page access leads to at least one logical page access. Several logical page accesses occur if the page is accessed multiple times, e.g., if tuples on pages are repeatedly read in different stages of query processing. In these cases no physical access occurs if the page is available in the database cache.

The number of physical page accesses (Figure 3.6) increases linearly for all three mappings. For a very small database size (factor 0.001) the measured page numbers are skewed. This is the result of the small database size, the data distribution (the results are located in different subtrees in the MHC mapping) and the query which touches many pages compared to the overall database size. The sharp increase for the pre/post mapping shows a polynomial behaviour for the number of pages.

The number of logical page accesses shows a linear increase for both MHC mappings. For the pre/post mapping the logical page accesses show a quadratic increase.

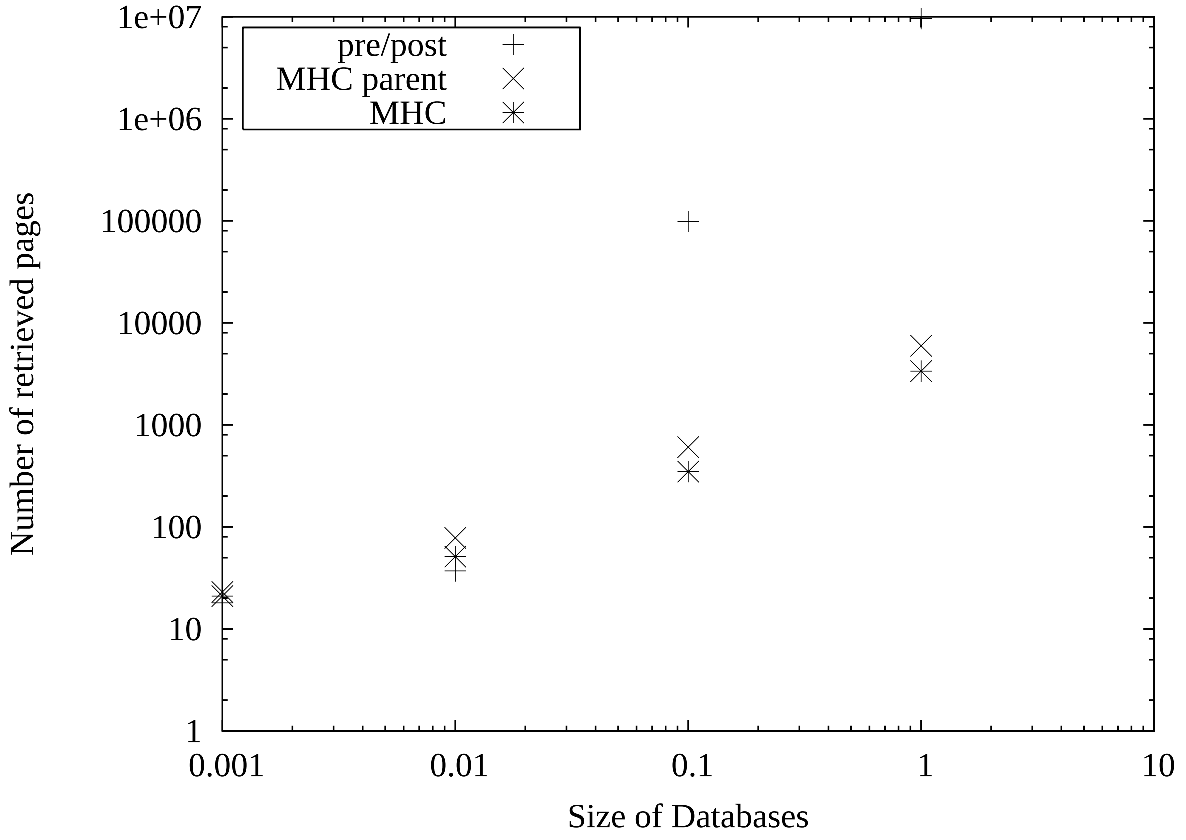


Figure 3.6: Performance comparison of MHC and pre/post mapping (`//open_auction//description`) physical page accesses, log-log scale

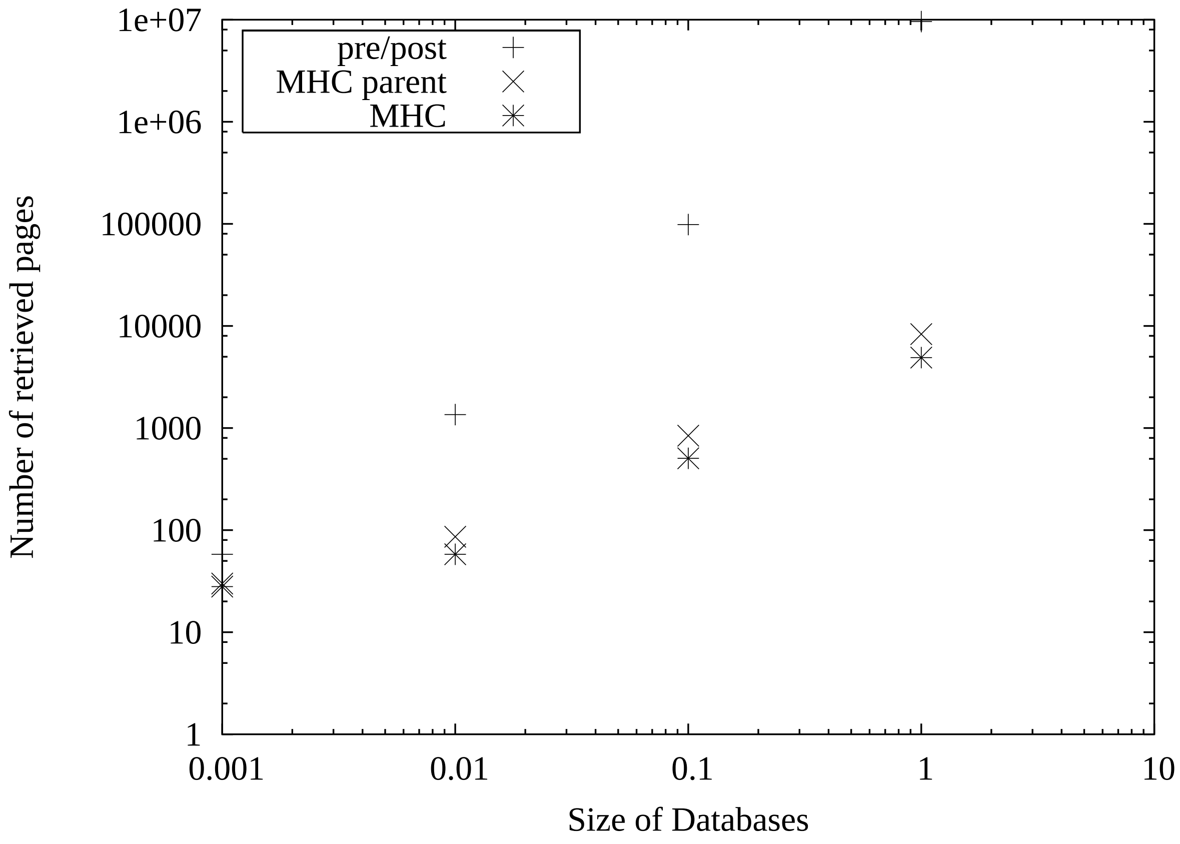


Figure 3.7: Performance comparison of MHC and pre/post mapping (`//open_auction//description`) logical page accesses, log-log scale

This behaviour becomes clearer when taking a closer look on the SQL queries. It is important to note here that both queries use an optimization which takes the tree structure of the data into account. In the case of the pre/post mapping the optimization shrinks the size of the rectangular query boxes (expressed here by the additions on the post and pre values in the query). Without this optimization the mapping would not be useful and the running times and the number of pages would be unacceptable. This optimization is presented already in the original publication on the pre/post mapping [Gru02]. Both variants of the MHC mapping use knowledge about the tree structure as well when using an equi-join on the first 72 bits of the compound surrogate. The 72 bits correspond to the depth of the `description` tag in the document (and in the DTD of course).

For the pre/post mapping the following statement was used:

```
SELECT v2.* FROM grust v1, grust v2 WHERE
v1.tag = 'open_auction' AND v1.flag='E' AND v2.flag='E'
AND v2.tag = 'description' AND v2.pre > v1.pre
AND v2.pre < v1.post+3 AND v2.post < v1.post
AND v2.post > v1.pre-3;
```

For the MHC and MHC_parent the following query was formulated according to our considerations in section 3.3.4.

```
SELECT v2.* FROM typedim v1, typedim v2 WHERE
v1.path='open_auction' AND v2.path = 'description' AND
v1.surr SUBRANGE(1,72) = v2.surr SUBRANGE(1,72);
```

The versions for MHC and MHC_parent are identical though as the query does not need a parent relationship.

The above SQL query for the pre/post mapping is formulated using a semi join. The run-time of the query therefore is strongly dependent on semi-join optimization and processing. The optimizer of Transbase chooses a query plan which results in a nested loop processing of the query.

Query2: `//name/following-sibling::*`

Query 2 returns the following siblings of all `name` tags in the document. This query is of completely different nature than Query 1 as it has to explicitly use the parent relationship of the hierarchical data to identify the following siblings.

For the pre/post mapping the query can be formulated quite easily as the parent is explicitly stored for every node.

```
SELECT v2.tag FROM grust v1, grust v2 WHERE
v1.tag='name' AND v1.parent = v2.parent AND
v1.pre < v2.pre AND v2.flag='E'
```

The query for the MHC_parent mapping is almost identical to the query for the pre/post mapping as the parent relationship is also stored for every node.

```
SELECT v2.path FROM typedim v1, typedim v2 WHERE
v1.path='name' AND v1.parent = v2.parent AND
v1.surr < v2.surr AND v2.flag = 'E';
```

The query for the MHC mapping in contrast is completly different. This is a result of the comparisons for the bit strings and the prefixes of the bit strings as already discussed in section 3.3.4. As the `name` tag occurs in the document at several different hierarchy levels it is not sufficient to check the bit strings with a fixed length prefix but the prefix has to be dynamically calculated for every join surrogate. Transbase uses the very flexible statement `findbit` with the functionality which was described earlier in sections 3.3.3 and 3.3.4. Additionally the following siblings have to be identified with the < relation and the remaining bits have to be cleared (otherwise descendants of the following siblings would also be returned).

```
select v2.path from typedim v1, typedim v2 where
v1.surr subrange(1, ((findbit(v1.surr, 0)-1)/24)*24) =
v2.surr subrange(1, ((findbit(v1.surr, 0)-1)/24)*24) and
v1.surr subrange( ((findbit(v1.surr, 0)-1)/24)*24+1,
                  (((findbit(v1.surr, 0)-1)/24)+1)*24) <
```

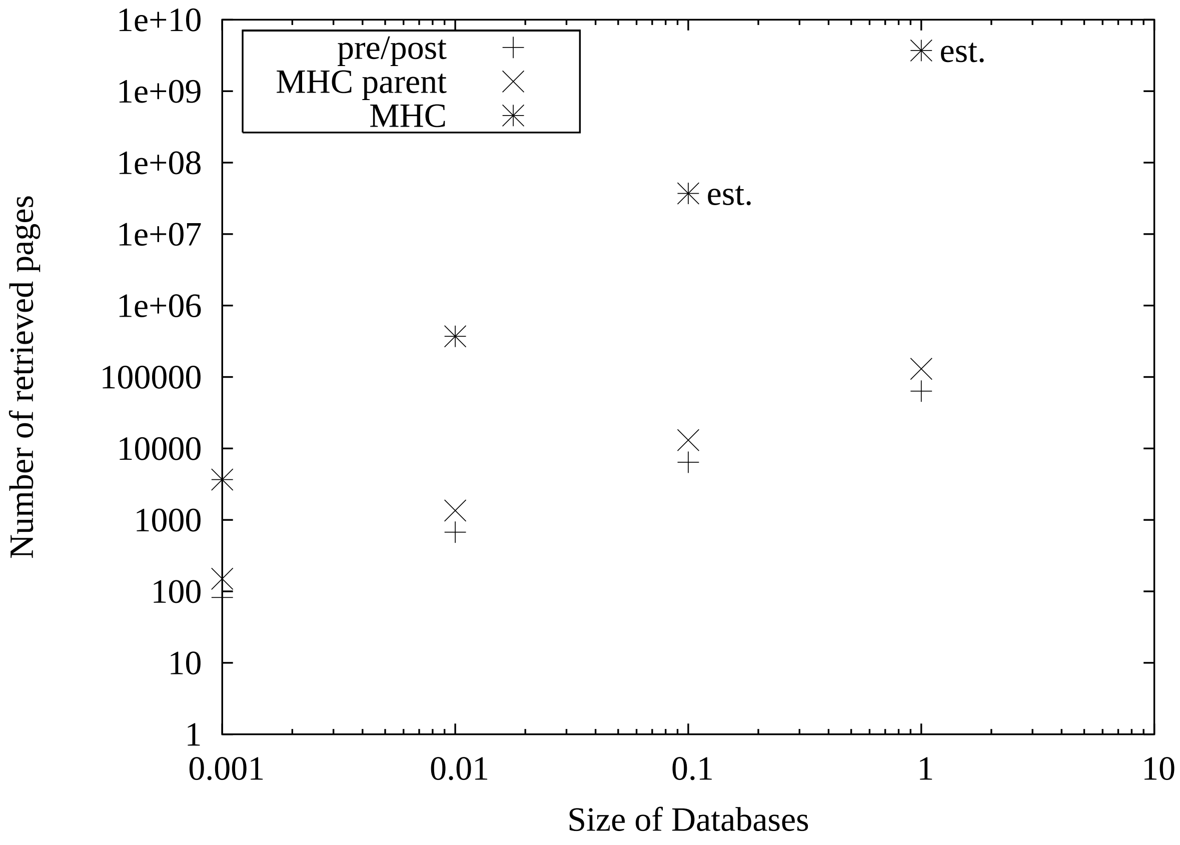


Figure 3.8: Performance comparison of MHC and pre/post mapping (`//name/following-sibling::*`) physical accesses, log-log scale

```
v2.surr  subrange( ((findbit(v1.surr, 0)−1)/24)∗24+1,
                     (((findbit(v1.surr, 0)−1)/24)+1)∗24) and
v2.surr  subrange( (((findbit(v1.surr, 0)−1)/24)+1)∗24+1, 288)
          = 0b0
and v1.path='name' and v2.flag='E';
```

Table 3.6 shows the runnning times for Query 2. The two mappings pre/post and MHC_parent show a similar behaviour where the time increases linearly. The MHC mapping behaves different for this query as it shows a polynomial increase for the query time. The increase is so significant that it was not possible to measure the query for small XMLgen factors above 0.01. In Table 3.6 these results were replaced with estimated running times (marked with *est.* in the table cells).

Figures 3.8 and 3.9 show the number of physical and logical page accesses for all three mappings. Again the same results occur as for the time benchmarks. This effect occurs because the query is processed by calculating the cross product of the two typedim tables and then performing a postfiltering for all tuples. Only a restriction on the tag name is

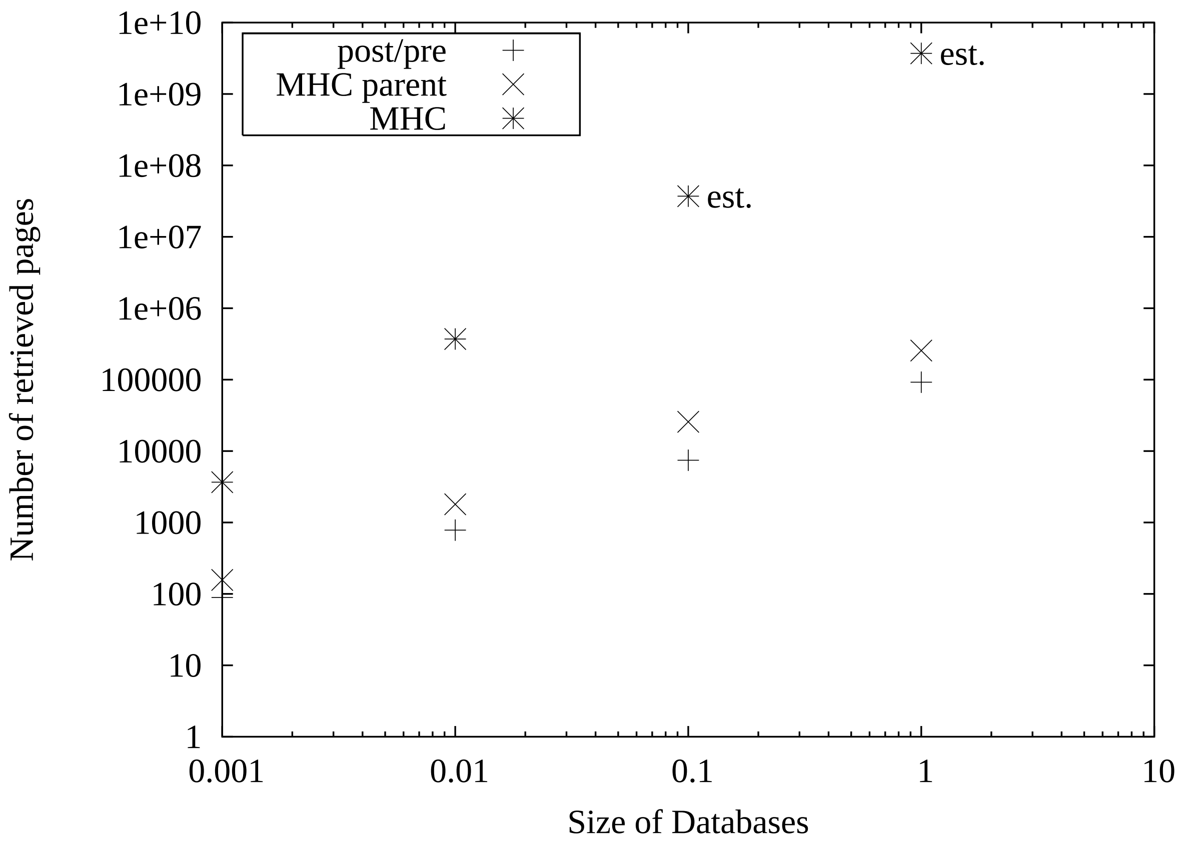


Figure 3.9: Performance comparison of MHC and pre/post mapping (`//name/following-sibling::*`) logical accesses

	pre/post	MHC_parent	MHC
0.001	0.185 sec	0.235 sec	11.463 sec
0.01	1.020 sec	1.775 sec	1115.068 sec
0.1	8.812 sec	21.482 sec	111500 sec (est.)
1.0	102.654 sec	208.383 sec	11150000 sec (est.)

Table 3.6: Running times of Query 2 for different mappings and database sizes, log-log scale

used for one table.

The two diagrams show a special relation between the pre/post mapping and the MHC_parent mapping. Although the two queries are equivalent and also processed equivalently (except for the different datatypes) MHC_parent requires more page accesses than pre/post. This difference is also visible in Table 3.6, which shows the running times, and is a result of the significantly different tuple sizes for the different mappings. A closer look reveals a size requirement of 12 (4+4+4) bytes for the three attributes (pre, post and parent) in the pre/post mapping. The MHC_parent mapping requires 36 bytes (288 bits) for each of the attributes resulting in an overall size requirement of 72 bytes. This great difference results in a smaller page utilization for the MHC_parent mapping and a larger overall database size which leads to the above described effects as more pages have to be retrieved for the same query result.

3.4 Summary

In this chapter we have introduced a numbering scheme based on MHC to store XML data in relational database systems. We have seen that processing XPath axes on our implementation can be performed. We have shown an implementation for the numbering scheme based on bitstrings. For some of the axes knowledge about the underlying hierarchy (DTD) is required.

In the comparision with the pre/post mapping suggested by Grust we have seen that both mappings require special features in the optimizers of relational database systems. The pre/post mapping requires an efficient processing of semi joins while in contrast our proposed MHC mapping requires the efficient processing of prefixes in bitstrings and the ability to exploit that prefixes can be transformed into ranges on our bit representation. For some axes (parent, ancestor) our implementation requires User Defined Functions but the calculation can be solely performed in main memory. The processing of XPath axes in our implementation becomes significantly easier when the parent node is additionally stored for every node in the document. This requires additional space by an approx. factor of 2. The overall space requirements for the MHC mapping is higher than for the pre/post mapping as the length of compound surrogates depends on the depth of the document and the number of nodes in each level while pre/post scales with the number of nodes in the document.

In the following chapter we will discuss the storage of several XML documents and the processing of queries on the stored documents.

Chapter 4

Multidimensional Mapping and Indexing of XML

4.1 Introduction

In Chapter 3 the idea of storing XML with the help of Multidimensional Hierarchical Clustering (MHC) is intensively discussed in the aspect of navigation. In this chapter we do not concentrate on navigation alone as we did before but we use the already presented techniques (MHC) to show how to store XML data in relational database management systems and how to query the stored data. The largest difference to the previous chapter is the additional consideration of data. We also do not restrict storage to one document but propose a solution for the storage of several documents following one DTD.

The innovation presented in this chapter is the multidimensional approach to store XML (Section 4.2). We present an implementation of the model in Section 4.3. A multidimensional model apparently motivates the use of multidimensional index structures to speed up query processing. An intensive discussion of the impact of several multidimensional index structures (UB-Tree and three variants of B-Tree compound indexes) can be found in Section 4.4. In this section the multidimensional mapping is also compared to the edge mapping approach.

4.2 A Multidimensional Model to Store XML in Relational Database Systems

Paths are a fundamental feature of XML. Many approaches to speed up queries therefore concentrate on the efficient indexing of paths and path expressions. When discussing paths it is also important to note the order of paths which is inherent in the XML documents. The ordering of XML documents is a very important aspect especially in the case of indexing. The DTD of an XML document already defines the ordering of the tags in the document. This is especially important when tags with the same substructure occur several times but with different data. This can be seen from example 4.1, which has also been used in Chapter 3.

Example 4.1

```
<author><FN>Michael</FN><LN>Bauer</LN></author>
<author><FN>Rudolf</FN><LN>Bayer</LN></author>
```

Order is also important for query languages. When the above example (a fragment from a larger document) is queried with the predicate **author/FN = Michael and author/LN = Bayer** the semantics of the predicate is not clear at first hand. In the document fragment the paths to the values are the same but the structure and the order of the paths is of great importance as it acts as a grouping feature that is very relevant for queries. The above predicate can be reformulated so that the semantics is clear and the two parts of the query have to be valid in the same subtree. A correct version of the above predicate in XPath returning the first line from the above fragment is formulated as **author[FN = "Michael" and LN = "Bauer"]**. To be able to evaluate such a predicate it is apparently necessary to preserve the document structure for the stored document.

Finally ordering might have a severe impact on the performance of answering queries. In every database instance the data is stored in a certain physical order on external storage devices. Clustering the stored data in document order usually can be achieved, nevertheless an order independent of the document order might be more useful for certain queries. Nevertheless, the original order of the stored document has to be preserved in some way, otherwise it is impossible to reconstruct the document in the same order as it was originally available.

In the following we propose three building blocks of XML documents.

Paths: The notation of paths is a basic concept of XML and query languages for XML. Due to the graph-like nature of XML it is necessary to preserve path information and order for query processing.

Values: We define values as the content of XML tags. Our proposed scheme can deal with both, data-centric and document-centric XML. Attributes in XML are modelled by using the @notation in the paths as known from XPath.

Document Identifiers: Document identifiers group paths for one document and are the results in the above mentioned selection. We assume that this identifier is available from the XML data itself. If this is not the case it can be easily computed when the document is processed before it is inserted into the database.

The three building blocks of XML documents form a three-dimensional cube. In this three-dimensional model we can now easily answer queries for both the selection and the projection problem as follows.

Both selection and projection restrict the three-dimensional universe in two dimensions. The input for evaluation of the XPath predicate of the selection is one (or more) path expressions and one (or more) values which correspond to the path expression and form a predicate. The output of the selection is a set of document identifiers which match path expression(s) and the value(s).

In the projection the document identifiers and the path expressions are the input of the query. The results are values. For better readability and post-processing capabilities the document identifiers as well as the output paths are sometimes returned as well.

4.3 Implementation

4.3.1 The Schema

The implementation of our mapping scheme is based on two relations. The core is a table with three attributes, which we refer to as **xmltriple**. The table **xmltriple** holds the attributes did, val, and surr (see Table 4.1 for an example with two tuples and 4 bits per surrogate). For each value in a document, **xmltriple** stores the corresponding path information as a compound surrogate and the document, which contains this value, as a document id.

For mapping XML document paths to compound surrogates we use an additional table **typedim** with the two attributes path and surr (Table 4.2). The table does not contain any information about paths on a per document basis but only stores complete paths to leafs. We already mentioned this optimization in the previous chapter, which reduces the size of **typedim** significantly, so **typedim** is typically very small compared to **xmltriple**.

`did`	`val`	`surr`
1	Rudolf	0b0010 0001 0001 0001 0000 0000 0000 0000
1	Bayer	0b0010 0001 0010 0001 0000 0000 0000 0000

Table 4.1: Relation `xmltriple`

`surr`	`path`
0b0010 0001 0001 0001 0000 0000 0000 0000	/Author[1]/FN[1]
0b0010 0001 0010 0001 0000 0000 0000 0000	/Author[1]/LN[1]

Table 4.2: Relation `typedim`

4.3.2 The Query-Rewriting

XML Queries on XML documents have to be rewritten to SQL so that RDBMS are able to process them. The method for our mapping is shown in Figure 4.1.

$$\begin{tikzcd}
\text{SP} \arrow[d] & & & & \text{RP} \arrow[d] & \\
\text{MHC(SP)} \arrow[dr] & \text{SV} \arrow[d] & & & \text{MHC(RP)} \arrow[ddll] \arrow[dr] & \\
& \text{SDID} \arrow[dr] & & & & \text{RPaths} \arrow[ddll] \\
& & \text{RV} \arrow[dr] & & & \\
& & & \text{Query−Result} & &
\end{tikzcd}$$

Figure 4.1: Steps to rewrite the query into SQL

To illustrate the rewriting process we have chosen an example query from a typical library scenario. The query returns the title and keywords of all scientific papers that were written by a certain author. The same query will later be used in our measurements.

Example 4.2

```
for $b in document("http://www.in.tum.de")//paper
where $b/author[FN = "Michael" and LN = "Bauer]
return
  $b/title
  $b/keyword
```

The initial query is separated into the already mentioned two components *selection* and *projection*. The selection consists of a selection path (SP) and selection value (SV), which can be seen in the left branches of the diagram in Figure 4.1. The paths are mapped to compound surrogates by a lookup in the **typedim** table (MHC(SP)). The compound surrogates and search values are then used to identify the document ids which satisfy the selection predicate (SDID) by querying the **xmltriple** relation.

The projection, which outputs the result values (RV), is processed similarly. The result paths (RP) are transformed into compound surrogates. In the next step the result values are returned by using the retrieved document ids and the compound surrogates as input. In the final step, the compound surrogates of the results have to be replaced with the corresponding paths (RPaths). More complex queries might require recursive application of this schema.

As we are examining a relational mapping of XML data the queries of each step have to be rewritten to SQL statements. Some of the statements depend on the output of previous queries. This may lead to long statements (especially for many hits in the document ids) and for ease of presentation we only give very short example statements. It would of course be possible to rewrite the query into one large SQL statement, but this would hide the diversity of the query parts and significant facts about the nature of XML queries.

Example 4.3 *1. Step: MHC(SP)*

```
select surr from typedim
    where attr like 'Author[%]/FN[1]' order by surr;
select surr from typedim
    where attr like 'Author[%]/LN[1]' order by surr;
```

2. Step: SDID

```
select distinct x1.did from xmltriple x1, xmltriple x2
    where
    (x1.surr = 0b00100001000100010000000000000000
    or x1.surr = 0b00100010000100010000000000000000
    ...............    result of MHC(SP)
    or x1.surr = 0b00101000000100010000000000000000)
    and
    x1.val = 'Michael'
    and
    (x2.surr = 0b00100001001000010000000000000000
    or x2.surr = 0b00100010001000010000000000000000
    or .............   result of MHC(SP)
    or x2.surr = 0b00101000001000010000000000000000)
    and
    x2.val = 'Bauer'
    and
    x1.did = x2.did
    order by x1.did
```

3. Step: MHC(RP)

```
select surr from typedim
    where attr like 'Title[1]'
       or attr like 'Keyword[%]' order by surr;
```

4. Step: RV

```
select did,val,attr,typedim.surr from xmltriple,typedim
    where
    (typedim.surr = 0b00010001000000000000000000000000
    or
    typedim.surr = 0b00110001000000000000000000000000
    or .........  result of MHC(RP)
    or
    typedim.surr = 0b00111000000000000000000000000000)
    and ( did = 76 or .....  result of SDid
    or did = 9783 )
    and typedim.surr = xmltriple.surr
    order by did,typedim.surr
```

4.4 Measurements

4.4.1 The Data and Measurement Environment

We have chosen a DTD from a digital library scenario for our measurements. The DTD of the documents is a typical specification for scientific papers. We generated 10000 different documents using the XMLGenerator tool from IBM Alphaworks [XMLc]. The raw size of the XML data is approx. 50 MB. We have used a similar distribution [CSF$^+$01] as in the DBLP database [DBL] for authors which results in approx. 400 different authors for 10000 documents. From these documents we generated flat files for bulk loading the databases. The sizes of the database are approx. 25 MB and they differ slightly depending on the used index. For our measurements we ran the already above mentioned query on the different indexed tables.

All measurements were performed with the relational database system TransBase[1], which won the European Information Technology Prize 2001 for its pioneering implementation of UB-Tree indexes. We chose a page size of 2KB and limited the database cache to 128 KB. With a cache size this small only few pages can be kept in the cache. Completely eliminating the cache would severely decrease performance as even index pages would no longer reside in the cache.

[1]http://www.transaction.de

The database system was installed on a Sun Ultra 10 (400MHz, 512MB main memory) and the measurements were performed on a Seagate ST39111A (73.4GB) U160 hard disk.

In the following we use abbreviations to denote the different indexing methods.

- *UB-Tree* denotes the indexing with a three-dimensional UB-Tree, the index attributes are *did, surr, value.*
- *DidSurr* denotes indexing with a compound B-tree with the index attributes *did* and *surr* (in this order),
- *DidSurr_Val* adds an additional secondary B-Tree index on *val.*
- *SurrValDid* denotes the use of the compound B-tree with the index attributes *surr*, *val*, and *did.*

In addition, we measured two variants of the Edge Mapping approach. Edge Mapping is introduced in Section 1.3.1 and further discussed in Section 4.4.4.

We rewrote the query from Section 4.3.2 to SQL as shown above and measured both, the elapsed time and the number of pages that were accessed for answering the SQL queries. The number of retrieved pages is further divided into the overall number of physical page accesses, logical accesses to the index pages, and logical accesses to the data pages.

The same explanation which was already given in Chapter 3 is of course also valid in this context. The physical page accesses occur whenever the database system requests a page from secondary storage, i.e. the page is not available from database cache. Logical page accesses occur whenever a page is accessed by the database system. One physical page access leads to at least one logical page access. Several logical page accesses occur if the page is accessed several times, e.g., if tuples on pages are repeatedly read in different stages of query processing. In these cases no physical access occurs if the page is available in the database cache.

We analyze selection and projection separately and for our example data set (10000 documents) the following numbers of tuples were returned for each processing step (Table 4.3).

Query Step	Selection (SDID)	Projection (RV)
Number of Tuples	104	554

Table 4.3: Number of tuples returned for each processing step

4.4.2 Selection

As noted above, we have separated the queries into a selection and projection part. The selection restricts values and compound surrogates and outputs a set of document ids. The set of document ids is then further processed in the projection.

The selection query is graphically illustrated in Figure 4.2 (due to reasons of visibility the figure only shows 8 point restrictions). The query cuts through the cube as the two dimensions are restricted, while the third is variable. The query is located on two parallel planes orthogonal to the value dimension and is processed using 16 point restrictions (one for each surrogate and value restriction). The compound surrogates are dense in their dimension.

UB-Tree

For the UB-Tree the results of this query are located on the same page with a high propability due to the space-filling Z-curve. As the query is processed by repeatingly stabbing through the three-dimensional space pages are read severaly times; caching can be used in this situation to speed up query processing. In this case the caching is intra-query as the query processing itself benefits from the reuse of pages that were already read from secondary storage at earlier stages of query processing.

The consequence is that the average time per page sharply decreases (0.3 ms/physical page), as the page is processed only in memory. This is shown by the measurements in Table 4.4 and Table 4.5.

Index	log. Idxp.	log. Datp.	phys. Pages	DB size (pages)
UB-Tree	918	782	693	20428
DidSurr	213	11946	11957	14426
SurrValDid	39	22	36	12132
DidSurr_Val	215	705	1208	18656
Edge_Compound	386566	241118	49776	41585
Edge_Secondary	5689	7473	6526	93999

Table 4.4: Page numbers for the selection

Compound B-Tree *DidSurr*

For a compound B-Tree on the attributes *did* and *surr* the scenario is completely different compared to the UB-Tree. The query processor of the database system cannot use any restriction on the *did* attribute, so it has to start with the smallest value for the compound surrogate dimension, starts to read all data pages and performs a post-filtering. Table 4.4 shows that the query reads almost all pages of the database. The query is slowest among all the others (Table 4.5), still it achieves a high page rate (1.7 ms/physical page). This hints that the query is highly supported from caching. This time it is not intra-query caching as with the UB-Tree, but caching from the operating system.

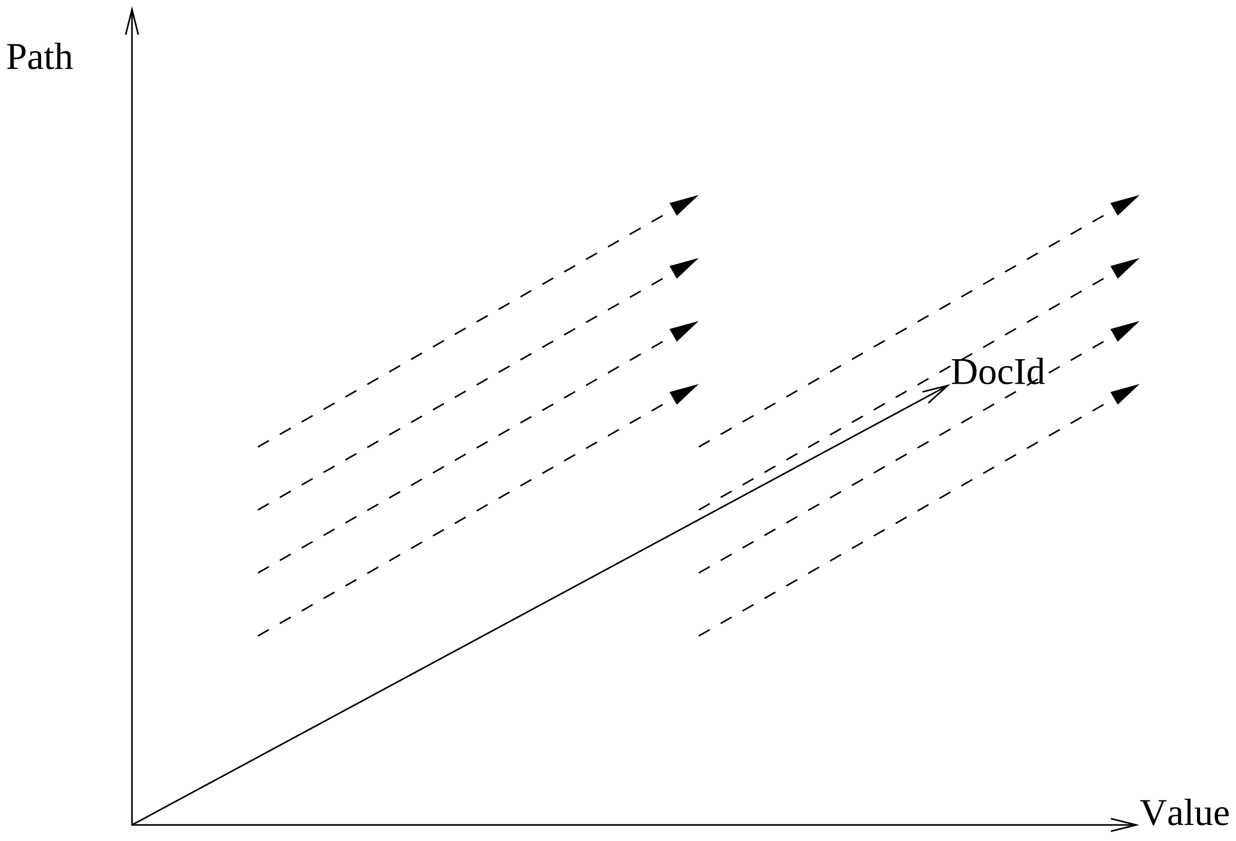


Figure 4.2: Point restrictions for selection query

Compound B-Tree *SurrValDid*

In our third measurement the selection is performed with **xmltriple** indexed with a compound B-Tree on the compound surrogate attribute, the value attribute, and the document id attribute. This index exactly supports the restricitions of the query. The number of physical pages read is only 5% of the physical pages read for the UB-Tree (Table 4.4).

Compound B-Tree *DidSurr* with secondary index *Val*

A closer look on the restrictions of the selection query and on the result for the three indexes discussed above, show that it is important for the performance of the index to directly support a restriction on the value attribute. Depending on the data, especially if value restricts stronger than structure (the structure here is represented by the surr attribute), a secondary index on the value attribute should improve the scenario for the *DidSurr* index. The creation of the secondary index results in an increase of the database size by approx. 30% but is still below the size of the UB-Tree index. Our measurements show that both, the elapsed time and the number of retrieved pages are reduced as expected (Table 4.5 and Table 4.4) although the performance of the *SurrValDid* index is of course

Index	Selection	Projection	Total
UB-Tree	0.26s	8.73s	8.99s
DidSurr	20.3s	1.96s	22.3s
SurrValDid	0.02s	9.18s	9.20s
DidSurr_Val	0.95s	2.16s	3.11s
Edge_Compound	60.3s	0.58s	60.9s
Edge_Secondary	6.65s	2.50s	9.15s
Tupel	104 (DocIds)	554 (Tags)	

Table 4.5: Running times for selection and projection queries

not reached.

4.4.3 Projection

The projection query outputs values and restricts the document ids and the compound surrogates. The restrictions are no range restrictions but point restrictions. The cardinality of the set of points in the document id dimension depends on the result set of the selection query. It is important to note that the result set of the selection query is usually not a range (a range would be quite unlikely). The compound surrogates in the other dimension are restricted to 9 point values (one compound surrogate for title and 8 compound surrogates for keywords). All values which answer the query are consequently located on $9 \times 104 = 936$ parallel straight lines intersecting the three-dimensional space. Figure 4.3 sketches the scenario with three lines due to the sake of clarity.

UB-Tree

Since the straight lines completely stab through the universe (there is no restriction in the value dimension) we can deduce some more information about the processed data by calculating the expected number of page accesses. According to Table 4.6 the size of the database is 20428 pages. This results in approx. $2^{15} = 2^{3*5}$ pages. This leads to approx. $2^5 = 32$ pages in each of the three dimensions meaning that each of the 936 straight lines touches 32 pages. This sums up to 29952 logical data page accesses.

Figure 4.6 presents the page numbers for the projection query from our performance tests. The accessed logical data pages are close to the results calculated above. They are not exactly the same because we have assumed a uniform distribution of the data. This is not the case for the data we used.

Another drawback for the UB-Tree in this measurement is the very high number of physical accesses to the pages. The numbers show that for our simple example query almost one third of the database is read (6441 pages of 20428 pages) although the result set is very small. Most of these physical page accesses are data pages as only 1% of the overall pages in the database are index pages. The high number of physical page acccesses is a combination of the way the query is processed (as explained above), the distribution of the selection results which are not ranges but spread over the *docid* dimension and the clustering of the UB-Tree which does not favour one or two attributes but treats all attributes in an equal manner.

Index	log. Idxp.	log. Datp.	phys. Pages	DB size
UB-Tree	39700	31043	6441	20428
DidSurr	384	436	305	14426
SurrValDid	6	463	468	12132
DidSurr_Val	384	436	305	18656
Edge_Compound	374	517	508	41585
Edge_Secondary	2278	2488	1353	93999

Table 4.6: Page numbers for the projection

Compound B-Tree *DidSurr*

The compound index on the attributes *did* and *surr* is the fastest index for projection as the query restricts exactly the index attributes to points. As there are more dids than compound surrogates there are more index pages read than for the *SurrValDid* index.

Compound B-Tree *SurrValDid*

Indexing the **xmltriple** relation with a compound B-Tree (index attributes *surr*, *value*, *did*) leads to low page numbers in comparison to the UB-Tree. In constrast the running time of the query is almost the same as for the UB-Tree. A closer look on the way the query is processed reveals that both numbers are reasonable. The projection query restricts on the surrogates and on *did* and there is no restriction on the values. The system starts with reading the data pages starting with the smallest value for value and *did* until it terminates when the surrogate range is processed for the keyword surrogates. The system accesses the B-Tree a second time for the title surrogat. The results of the projection query are determined by post filtering the retrieved pages. The retrieved 463 data pages carry approx. $463 * 100 = 46300$ tuples. The post filtering has to be done for each of the 104 tuples that result from the selection. This leads to $104 * 46300 = 4815200$ overall comparisons. The observation shows that the projection for *SurrValDid* is not I/O bound but CPU bound.

Compound B-Tree *DidSurr* with secondary index *Val*

In the projection part of the query there is no restriction on the value attribute and consequently the secondary index can not be used by the query optimizer. The resulting measurements are therefore the same as for the *DidSurr* index.

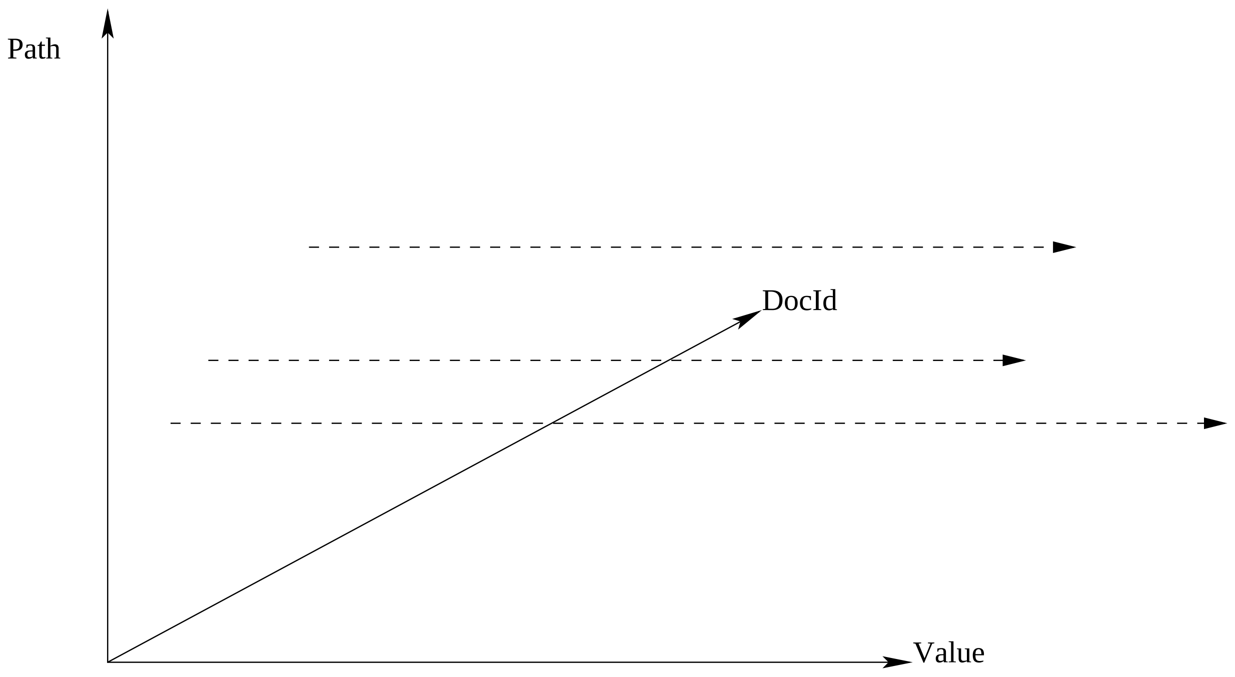


Figure 4.3: Point restriction for projection query

4.4.4 Comparison with Edge Mapping

A common way to estimate the performance of a mapping scheme is a comparison with the Edge Mapping for XML data[FK99]. We use the slightly modified version of the Edge Mapping from [CSF$^+$01].

For our measurements we indexed the root table with a primary compound B-Tree on the attributes (*parentid*, *childid*) (*Edge_Compound* in Tables 4.4-4.6). In a second variant we created secondary indexes on each of the attributes *parentid*, *childid*, and *label* (*Edge_Secondary* in Tables 4.4-4.6). The second variant comes closest to the measurements in [CSF$^+$01].

As for the previous mapping we also examine selection and projection separatly. Since Edge Mapping explicitly stores the graph structure the space requirements for the *edge* table are much higher than for our proposed XML mapping in which we only store the paths from the root to leafs (Table 4.4).

Following paths in the *edge* table leads to a long series of self joins (depending on the length of the path) in the rewritten SQL queries.

In the first variant the selection is very slow since no restrictions can be used on any index attributes (60.2s) and the RDBMS performs a full table scan. The scenario is different for the projection. Here the Edge Mapping is the fastest among our measurements. The restriction on the 104 document ids for the projection leads to a strong restriction on the *parentid* attribute and for every self join the intermediate results are further reduced. In addition the projection query is heavily supported by intra-query caching effects as intermediate results that were already processed for the self-joins can be reused for later stages of the query.

The second variant (which uses secondary indexes on *parentid*, *childid* and *label*) reaches a remarkable database size of 93999 pages (almost half of this size is occupied by the secondary indexes). This is almost 7 times the size of the smallest database size for our proposed multidimensional mapping and is larger than the original raw XML data. Using a hand-tuned query plan we could lower the selection query to 6.65s. We hand-tuned the plan as the original plan used non-optimal join sequences which lead to an original running time of >300s. The projection query is also hand-tuned (otherwise >145s). It is now among the fastest query as not many self-joins have to be performed to reassemble the paths (we only query for title and keywords which are toplevel tags). In addition the restrictions on surrogates and document ids are also very well supported by the secondary indexes, but the overall running time of selection and projection (9.15s) are severly declined by the tremendous database size.

4.5 Measurements with DBLP Data

Index	log. Idxp.	log. Datp.	phys. Pages	DB size (2KB pages)
UB-Tree	15238	13043	15476	93283
DidSurr	3	80403	80537	87895
SurrValDid	73	51	84	57071
DidSurr_Val	6	65	144	127512
Edge_Compound	113786	310987	211965	125474
Edge_Secondary	453	470	535	274529

Table 4.7: Page numbers for the selection (DBLP data)

Besides the measurements with generated data we also want to confirm our results with real world data. The DBLP database [DBL], a very well maintained collection of computer science bibliography, for some time now offers its content as XML documents. Every publication in the database is available as a single XML document. Our snapshot already contained more than 270,000 different documents, leading to approx. 280 MB of data, while the data in DBLP currently is still growing.

The characteristics of the data strongly differs from the generated data we used previously. In DBLP the documents are grouped into several main classes (books, journals, conferences, etc.) which actually results in several types of documents (while we had only one document type `<paper>` in the generated data). In addition the data is "flatter" (the structure is not as deep as in the case of our generated data). Example 4.4 shows a sample DBLP XML document.

Example 4.4

```
<?xml version="1.0"?>
<!DOCTYPE dblp SYSTEM "dblp.dtd">
<dblp>
<article key="journals/cacm/Knuth74">
<author>Donald E. Knuth</author>
<title>Computer Programming as an Art.</title>
<pages>667-673</pages>
<year>1974</year>
<volume>17</volume>
<journal>CACM</journal>
<number>12</number>
<url>db/journals/cacm/cacm17.html#Knuth74</url>
</article>
</dblp>
```

We measured a query which at first glance looks very similar to our previous one.

```
for $b in
  document("http://dblp.uni-trier.de/")//dblp
where $b//author = "John Smith"
return
 <result>
  $b//title
  </result>
```

Due to the flat structure of the DBLP data the selection and the projection parts are equivalent to `/*/*/title`, and `/*/*/author = "John Smith"` respectively. This is important to note as `//author` in our generated data would also refer to the authors in the `<bibrec>` subtree.

In the following section we will revisit the index structures we have already discussed in Section 4.4. Due to the sake of readability we will omit a detailed discussion for each index structure but will only point out new insights.

4.5.1 Selection for DBLP Data

It is apparent that there are more significant gaps between the different index structures. In the case of *DidSurr* it becomes clear that the database system has to perform an index scan (almost the complete database is read) followed by a postfiltering stage to retrieve the correct result tuples (Table 4.7). In the case of the *SurrValDid* index it is astounding that the number of logical datapage accesses is only 51 although we have to retrieve 65 document identifiers and have a restriction on 112 compound surrogates (due to the various occurences of a possible author tag in journal, books, etc. including their repetition numbers). The sorted order of compound surrogates is matched against the tuples on the page. The page tuples are again sorted by compound surrogates due to the index. Because of the author distribution in the DBLP data there is only a low probability that there are more than eight authors and the branching of the index collapses. The number of retrieved pages is then further reduced as a comparison of the last tuple on the page allows to jump to a non adjacent page defined by the next compound surrogate from the search predicate. This fact enables the index to physically retrieve even less pages than the *DidSurr_Val* index.

The running times for the *Edge_compound* variant are again very high due to the bad index support. The *Edge_secondary* variant again proves its speed but suffers from the tremendous size of the secondary indexes. This leads to an overall database size which is two times larger than the original data.

Index	Selection	Projection	Total
UB-Tree	18.24s	35.66s	53.9s
DidSurr	583.7s	0.77s	584.5s
SurrValDid	0.25s	31.45s	31.7s
DidSurr_Val	0.57s	0.72s	1.29s
Edge_Compound	1118.6s	0.96s	1119.5s
Edge_Secondary	3.17s	1.06s	4.23s
Tupel	65 (DocIds)	65 (Tags)	

Table 4.8: Running times for selection and projection queries (DBLP data)

4.5.2 Projection for DBLP Data

Index	log. Idxp.	log. Datp.	phys. Pages	DB size (2KB pages)
UB-Tree	29380	21308	22903	93283
DidSurr	139	79	152	87895
SurrValDid	32	11809	11837	57071
DidSurr_Val	139	79	152	127512
Edge_Compound	448	253	232	125474
Edge_Secondary	902	671	508	274529

Table 4.9: Page numbers for the projection (DBLP data)

The projection also shows a similar scenario for the DBLP data as for the generated data. The above mentioned characteristics of the DBLP data in combination with the query statement influences the query execution especially for the UB-Tree. The `//title` in the query statement leads to compound surrogates even for sections in the DBLP where no corresponding document identifiers from the selection step are located (thesis, www, etc.). This unneccessarily increases the number of pages accesses. An automatic optimization would require knowledge about the semantics of the data, which we did not consider as an option in this context.

The UB-Tree database size is between 2^{16} and 2^{17} pages (Table 4.9), which results in between 2^5 and 2^6 pages in one dimension. A calculation assuming 32 and 64 pages in every dimension, 9 compound surrogates (for the various occurences of title) and 65 document identifiers, which were hit in the selection, leads to $32 \times 9 \times 65 = 18720$ and $64 \times 9 \times 65 = 37440$ pages. In this range the actual numbers of logical data page accesses is located as can be seen from Table 4.9. The page numbers for the indexes *DidSurr* and *DidSurr_Val* are identical as expected. Both indexes support exactly the restrictions of the projection query.

Figure 4.4 presents the dependancy of overall query running times (selection and projection) and database size for the different indexing methods for DBLP data. The time scale is logarithmic as the discrepancy between *Edge_compound* and *DidSurr_Val* would severly degrade the illustration. The vertical line marks the size of the XML raw data.

4.5.3 Reconstruction of Documents

For the DBLP data set we also investigated the reconstruction performance of the different indexing approaches. Table 4.10 shows the running time to retrieve a complete document. As expected, the DidSurr variants are the fastest as they store the documents basically in the original order. The multidimensional index, however, requires more time (and I/O) as the three-dimensional clustering also includes the value dimension, which spoils the compact storage of a single document. This is even more the case for SurrValDid where the paths of one document are completely scattered. What is surprising at first glance is

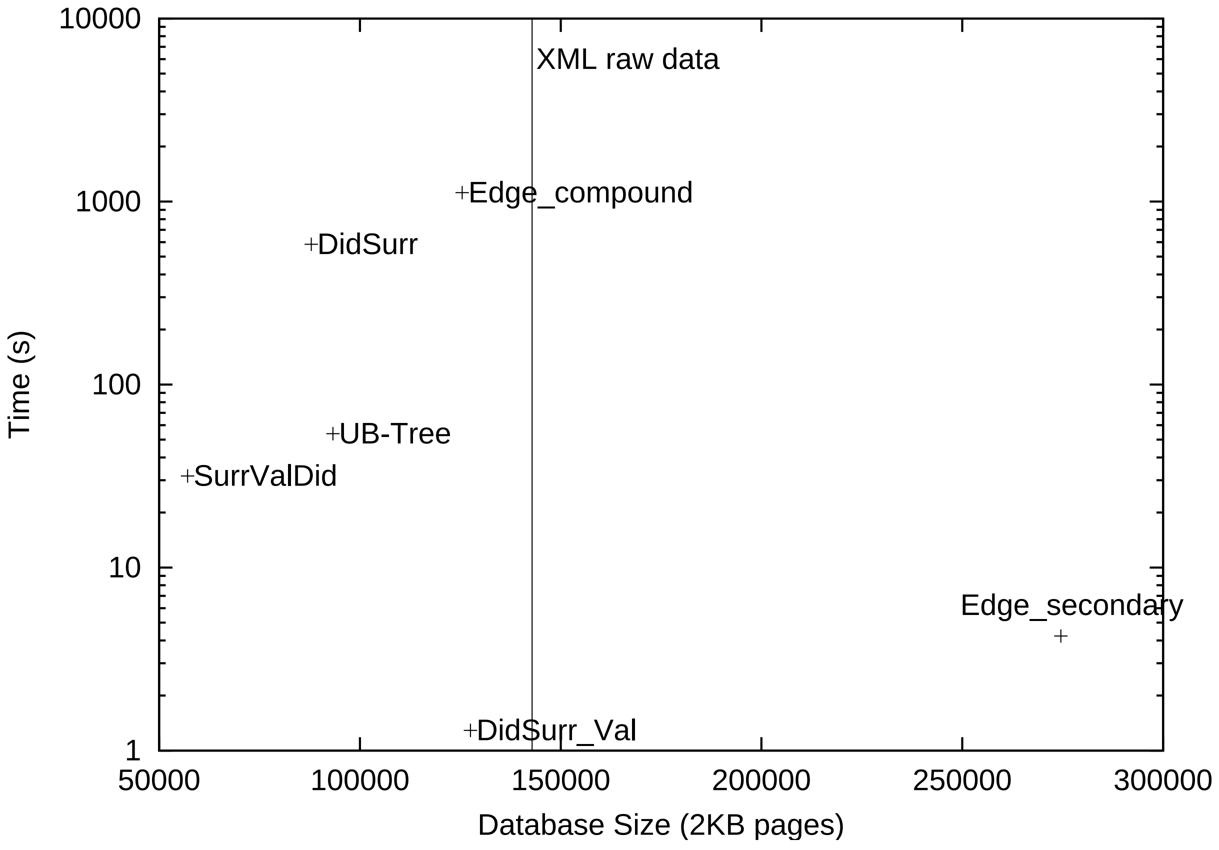


Figure 4.4: Dependency of query running time and database size for DBLP data

the performance of the Edge Mapping. The complete shredding of the document does not have an influence on the reconstruction time. The explanation of this behavior lies in the structure of the DBLP documents and in the storage scheme. In our case, the identifiers of the nodes are consecutive within one document leading to a clustered organization. Consequently, the required I/O to retrieve the documents for Edge Mapping is the same as for DidSurr. What is considered to be costly for Edge Mapping is the reconstruction of paths. However, in the case of DBLP we have very flat documents so that this aspect does not show up here. We expect that the reconstruction performance for Edge Mapping will decrease for deeper and bushier documents.

4.6 Summary

In this chapter we have described a multidimensional mapping scheme for XML and relational database systems. In addition we have analyzed four different multidimensional indexing methods for our mapping (the UB-Tree and three variants of a compound B-Tree) and compared them to the well-known edge mapping. Our measurements showed that path

Index	Time
UB-Tree	0.692s
DidSurr	0.053s
SurrValDid	36.197s
DidSurr_Val	0.045s
Edge_Compound	0.069s
Edge_Secondary	0.063s

Table 4.10: Time for reconstruction of a single document

compression using MHC allows to store and index paths in XML documents efficiently in RDBMS. MHC preserves the order of the original documents and gives a compact and prefix-free representation of paths in XML documents. Two optimizations (leaving out root nodes and only storing complete paths to leaves) show that the technique can be further adapted to application scenarios. The three-dimensional model even allows simultaneous navigation over several documents due to the separate dimension for document ids.

The advantages of the proposed model are supported by the presented benchmarks. In the overall running times of the queries we were faster in every case than edge mapping and had an up to seven times smaller database size. Additionally we have described the orthogonal nature of typical XML queries in this chapter which have impacts on the used indexes.

Overall our results can be viewed from different perspectives. Due to the special orthogonal requirements for selection and projection, a combination of the two compound B-Trees is the most promising one with respect to elapsed query time. The use of two indexes obviously requires more maintenance work for insertion as a tuple is effectivly inserted two times into the indexes. Another drawback is the consumption of additional storage for the second index (the use of additional indexes renders one variant of the edge mapping unusable). A very promising result is the use of a secondary index in our mapping to avoid the limitations of one index. This combines the advantages of two indexes while reducing space requirements.

The effects of different restrictions on the data are also worth noting in this discussion as the selectivity of value and structure attributes strongly influence query running time. Restrictions on value attributes are usually common for data-centric documents, especially when the documents come from relational sources (as relational data on its own does not contain any structure). In contrast to this, queries on document-centric documents often carry restrictions on structure. When paths are not fully specified then the results of Chapter 3 about navigation in XML documents have to be considered.

These considerations have to be especially taken care of when chosing indexes in database schemas for storing XML.

The results of this chapter have been in published at BTW 2003 conference [BRB03].

Chapter 5

XML in Digital Library Systems

5.1 Goal of Digital Library Systems

The first ideas for a collection of all human knowledge which should be available everywhere date back to the 1940s when Vannevar Bush [Bus45] came up with a system design based on microfilm. Bush's idea was heavily influenced by the available technology in the 1940s when microfilm reduced the space requirements for printed material severely. Microfilm would have allowed to store 10000 pages of the "Encyclopedia Britannica" on about the size of one sheet of conventional A4 paper. Bush called the machine *Memex* and even proposed a method to connect information of interest. This method is nowadays known as creating links between nodes in a hypertext system. The perspective of several thousands of these collections with links on the desk of every scientist would have changed scientific work significantly. The system though was never developed and remained an idealistic approach.

The original idea was highly supported again in the 90s of the 20th century with the sharply increasing use of the Internet and its overlay network, the World Wide Web. This led researchers to the idea of digital libraries which should originally enable people to access all knowledge they desire by purely digital means and that knowledge becomes available to everyone anywhere.

The euphoria in the field in combination with the rapid spread of Internet technology led to a very fast development and installation of many different systems. While the first digital library systems were usually targeted at local use, later systems emerged into large scale applications with a huge amount of available material [ACM]. This happened especially when publishers started to move existing material into the systems so that most of it became available in digital form for the first time.

With this development digital library systems came one step closer to the original idea. There are many technical challenges which are still unsolved arising from the design of different user interfaces, different presentation formats and different functionality.

5.2 Application Fields of XML in Digital Libraries

The original design of XML as a successor of HTML already shows that XML is very strongly driven by web based technology. As web technology is essential nowadays in digital library systems, XML also became a very important topic in this field. Due to its very flexible nature XML is adopted in different areas of digital library systems. In the following we give a short overview on three different application areas of XML in digital library systems, namely as an exchange format of data, as a presentation format and as a flexible, reliable and long term archival format.

5.2.1 XML as Exchange Format

The application of XML as an exchange format is exactly the application that XML was originally designed for. Data exchange is fundamental in interoperability, which became a very important topic in research on digital libraries. Most applications for interoperability suffer from the lack of a general exchange format. This motivated the development of many different data formats in different projects for interoperability and also motivated the development of mediators which handle the data exchange with different systems. Besides the high effort in the design and implementation of such mediator systems there are also fundamental obstacles on the way to a general format which are caused by the expressivness of the source and target data format of the mediators. It is possible to transfer a simple format into a more complex format while it is not generally possible to transfer a complex format into a format which only allows simple constructs.

Apparently these problems occur because there is no generally standardized format available which offers the desired flexibility. This is a starting point for XML which was standardized from the beginning and also had the advantage that the work of the W3C is in general seen as very valuable. The wide spread use of XML was also supported by the availability of often freely developed tools which can be easily integrated into systems.

Examples were XML is used as an exchange format in digital library systems are numberless. Mostly XML found an application in the exchange of metadata and content data. A very prominent example in this context is the OpenArchives Initiative ([LdS01, OAI] and Section 7.6) which defines an XML based format for the exchange of metadata (and also a protocol for harvesting the metadata which is not a topic here). The proposed metadata format is based on Dublin Core which became a de-facto standard for cross-discipline metadata.

The Resource Description Framework (RDF) is another application of XML as an exchange format [RDF] with a tight link to digital libraries. RDF provides a lightweight ontology system to support the exchange of knowledge on the Web and is an integral part of the sematic web activity. In the beginning RDF was purely targeted towards metadata of documents on the World Wide Web but later emerged into a framework which enables the description of resources which can be identified on the web (but not necessarily retrieved). This can range from information about specifications and prices in a webshop (e.g. for selling digital material) to the preferences of users when retrieving information. This makes RDF very valuable for digital libraries.

5.2.2 XML as Presentation Format

The name Extensible Markup Language (XML) already explains the design of XML as a language which allows the description of other markup languages. The most widely known markup language for presentation purposes is apparently HTML. HTML was originally designed for simple presentation purposes only and uses predefined markup up. Markup languages formulated in XML allow a strict separation of content and presentation and allow user-defined document types. This ability though is only important for data which is presented to humans. Purely machine readable data which is never viewed by humans is considered to be an exchange format and is discussed in Section 5.2.1.

The above definition allows a strong separation between the exchange format and the presentation format. The separation is usually also visible in implementations of systems. External access to local data (data exchange) is equivalent to allowing access below the presentation layer.

In contrast to HTML the actual presentation and rendering of XML documents is not done directly by browsers (as they are available for HTML). This is due to the fact that the semantic of XML tags is not fixed as with HTML. When XML should be rendered/presented the semantics of XML tags for presentation is provided by Cascading Style Sheets (CSS) or with the more powerful Extensible Stylesheet Language for Transformations (XSLT). XSLT even allows the transformation into other XML documents, HTML documents and even PDF.

5.2.3 XML as Archival Format

The problem of archiving digital material became especially important when digital library systems grew and offered more documents. The archiving problem is basically twofold. It includes a storage problem which means that the data should be physically readable over an arbitrary amount of time and the actual archiving problem which focuses on the data format of the archived digital material. The storage problem is a technical problem and can be solved by storing the data redundantly, using long-lasting media and copying data to modern media. The storage problem is not further discussed here as it is not an application field of XML in digital library systems.

As mentioned already, the archiving problem is not a technical problem but deals with the used data format for storage. For archiving, the syntax of data is essential on the one hand, but on the other hand the syntax of the data is useless without semantics. In XML the syntax is given by two aspects. The first one is that XML documents are supposed to be *wellformed*. The second aspect are *valid* XML documents which means that the document is checked against a DTD or XML Schema. This not only ensures syntax but also semantics to a certain extent (e.g. datatypes in XSchema). It is important to note that the interpretation of data is completely excluded from these mechanisms. Some of the interpretation can be implicitly embedded into the tag names (e.g. <`author`>, <`title`>, etc.) which even makes the tagnames human readable. This often was an argument for XML but adds significant overhead to the XML document. Already adding linefeeds to documents on each line (lenght of a line between 10 and 80 characters) causes an overhead between approx. 1% and 10% and detailed tagnames add even more overhead. Shortening the tagnames is often used as an argument against the overhead, but this procedure looses all (human) readability and makes the format as hard to understand as every other binary format. The situation can become even worse when the DTD is formulated without any restrictions on the usage of tags. In a careless design with Kleene stars in all <`!ELEMENT`> expressions for all tags almost all advantages of syntax validation are lost as all tags can occur everywhere. A clear and stable design of an XML DTD or Schema is as important as clear and stable design in software engineering.

When using XML as an format for archiving, XSLT (Extensible Stylesheet Language for Transformation) is a very valuable tool. It enables the generation of other formats (e.g. HTML or PDF) from the original in XML.

To be able to archive documents in XML it is necessary to prepare documents in XML. Tools for widely used programs like LaTeX and Microsoft Word are available and offered on a commercial basis.

There are various projects which use XML as an archival format especially in digital libraries. Mostly these projects are initiated by libraries who want to archive theses and disserations. Two of these projects are the DiVA Project from the University of Uppsala and the DissOnline project at Die Deutsche Bibliothek (German National Library). The common approach in these projects is that XML is not meant to replace the original documents but to act as an additional format for safety reasons which is still available when the originally created formats become impossible to use. This fact is strongly supported by the openly documented features of XML formats.

Chapter 6

An Introduction to the OMNIS/2 System

6.1 Current State and Demands of Digital Library Systems

As mentioned in Section 5.1 many different digital libary systems were developed. Current digital library systems support various ways of retrieving documents, ranging from mere catalog retrieval (OPAC systems) and full text search, to content based search in media libraries like video libraries or music libraries. The following features, however, are available only in parts or are missing at all in current digital library systems:

- The integration of full text retrieval library systems and *multi*media database systems, which may be distributed or remote, into a progressive, interactive, multimedia digital library system, which offers the option of transparently including other systems (with often large collections of data) and even introducing cross references among them.
- There is a need for a generic management of metadata, not only for documents which are stored in the multimedia digital library system, but also for any document of additionally connected digital library systems. This is necessary in order to make use of modern filtering and retrieval techniques and also for the transparent linking and processing of documents beyond the boundaries of a single digital library system.
- Automatically generated links between documents inside of a digital library and among documents of different digital library systems. Every retrieved article from a digital library system contains further bibliographic references which are mostly stored in the same digital library system. To retrieve these references the user usually has to initiate another search query, possibly with further restrictions, until the correct result is returned. An automatic linking between documents is not supported today.

- In addition to automatically generated links users want to create and follow personal relevant links to other documents in the same digital library system or to other systems. This feature is often asked for by research groups who work with specialized library systems like VD17 (a digital library of all German printings of the 17th century, a former project) [DHW97].

- Current library retrieval systems lack the possibilty of adding personal annotations to documents, as this would require write permission for all users of the digital library system. Annotations are helpful, though, to explore content for oneself or for a certain user group or to simply discuss a topic. Discussions require annotations on annotations. If various lengths and various types of media annotations are possible, a user friendly authoring tool and the integration of a multimedia database system for storage purposes are required. Annotations are supported in some newly developed media libraries and personal libraries, but not in catalog and retrieval systems.

- Almost all current digital library systems miss a personalization scheme which every user can feed with personal interests. The personalization can, on the one hand, be used with an existing pull-technology for an additional semantic filtering of information or, on the other hand, as a push-technology to inform a user about new relevant incoming titles. The personalization feature should be adaptive to recognize shifts in the users' interests, but it should also be a corporate tool to create recommendations.

In the following we will further discuss the above mentioned items and how they can be implemented in a modern system. We will leave out two demands though. The fundamentals of the personalization scheme are described in other publications [SK00]. Automatic linking has been a topic of the CiteSeer Project [Cit] and working solutions have been developed there.

6.2 Architecture and Design Overview of OMNIS/2

6.2.1 Design Principles

Our idea is originating from the fact that we can not change or do not want to change the existing digital libraries as they are either well established systems or the effort to change them would be tremendous. We can consider the existing systems however as large containers of documents with a powerful query language. Therefore our extensions act as a metasystem for these digital library systems, enhancing them by new functionalities. Even if our metasystem will be integrated into one of the digital library systems in the future, this would be just an additional layer on top of the original system. Thus, in an abstract view it would still be the same architecture. Our metasystem is able to search in one or various digital libraries and to automatically link their documents, to annotate them, to extend them with multimedia components and to personalize them. The original documents themselves remain in the original database systems and are never modified not even for added anchors. Documents are represented in an own database of the system (referred to as meta database) simply by their logical addresses and meta data. The linking, including the anchor positions, is stored in the metasystem exclusively and is included dynamically into the retrieved documents at run-time. In the same way documents can be annotated with user-related, group-related, or general annotations. These annotations can be arbitrary long texts, multimedia annotations, or even annotations on annotations. In addition, we have used a concept for personalization (GRAS algorithm [SK00], not further described in this work), which supports a better filtering and a personal ranking of the results of the underlying digital library systems. The GRAS algorithm unlike other personalization schemes uses Gaussian curves to describe user and object profiles, which results in a better overall quality of retrieval. The personalization feature can also be used as a messaging system about new relevant documents in the library. Both push and pull strategies are possible. The annotations and the data necessary for the personalization feature (i.e. user and object profiles to all documents) are also stored in the meta database. Often users like to create, to store, and to retrieve additional personal documents together with the original documents. Since these personal documents may consist of texts, tables, graphics, audio, or video the metasystem itself is required to offer all features of an interactive, multimedia database system including user workspaces, full text indexes, etc. For uniformity and consistency reasons the local documents are handled like documents in external digital libraries except for the write permission. If they are annotated or enriched with links, this is again stored outside the document in the meta database. This concept can be seen as an extension of the Amsterdam Hypermedia Model [HBvR94], which by itself is an extension of the Dexter Reference Model [HS94], clearly separating the document layer (within-component layer) from the linkage layer (storage layer). OMNIS/2 supports a more powerful link concept than the currently existing one in the WWW: n:m links, bidirectional links and newly developed temporal links and overlapping links are supported, while the consistency of links is still ensured.

Summarizing, the metasystem can, on the one hand, be seen as an integrated, interactive digital library system with full text, multimedia and hypermedia documents, and on the other hand, as a meta system for existing digital library systems, extending these by hypermedia elements including user workspaces, automatic and personal links, annotations, personal multimedia documents, a transparent access to further library systems, an integrated management of meta data, and a unified interactive user interface.

6.2.2 Integration of External Systems

In general, one can distinguish two possibilities of integrating an existing digital library system into OMNIS/2: A tight integration and an integration with the help of XML. The tight integration utilizes an internal interface to the relevant systems. Then the systems are directly integrated into OMNIS/2 by using these interfaces. Such an approach has not been further elaborated for OMNIS/2 and remains a possible approach for future systems. If systems use the XML integration OMNIS/2 manages the shared DTD for XML and additionally uses the tagged information for its management of meta data.

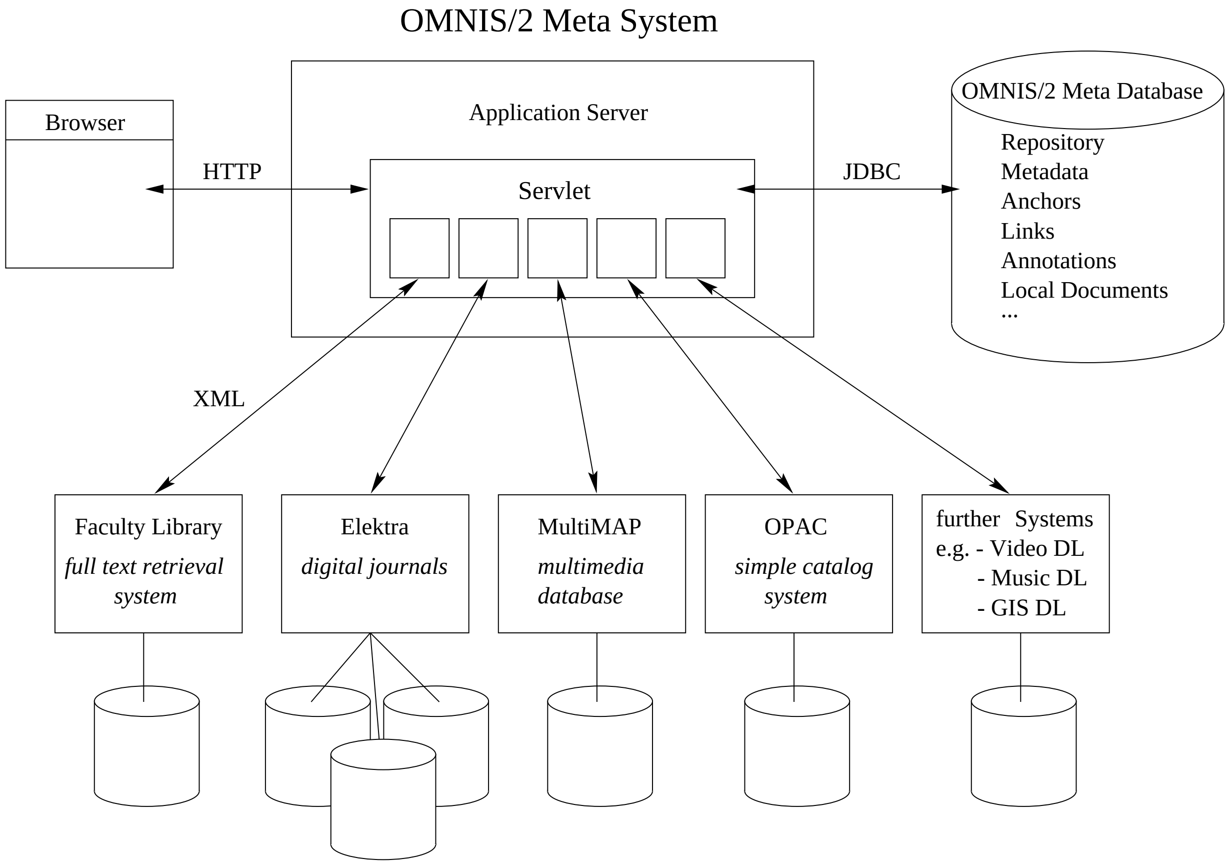


Figure 6.1: Architecture of OMNIS/2

6.2.3 Use of XML in OMNIS/2

XML is used in OMNIS/2 in two aspects. It is used as an exchange format and as a presentation format. The exchange format is already mentioned above for the coupling of external systems. For this, the DTD of the external XML format is used by OMNIS/2. For processing all extenal XML formats are transformed into an internal XML format (which is described later). The internal format is then enriched with links and additional information before it is presented to the browsers of the users. In this last step XML is used as a presentation format. As of this writing there are no XML-only browsers so it was decided to transform the internal XML format to HTML so that common browsers can display the documents. As users can upload their own documents it is of course possible to store documents in XML in the system. The documents though already have to be in the internal XML format. An exception are documents from the OpenArchives initiative which are treated as an external format (Section 7.6).

6.3 Implementation

An important aspect of OMNIS/2 is that neither the integrated digital library systems nor the stored documents need to be modified. It is not even required to update the original documents in order to mark the links, respectively the anchors or the starting points of annotations. This is possible with the help of a three layer architecture similar to the Amsterdam Hypermedia Reference Model [HBvR94]. The integrated digital library systems correspond to the within-component-layer, which holds the documents (Fig. 6.1 shows the overall architecture). The composition, linking and annotation is performed in the meta layer, which consists of an application server and a database connection to the meta database. This layer represents the storage-layer of the Dexter and the Amsterdam Reference Model. The third layer is the presentation layer running in the user's web browser. This is the conceptional view on the architecture, which is shown in Fig. 6.1 vertically.

The integrated systems shown in Fig. 6.1 are only examples. In a more technical view on the architecture, which is shown in Fig. 6.1 horizontally in the upper part of the figure, the so called "three-tier architecture" becomes visible which is the foundation of todays implementations of the web access to the database. In the implementation we have chosen a server-side access to the meta database by using Java servlets and JDBC to lower the load on the clients's side (to support thin-clients). The servlet extends the webserver to a complete application server which holds the core of the metasystem. The communication of the browser and the webserver is performed via the standard HTTP protocol or by exchanging java classes, if applets are necessary for the presentation. User input is forwarded by the browser to the servlet and is processed there. All servlets are connected to the meta database with JDBC.

If a query is initiated for one of the coupled digital libraries then the query has to be transformed by the servlet into the URL-format of the relevant library system and sent to it. If a link is followed into a document of one of the coupled databases a corresponding query has to be generated by the servlet with the help of the information in the meta database. In the case of the XML coupling, the metasystem parses the retrieved XML documents with an XML parser and enriches them with the additional information from the meta database (like link anchors, annotations, etc.). To do this, a second query to the meta database is initiated. This query is always necessary. The resulting, enhanced document is currently transformed into HTML containing Java applets or plugin applications and is sent to the WWW browser for presentation.

In this context the architecture of the meta database plays a key role. It has to store metadata, link anchors, annotations of various media types, and local documents which can also consist of different media types. The amount of stored data is expected to grow rapidly as anchor and link information has to be stored for every single object that was edited by users. Additionally, the personalization feature requires the storage of object profiles for all documents in the meta database. Except for the anchor values the data is strongly structured. It is important to note though that there is also semi-structured metadata from the underlying digital libraries if these systems use metadata in XML. The aspects which have to be taken care of in the meta database for semistructured data include storage, retrieval, internal representation, fragmentation, and indexing. As the metasystem is based on several underlying systems and also maintains its own meta database inter-system consistency of the stored metadata and the underlying digital libraries is a very important aspect.

An overview of the architecture of OMNIS/2 and its implementation has been published at ECDL 2000 conference [SB00].

6.4 Related Work

6.4.1 ComMentor

One of the first projects to offer shared annotations for web pages was the ComMentor project from Stanford University [RMW95]. Although ComMentor was developed very early after the World Wide Web became popular it already incorporates most (design) features that were later picked up by other systems. While the general description of ComMentor discusses adding annotations to arbitrary documents, the implementation proposed by ComMentor is focussed primarily on the WWW. There is a separation between the content providers which offer documents and third-party providers which offer annotations. Documents and annotations reside on different servers which separates the annotations from documents and defines the annotations as server side annotations. The annotations are merged with the original documents by the browser whenever the documents are retrieved from the content providers. Annotations can be either transferred with the help of a special MIME content type or embedded into HTML by using special META tags which are an extension to HTML. Annotations in ComMentor are administered in sets with an access control. This allows e.g. private sets of annotations, group sets, and public sets.

The prototype implementation of ComMentor was directly integrated into several tools. The functionality of the annotations server was implemented in an NCSA http server. The merging functionality of documents and annotations was directly integrated into the NCSA Mosaic webbrowser. The webbrowser was also extended with special GUI features to allow the display of annotations as inline text, pop-up windows, etc. and with components to allow adding annotation to web documents.

The publications about ComMentor even contain a description of robust locations for annotations. The approach is to store redundant information for locations (e.g. tags and content strings) which we will further discuss in Chapter 8.

In an overall summary the implementaiton of ComMentor is quite proprietary but it is important to note that during the project phase of ComMentor there were no widely available standards or tools for development of software on the WWW.

6.4.2 OSF/GrAnT

The OSF Research Institute developed a system for a browser independent solution for annotations and other meta-information on the World Wide Web[SMB96]. The OSF approach uses a s.c. *GrAnt* (Group Annotation Transducer) to enrich existing web pages with additional information. A GrAnT is a special form of an *application-specific stream transducer* which was presented by some of the authors in another publication[BMMM95]. Nowadays such approaches are known as proxy servers or special forms of application servers. The advantage of an implementation of the system as a proxy server is that users can choose from a variety of servers where documents to be annotated reside on. Otherwise users would be tied to one server which provides the annotation facility. In addition no special browsers are required to work with the system as all annotation functionality is implemented in the proxy server. The proxy server also allows to store the annotations on a special annotation server. Initally the annotation server of the OSF/GrAnT system did not provide any database functionality but considerations about this can be found in the outlook of the work.

The proposed system especially concentrates on work groups who share knowledge through the system by annotating documents. This is also reflected in the rights management of the system which is built on public groups which can be joined by any user or private groups which can only be joined by administrative validation. Each annotation on the annotation server is an element of an adminitratively defined annotation set. Each annotation set is labeled with a topic which allows a set specific filtering. Each group is permitted access to one or more annotation sets. In the original proposal annotations in OSF/GrAnT can only consist of text annotations.

Parts of the implementation of the system are based on code from the Commentor system (see Section 6.4.1) and turn the modified Modiac browser with embedded annotation support into a proxy based system.

6.4.3 Project Clio - Grinnel College

At Grinnel College, Iowa a system was deleoped in 1999 to offer students the possibility to work with web pages as they are used to with printed material [LMR99]. The idea behind the project is to improve teaching for students so that they can become active learners although the World Wide Web originally only allows passive interaction. Due to this background a major part of the project is based on surveys for end users. There are intensive discussions on the usability of the user interface but also some information on the concepts used in the system. The concepts propose the annotation of local and remote pages as well as annotations at author-defined and automatically generated positions on a page. The authors also propose a rights-management system for annotations which provides multiple protection levels (readability, writeability). In the project description it is mentioned that the system also supports annotations on changing pages. It is proposed to use heuristics based on approximate text matching to determine the annotation points in changed documents, but there is no detailed technical description of the methods available.

The main implementation of the systems is designed as as a CGI script in a webserver and uses server-side annotations which can reside on different annotation servers. From the available literature it is not clear how the exact communication is performed. Displaying the annotations on the clients of the users requires a JavaScript enabled browser.

6.4.4 Annotea

Annotea [KK01] is a Web-based shared annotation system developed by members of the W3C. The project offers a solution based on re-using as much existing W3C technology as possible. Annotea is based on RDF, XPointer, XLink, HTTP, and XHTML, which allows a non-proprietory design and architecture. Annotations in Annotea are modeled as metadata and viewed as statements of an author about a webpage. With this approach Annotea fits very well with the Semantic Web initiative, which is also based on the above mentioned technology. As the annotations are based on RDF, the properties of annotations must be described with an RDF schema. The annotations themselves are stored in generic RDF databases which allows reuse of the annotations for other uses. HTTP servers are the frontends for the RDF databases and all communication with the servers is performed by using HTTP. Annotea distinguishes between local and remote annotations which means that annotations can either be stored locally on the client side or remotely on servers in RDF databases. This separation goes hand in hand with a rights management which considers local annotations to be private and remote annotations to be public. Further access restrictions can be imposed on the annotations by using common access control of HTTP servers as they are frontends to the RDF databases. The merging of documents and annotations is performed on the client side with the help of a special implementation of a browser and does not use a proxy-based approach as other systems do. Annotea uses a special enhanced version of the W3C reference browser Amaya as a prototype implementation. In contrast to other approaches Annotea divides annotation into three components: the body of the annotation, which contains the text or graphical information of the annotation, the link to the annotated document with a location within the document and metadata information about the author of the annotation, etc. The position of the annotation is described by using XPointer. In cases where the underlying document changes and the position in the document is no longer valid a method is proposed which embedes text information into the XPointer statement so that text matching can be performed to find the location again but no further technical details are given. The features of Annotea are also integrated into the open source internet application suite *Mozilla* [Moz] within the project *Annozilla* [Ann].

6.4.5 Hyperwave/Hyper-G

The Hyperwave system was formerly known as Hyper-G [Mau96]. The roots of Hyperwave date back even before the World Wide Web when Gopher was widely used as an information system on the Internet. Hyperwave is designed to be an information server for knowledge and document management (for the Internet but even more for intranets). This means that all documents are not simply copied onto the system but they are stored in the server with special server tools thus providing automatic fulltext indexing of documents, creation of metadata records, a version management for the documents in the server and a rights management for users of the server. A notable feature in Hyperwave is the strict separation of links from documents. Links in Hyperwave are bidirectional which results in links being removed when documents to which they are pointing to are removed. This even works across several distributed Hyperwave servers and ensures referential integrity as all documents are moved with special server tools. In its early versions Hyperwave provided special authoring tools and also special viewer tools which users had to use to benefit from the features of Hyperwave. In these early versions Hyperwave supported links in file formats like PostScript and MPEG which did not support links. Viewing these enriched documents required special tools to combine the linking and the file format. The support of these tools (e.g. Harmony and Amadeus) was stopped when web browsers became widely popular and parts of operating systems. In current versions some authoring capabilities are only available as extensions for the Windows platform.

The Hyperwave Server in its original design (beginning of the 1990s) already contains features which the World Wide Web does not support and probably will never support due to its heterogenious nature. It is important to note though that all these features are not based on open standards and are only valid for documents which are stored on the Hyperwave server.

6.4.6 Wikipedia

Wikipedia [Wik] is a project which started in January, 2001. Is is an effort to build a large encyclopedia based on s.c. WikiWikis. A WikiWiki is a collection of interlinked web pages, any of which can be visited and edited by anyone at any time. A wiki page, which is a single document, is written in a simple markup language (even simpler than HTML) using a web browser. The editing is offered by a web based frontend of a wiki engine. For the presentation on the Web the wiki pages are transformed to HTML by the wiki engine. There is a large number of different wiki engines available ranging from small applications to complex content management systems. The used markup language though differs from wiki engine to wiki engine and could also be XML (although it is not used in Wikipedia at the moment).

The idea of Wikipedia is heavily based on the philosophy of the free software movement. People collaboratively write and revise articles and publish them. While vandalism of the pages sometimes occurs due to the open access and the lack of peer review it is not a significant problem as it is immediately recognized and corrected by the large group of participants. Over the time several different language version of the Wikipedia emerged ranging from the English original to German, Estonian, Hindi and several other languages. In October 2003 the English version of Wikipedia offered more than 166000 documents and is still growing rapidely. Due to the success of Wikipedia several other projets were founded like Wikitravel (a free collection of travel guides), Wiktionary (a free dictionary project), Wikibooks (a free textbook project), and Wikiquote (a free encyclopedia of quotes). Other efforts to build peer reviewed encyclopedias based on wikis were not successfull and were mostly abandonded.

Chapter 7

Components of the OMNIS/2 System

7.1 Integration of External Systems in Detail

In the introduction on OMNIS/2 the integration of external systems has been addressed in a short overview. As the integration is fundamental in OMNIS/2 it is important to discuss it in more detail. External systems are integrated by using XML. XML is hereby used as an exchange format between the external systems and OMNIS/2. The transfer of data is done in two steps. At first the external systems are queried by sending query strings. This is usually done via query facilities which the external systems offer to the outside world. It is also possible to send the query in XML to the external system if this is supported. Systems then return the answer to the query in XML. At this point it is important to distinguish two different possible answers. A query can either return a result list or it can return a single stand-alone document which corresponds to the query. Result lists are usually only compact lists of metadata and it can not be guaranteed that the order of the result list remains unchanged over separate queries. Problems especially arise if the lists do not contain any identifiers which allow adressing of different items of a search list. It is then not possible for OMNIS/2 to add additional data to the result list. The second option is the return of a single document to a query. Analog to above, identifiers have to be returned to be able to add annotations and links. For some systems there is no difference between the format of a result list and a stand-alone document. This is especially common for systems which only store metadata and no content documents. A system which conforms to this is the digital library system of the faculty of computer science at TU München, which has been integrated into OMNIS/2.

Result lists and single documents are tightly linked to navigational and associative browsing and further discussed in Chapter 8.

Every integrated system usually returns its own XML format which has to correspond to a DTD. For every integrated system OMNIS/2 requires knowledge about the XML format and its DTD. Knowledge about the DTD already allows validation of the format at a very early stage in processing the output of external systems. The DTD is usually declared at the beginning of the exchanged XML files and does not need to be stored in

OMNIS/2.

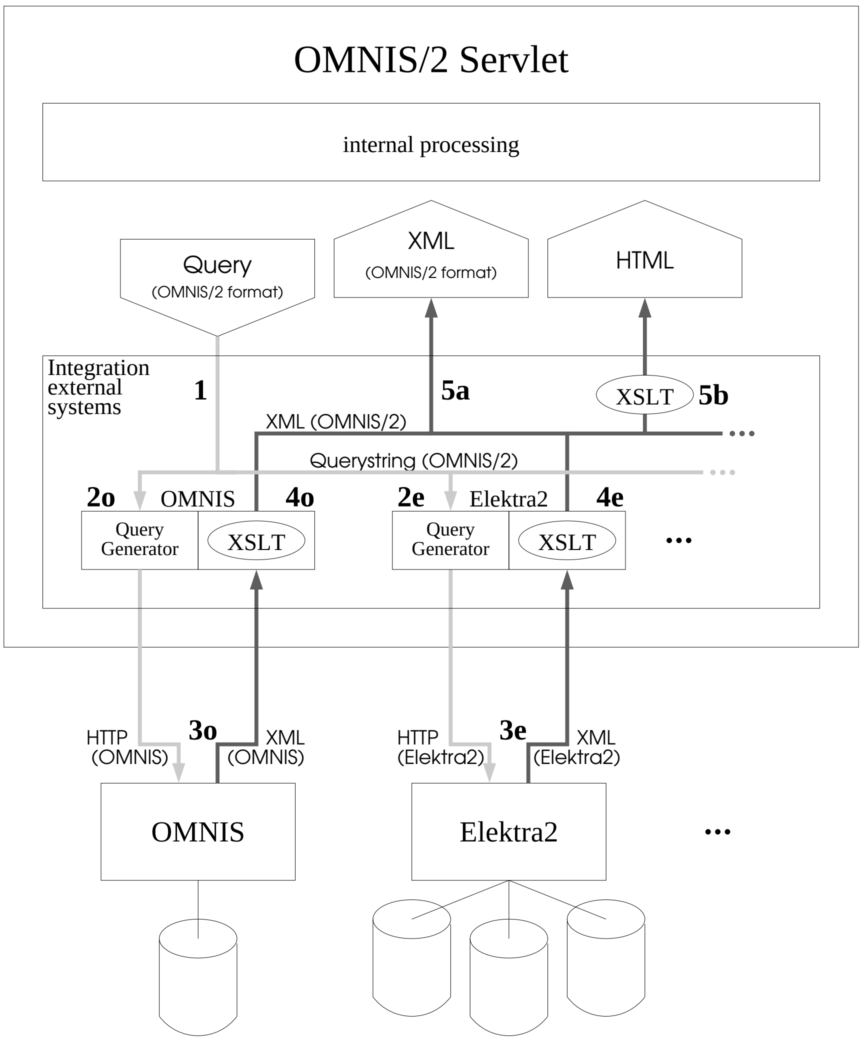


Figure 7.1: Querying external systems

To simplify later processing the integration of every external system is encapsulated into separate modules. These modules act as mediators and transform the external format into an internal OMNIS/2 XML format. This is done by XSLT stylesheets which have to be developed separately (one for every integrated system). The internal format was designed to be very flexible. The format can handle both, XML metadata and XML content data. For metadata it is based on the Dublin Core metadata set. While this set is very simple it is yet powerful enough to deal with all systems that were integrated into OMNIS/2. As many systems not only offer metadata but also content data the internal format was extended by certain tags which mark content. The content data can either contain a pointer to the content or the content explictly (this of course in a format which can be transported by XML).

The internal format apparently has to preserve the identifiers of the external systems. The structure of the identifiers is based on a similar concept as used in the OpenArchives Initiative (Section 7.6). The identifier of a system is preceeded by a prefix naming the source system (e.g. *elektra:43567* would reference the document *43567* from a system named *elektra*). This scheme is used in OMNIS/2 for all identifiers which refer to documents in external systems. When documents are to be retrieved with such an identifier, the identifier is transformed into the original format.

The internal OMNIS/2 format is the basis for all further processing steps within the system. Especially adding links to the documents on the fly is performed on this format.

Figure 7.1 shows the querying of an external system in detail. Querying and processing of the result is performed in several steps. The single steps correspond to the numbers in Figure 7.1.

1. The query is delivered to the external systems. The query can either be targeted to one system or several systems can be queried in parallel.
2. In this step the query string is processed by query generators and transformed into the special query format of each system. For both of the shown systems (OMNIS and Elektra2) the qery is transformed into HTTP requests.
3. The query is processed by the external system(s). For both systems the results are XML documents which are returned to the OMNIS/2 system.
4. The result of the query is transformed from the format of the external system into internal OMNIS/2 XML format. This is performed by an XSLT processor which uses an XSLT stylesheet for this purpose.
5. In a final step the results from the different systems are collected (if several systems were queried) and further processed in OMNIS/2 (added links, annotations, etc.).

7.2 The Object Oriented Document Model

7.2.1 Theoretical Preparations

As OMNIS/2 sits on top of established digital library systems it has to handle several different types of documents which emerge from the underlying digital library systems.

Following this we have at first examined different ways of how to categorize the documents. The most simple one is to categorize the documents according to their source. We call the documents from the underlying digital library systems external documents and refer to the documents uploaded by the users, i.e. userdefined documents and annotations (texts, graphics, etc.) as internal documents. Internal documents are stored in an own relational database system (the so called meta database, see Section 6.2 for details).

The content of internal and external documents strongly differs. Since OMNIS/2 does not use any harvesting the content of external documents is not explicitly stored by OMNIS/2 but remains in the underlying digital library systems and is only retrieved upon request (an exception are documents from the OpenArchives Initiative which will be discussed later). In our system we only hold a persistent unique identifier for every external document together with some metadata and linking information. For local documents (where we store the content locally of course) we can easily assure this, for external documents it is an requirment for the underlying digital library systems so that they can be utilized for OMNIS/2. This fact is especially important in the context of a posteriori cross linking.

We can also categorize our documents according to structure. Our design of the system as both, a metasystem and a stand-alone multimedia database system, requires that there are not only simple stand-alone documents from various sources. Users must have the ability to compose their own documents from various existing (i.e. external) and user-defined (i.e. internal) documents. The document model of OMNIS/2 therefore distinguishes two types of documents, composites and atoms, thereby following the Dexter Hypertext Model [HS94]. Composites either consist of one or more atoms, one or more composites or both. A composite can not exist for itself, but always has to contain at least one atom. Consequently composites are internal documents and atoms are either local or external. This results in a hierarchical document structure (see Fig. 7.2 for the BNF-notion of the document structure and Fig. 7.3 for an example). This hierarchy is actually a DAG (directed acyclic graph), thus atoms and composites can be shared within the same hierarchy and can occur in several different levels.

```
<Document>  ::=  <Composite> | <Atom>
<Composite> ::= {<Composite> | <Atom>}+
<Atom>      ::= internal Atom |
                external Atom
```

Figure 7.2: Document hierarchy in BNF-notion

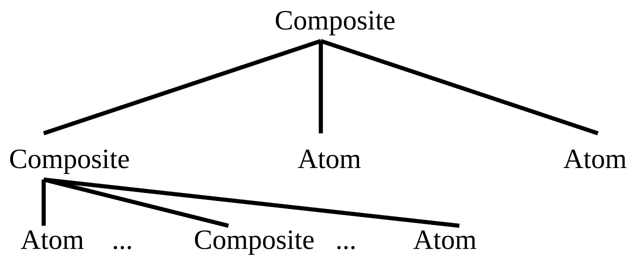


Figure 7.3: Example of a hierarchical document structure

Atoms are of a single type or may even be complex if they are accessible as a whole in an external system. In the current system we consider text, images, audio, video and external (i.e. the external documents) as document types. The external documents are further divided into the various document types originating from the integrated external digital library systems. This separation implies a strong encapsulation of document specific features in the document types and motivates the implementation of this document model in an object oriented way.

Another consideration comes from the fact that due to the heterogenity of the underlying systems OMNIS/2 has to find a way to access the different documents in a uniform way.

Taking all these considerations into account we decided that an object oriented design would follow exactly our needs. In our model every document is seen as an object and can exist for its own at runtime. Every object is identified by a persistant unique identifier (UID). The document objects have the ability to create themselves from the database on request. In addition they carry all information about themselves (i.e. anchors, anntotations) and can display themselves for the presentation to the user. A second display method enables a different presentation e.g. for an authoring tool. In the case of composite documents the display method simply calls the display methods of its child documents.

This design enables us to use object-oriented programming techniques although the underlying systems do not offer object oriented features.

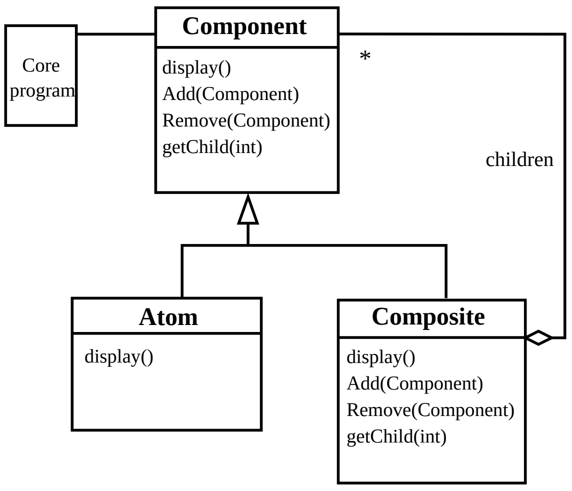


Figure 7.4: Composite design pattern (UML)

7.2.2 Implementation

A very suitable method to implement our document model uses the composite design pattern (a structural design pattern) [GHJV94]. This design pattern enables us to represent part-whole hierarchies using objects and also provides a uniform interface to both composites and atoms (see Fig. 7.4 for the design pattern in UML). It gains its flexibility through the component class which acts as a container. This is especially important for the presentation of the documents with the display methods as it is easy for composite documents to simply call the display methods of their child documents (regardlessly whether they are atoms or composites) which form the document hierarchy. For OMNIS/2 we decided that the documents display themselves in XML, which enables us to use modern techniques like XSLT to process (annotate, link) the documents. The hierarchical structure of XML in addition assists us to preserve the hierarchical structure of our document model. The solution overall combines several important aspects. It is slim but nevertheless very powerful and reduces implementation time drastically (this is also supported by our decision of using Java for the implementation). It hides heterogenity very well in our case by using encapsulation and it is still easily extensible.

The results of this section have been in published at the DLib Workshop 2001[BS01c].

7.2.3 The Anchor Model

In our context anchors follow the definition of the Dexter Hypertext Model [HS94] and in general are an abstraction of link sources and link destinations, while links connect source anchors and destination anchors. The anchoring mechanism is required to address and to refer to locations within the content of an individual document. The anchor and linking concept of OMNIS/2 is quite powerful. It supports 1:n links, bidirectional links and overlapping links. This link concept is apparently more powerful than the one currently used in the WorldWideWeb and requires appropriate management of anchors of course. As explained above already it is a very strong additional requirement that the anchors are separated from the actual documents (which makes the OMNIS/2 system special), as there is usually no access to the underlying systems.

We decided to use a similar approach to anchors as we used for the document model. The major difference though is that we store all anchors in our relational meta database, so there are no *external* anchors. For each type of document the anchors have to have the matching characteristics (i.e. for text documents there are text anchors). This encapsulation is necessary as for every document type the position description of the anchor can be completly different. Text anchors can be easily identified by simply using position and length attributes. For anchors in pictures or graphics we decided to use simple geometric descriptions (e.g. polygones, circles with center and radius, etc.). As we will see in Chapter 8 this can be further extended to handle changing and moving documents. In XML documents the anchors describe a position within an XML document. Using XPATH-style syntax to address parts of the document is very flexible and very suitable for our requirements.

Besides their exact position the anchors also need unique identifiers (AIDs) so that they can be associated with a document.

Strictly following the ideas of our object oriented document model it would have been possible to make the anchors part of the object hierarchy and to treat them as special occurences of documents. This would have enabled us to create structures which contain links on links (next to links on documents). We decided against this possibility although it would have been possible following the Amsterdam Hypermedia Model. We consider this feature as not important for a real world application and will not implement it. The anchors form their own object hierarchy which can be seen in Figure 7.5. As for documents we again put great effort into creating stand-alone objects. This results in anchors that create themselves from the database and have the ability to store themselves independently from the documents.

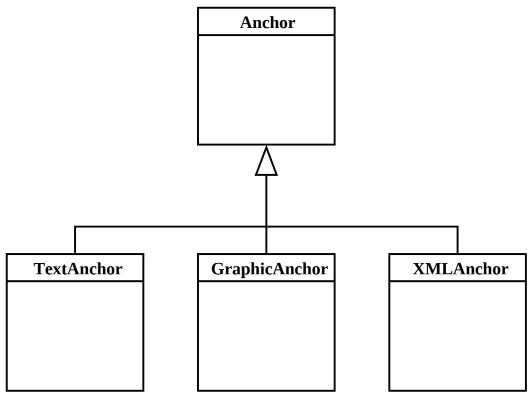


Figure 7.5: Class Diagram for anchors (UML)

Combining these different features we get a flexible concept which allows us to enrich documents by using 1:n links, bidirectional links and overlapping links.

The anchor model of OMNIS/2 has been published in [BS01b].

7.3 A Posteriori Cross linking of XML Documents

Of course the user has to access all documents through OMNIS/2 to benefit from the additional features. This can either be done by using the extended search facility of OMNIS/2 which itself searches in the underlying systems and annotates the result sets or by following links in documents which are presented (and therefore have been processed) by OMNIS/2. In any case retrieving new documents is processed as follows. The OMNIS/2 system first queries the local meta database to retrieve the information in which external system the requested documents reside. This requires of course that every document has a persistent unique identifier (UID) within the underlying system. For documents in the local database this can be easily achieved; for other digital library systems it is an requirement so that they can be utilized by OMNIS/2. After returning the answer the meta database is queried a second time to retrieve whether there are links or annotations on the retrieved documents. If so we receive all corresponding anchor information. Anchors in this context follow the definition of the Dexter Hypertext Model [HS94] and in general are an abstraction of link sources and link destinations. Besides their exact position the anchors also need unique identifiers (AIDs) so that they can be associated with a document. After this second query the main part of our a posteriori cross linking is performed. We add the links to the documents and present them to the user.

Merging original XML documents and their linking information can be done with the help of XSLT. The W3C promotes XSLT as a language to transform XML documents into other XML documents, HTML documents or ASCII text. The manipulations of XML are performed by an XSLT processor using an XSLT stylesheet. XSLT processors are available

```
<item>
<dc>
  <identifier>omn:1</identifier>
  <creator> Guenther Specht, Michael G. Bauer </creator>
  <title>
     OMNIS/2 - A Multimedia Meta System for existing Digital Libraries
  </title>
  <subject/>
  <publisher>Springer Verlag, Berlin Heidelberg</publisher>
  <date>2000</date>
</dc>
</item>
```

Figure 7.6: Sample XML document (fragment)

for a variety of host languages (C++, Java, Perl, etc.). XSLT itself though does not provide a way to retrieve data (i.e. links) from a database. It is possible however to use the host language to perform this task and to exchange data with the XSLT style sheet by setting parameters. This combination is the central technique of our idea to utilize a posteriori cross linking in digital libraries. In our case the host language is Java and we chose Xalan [Xal] from the Apache project as the XSLT processor.

In the following sections we present our techniques using a short XML fragment of a document (in this case it is a simple metadata record) (Fig. 7.6). In the example we assume that a link to an annotation should be created which gives further information on the title of the XML document. For optimization reasons we combine two steps into one. Links are added to the documents, while the documents are transformed from XML into HTML for the users' browsers.

7.3.1 Simple Linking

At first we retrieve the link information (with UID being the key) from the meta database via JDBC (as we use Java as the host language). The next step is to build a complete URL which points to the linked document (which contains e.g. additional information). The information on the URL is available from the target anchor. This URL is the parameter which is passed on to the XSLT stylesheet. With the data provided by the parameters the XSLT stylesheet then creates a valid HTML link at the desired position. If the link includes the whole content of a tag then the content can be easily enclosed into the <`A HREF`>...</`A`> hyperlink statement of HTML. The following excerpt (Fig. 7.7) shows part of a simple XSLT stylesheet in detail.

It applies a posteriori cross linking on a <`title`> tag. The parameter which holds the target URL is called `titlelink` and it defaults to the empty string (no link). Its value is set by the Java part of OMNIS/2 at run-time. The resulting title string is enclosed in

```
<xsl:param name="titlelink" select="''"/>
<xsl:template match="title">
 <H3>Title:
    <xsl:if test="$titlelink!=''">
      <A><xsl:attribute name="HREF">
         <xsl:value-of select="$titlelink"/>
         </xsl:attribute>
      <xsl:apply-templates/></A>
    </xsl:if>
    <xsl:if test="$titlelink=''">
      <xsl:apply-templates/>
    </xsl:if>
 </H3>
</xsl:template>
```

Figure 7.7: XSLT stylesheet for simple linking of tags (fragment)

```
<H3>Title:
<A HREF="http://www3.in.tum.de/servlet/omnis2?action=DisplayAnnotation&
id=anno:4">
OMNIS/2 - A Multimedia Meta System for existing Digital Libraries</A>
</H3>
```

Figure 7.8: generated HTML code (fragment)

`<H3>` tags (Fig. 7.8).

7.3.2 Linking Substrings within Tags

If only part of a tag is linked the technique becomes a bit more sophisticated (Fig. 7.9) and we have to address substrings of the tag and enclose those with the `<A HREF>...</A>` hyperlink statement (Fig. 7.10). The anchors have to carry information about the position of the substring within the tag content, but mostly simple integer identification is sufficient for this purpose. In the example below the parameters `start` and `length`, which are set by the Java part of OMNIS/2 at run-time, denote the postition of the link. For each anchor these values are stored in the meta database.

7.3.3 Advanced Linking using Java Extensions within XSLT

The above presented techniques show very simple linking mechanisms. They link only exactly once in a tag and (without fundamental changes) work only for standalone documents. If a list of documents is returned (e.g. as a result of querying a digital library

```
<xsl:param name="titlelink" select="''"/>
<xsl:template match="title">
 <H3>Title:
    <xsl:if test="$titlelink!=’’">
      <xsl:value-of select="substring(., 1, $start - 1)"/>
      <A><xsl:attribute name="HREF">
         <xsl:value-of select="$titlelink"/>
         </xsl:attribute>
         <xsl:value-of select="substring(., $start, $length)"/>
      </A>
    <xsl:value-of select="substring(., $start + $length + 1)"/>
    </xsl:if>
    <xsl:if test="$titlelink=’’">
      <xsl:apply-templates/>
    </xsl:if>
 </H3>
</xsl:template>
```

Figure 7.9: XSLT stylesheet for linking substrings (fragment)

```
<H3>Title: OMNIS/2 - A
<A HREF="http://www3.in.tum.de/servlet/omnis2?action=DisplayAnnotation&
id=anno:4">
Multimedia</A> Meta System for existing Digital Libraries
</H3>
```

Figure 7.10: generated HTML code for linking substrings (fragment)

```
 1 <?xml version="1.0"?>
 2 <xsl:stylesheet
 3     xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
 4     version="1.0"
 5     xmlns:java="http://xml.apache.org/xslt/java"
 6     exclude-result-prefixes="java">
 7  <xsl:template match="title">
 8    <xsl:variable name="titlelink"
 9      select="java:Anchor.getLink(string(../identifier), string(.))"/>
10    <H3>Title: <xsl:value-of select="$titlelink"\> </H3>
11  </xsl:template>
12 </xsl:stylesheet>
```

Figure 7.11: A Java call from an XSLT stylesheet (fragment)

system) the single result items cannot be linked with the above presented methods. Each item can be uniquely identified but with the techniques so far it was not possible to transfer information from the XSLT stylesheet to the host language. (The above shown techniques use only the way from the host language to the XSLT stylesheet.) Another problem is that it is not possible to transform only part of an XML document and then update the variables for the remaining transformations with the XSLT stylesheets we have presented up to now. This is of course tightly linked to the above mentioned problem.

Most XSLT processors offer extensions, which enable the use of programming languages from within XSLT stylesheets. As mentioned above we use the Xalan XSLT processor, which offers Java extensions besides others. With a new namespace defined at the beginning of the stylesheet it is then possible to directly call Java methods in a way similar to XSLT functions. As the Java methods can be called with arbitrary parameters we can transfer parts of the XML data to the Java method, manipulate it (e.g. query a database) and return data to the XSLT stylesheet for further transformations. This gives us the flexibility to work beyond the functionality of simple XSLT stylesheets and to achieve the desired results. For the sake of readability we only provide a very short excerpt of a Java extended stylesheet omiting the source code of the Java classes (which carry the JDBC database calls, etc.). `Line 9` in Fig. 7.11 shows the relevant call to the Java class.

A Posteriori crosslinking of existing XML documents has been published at ECDL 2001 conference [BS01a].

7.4 A Posteriori Cross Linking of PDF Documents

As shown in the previous section XML documents can be a posteriori cross linked by OMNIS/2 by using XSLT with Java extensions. This is conveniently done when the document is transformed into HTML before presentend to the user. Originally OMNIS/2 was desgined to process and cross-link XML documents only. Due to the widespread use of PDF including the support of hyperlinks in the format make PDF another interesting candidate for a posteriori cross linking. This widens the application of OMNIS/2 significantly as also documents which are typically stored in existing digital library systems as full-text or content documents can be enriched with links.

7.4.1 Introduction to PDF

The *Portable Document Format* (PDF) [Ado] is one of the most popular file formats for digital documents nowadays. PDF was developed in parallel to the rise of the World Wide Web and it was designed to be the successor of *PostScript* (PS). Both, PDF and PS, are languages to describe the layout and appearance of digital documents and both are used to prepare pre-print documents which can be reproduced in printers or printing machines but can also be viewed with software tools on monitors, etc. The experiences with PS though showed that it is important to increase the portability of digital documents across different platforms and to make the documents more flexible. This resulted in a change of features but also in a very restrictive change of the available functionality for PDF. While PS is a programming language based on a stack machine, PDF is not. PDF nevertheless offers all functionality to implement interactive applications (interactive documents respectively). Additionally PDF contains many features which are not available in PS. While PS is completely based on ASCII text, PDF offers besides the representation in ACSII text also a compact binary format which can also be used in a compressed representation further reducing the size of digital documents. Compression for graphics and pictures is also available with JPEG, LZW and other compression techniques which can be applied to pictures before they are embedded into the PDF document. In our context it is especially important that PDF also offers support for hyperlinks which can either reference parts of the current document or arbitrary external URLs (we will later discuss this feature in more detail). Another difference targets the handling of fonts. While in PS all fonts have to be embedded into the document, PDF allows to replace (e.g. non available) fonts for display or printing.

7.4.2 Adding Links to Existing PDF Documents

Links in PDF

Links in PDF are based on the concept of annotations in PDF which we are going to explain first. The concept of annotations in PDF is fundamentally different from annotations in OMNIS/2 and it is important that the two are not mixed. While in OMNIS/2 annotations are (arbitrary) metadata for documents (e.g. additional text information, additional graphical information, etc.) annotations in PDF allow to create an association between objects (e.g. sound, movies or text notes) with a specific position inside the document (i.e. a document page). Annotations in PDF are also a way in which the user can interact with the document by accessing additional textual information, playing a movie or a sound, etc. Most annotations in PDF can have two states, *open* and *closed.* A closed annotation is marked on a page in PDF by a special icon, box, line, etc. Closed annotations are transformed into open annotations when users access the annotation and activate it (e.g. by a mouse click). In PDF there are several different types of annotations which all are displayed differently and lead to different results. Some exemplary annotation types are listed in the following:

- Text Annotations. A text annotation is a note which contains text and is displayed in a pop-up window. Like some other annotation types it can have two states, open or closed. When it is closed an icon is displayed at a specific location in the PDF document. The icon can be activated by clicking on it and the associated text is displayed in an open pop-up window.
- Free Text Annotations. This annotation has also associated text. The difference between a free text annotation and a text annotation is that the former does not have two states so that the annotation is always open and the associated text is always displayed on a page.
- Sound and Movie Annotations. Closed sound and movie annotations are marked with an icon which indicates a sound or a movie file. When the annotation is opened it plays either the sound or the movie. Playing the media file is done by a special annotation handler which has to be supplied for special media files. For other standard annotations (e.g. text) PDF viewer applications usually already contain annotation handlers.
- Line, Polyline, and Polygon Annotations. Annotations can also be marked with a certain geometrical form and not with a special icon. This includes clickable lines, clickable polylines and annotations which are displayed as a polygon.
- Link Annotations. Link annotations represent hyperlinks. It can be used for navigating in the current document but it can also be used to represent an action like for example execute a specific application or go to an external URL. When a link annotation is defined we are not talking about the content of the document to which it is associated like with the other annotations but we only define the information about the position of the annotation in the page or if it has a destination (URL) or performs an action (jump to a different page).

A link annotation as explained above is a special type of an annotation and it is not associated with content of the (source) document. It is placed by using an absolute position (a geometric region) on the page. This conforms very well with the concept of OMNIS/2 which adds links like a mask on top of existing documents.

PDF also knows a different concept though. PDF can be used as *Tagged PDF* which means that PDF can be enriched with additional information which allows post-processing of the content. Post-processing includes simple extraction of text and graphics for pasting into other applications, processing text for such purposes as searching, indexing, and spell-checking and making content accessible to visually impaired persons. Of course the additional information must be included into the documents to achieve this (for text word breaks are especially marked so that hyphenation is not a problem, some languages require a very complex handling of this as spelling is sometimes changed for hyphenation). In the case of links s.c. *link elements* are introduced. Link elements explicitly perform the association between content within the document and the link annotation. A link element can be seen as a parent item and the link annotation and the content items as child items.

The Implementation of Cross Linking for PDF

Implementing cross linking for PDF can be performed straight forward. Available techniques like software packages and software libraries [Fac] support such a project. Adding links to PDf documents can also be done by the standard tools from the commercial Acrobat Package provided by Adobe but this does not comply with the requirements of OMNIS/2 where links should be added on the fly to documents right before they are presented to users.

Two variants for a cross linking component for OMNIS/2 are possible. When the exact position of the link annotation is known in advance it is possible to simply add the link annotation on the desired page at the correct position. All information about the position can be stored in a source anchor. The target anchor then stores information about the action for the link annotation (new position within the same document, different document or external URL). Secondly it is possible to add content information to the source anchor which allows the creation of tagged PDF. If this variant is chosen it is important to add additional information about the content to the source anchor as the original content is overstriked by the added link element (properties on italics, bold and underline have to be stored for text). Depending on the included content, the size of the source anchor might increase significantly.

The additional information about annotations and link elements, etc. are not stored in the original PDF document but by extending the document with an incremental update. Incremental updates are supported by PDF so that a modified document does not have to be completely rewritten. This means that the original information in the PDF document is still intact. This feature is important for cryptographically signed documents as the supplied digital signature can be validated with the original document and additions to the document can be uniquely identified. PDF also supports documents which can not be changed after there initial creation. In such a case adding links and annotations to the document is not possible of course. As such a document was explicitly created by the original author this is not considered to be a problem of the cross linking component.

7.4.3 PDF and the Dexter Hypertext Reference Model

The Dexter Hypertext Reference Model [HS94] (often abbreviated as Dexter Model) is an approach to develop a standardized model for hypertext and hypertext systems. The model provides a terminology for hypertext systems and hereby offers methods which allow to clearly show similarities or differences between various hypertext system. As PDF also offers functionality of a hypertext system it is interesting to show how PDF is different or similar to the functionality described in the Dexter model and how it fits into the model.

Figure 7.12 shows the three different layers (Run-Time Layer, Storage Layer, and Within-Component Layer) of the Dexter Hypertext Reference Model. Run-time layer and storage layer are connected by the presentation specification while the storage layer and the within-component layer are connected by anchoring. Both act as an interface between the corresponding layers.

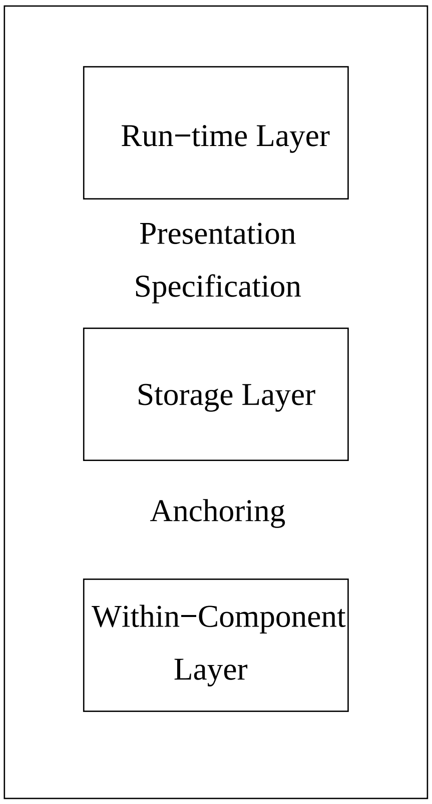


Figure 7.12: Different layers of the Dexter Hypertext Reference Model

Storage Layer

The storage layer deals with the network structure of links and nodes which is the core property of a hypertext system. In other words the storage layer describes a 'database' of the nodes and links. The storage layer is formed by different components which contain the information of the system (e.g. text files, sound files, movie files, etc.). The components can also be seen as data containers. In the Dexter Model a link is also a special type of a node which defines relations between other components. For the storage layer there is no difference in the different types and formats of components.

The mechanism for addressing in the Dexter Model is known as *anchoring*. Anchoring works as an interface between the storage layer and the within-component layer. Anchoring deals with addressing locations or items within the content of individual components.

In PDF a storage layer is created by the format internally. Anchoring is provided by structures which represent a position in a document or actions which represent positions outside a document (different PDF document or external URL). The actual linking is done by link annotations but while one to many links and bidirectional links are possible in the Dexter Model this has not been implemented in current versions of PDF. This could be later integrated though if demand exists. In the Dexter Model components are very abstract but as PDF is an implementation it of course has to deal with different formats. A lot of flexibility is gained though by the fact that certain plugin handlers can be supplied for annotations but also for links and special target formats of links (e.g. playing movies after clicking on a link). This is also very important for the within-component layer.

Within-Component Layer

The within-component layer in the Dexter Model focuses on the content of components as well as on the structure of components. As there are numberless different types of content (graphic, sound, animation, text, etc.) and structure (i.e. different file formats) within a component the within-component layer is not further specified in the Dexter Model. Special hypertext systems which implement (parts of) the Dexter Model of course have to provide support for the different content types and file formats. In PDF a within-component layer is also visible but as PDF can be seen as an implementation of the Dexter Model the layer is not as abstract as in the model. In PDF the available formats can be separated into two main categories. One category is formed by the objects in PDF (i.e. text, images, etc.), which are explicitly stored in the file body of a PDF file and which are directly understood by the viewer applications. The second category are all formats which can be understood by special plugin handlers and which also can reside externally and are not stored in the PDF file. For the internally stored formats the anchoring and linking techniques are functionable. For external formats this is not the case.

Run-Time Layer

Both, storage and within-component layer, deal with the hypertext systems only as a structure of passive data. The reality in the hypertext systems though is that they have to offer facilities to manipulate, access and view the network structure. All this functionality is covered by the run-time layer.

In PDF the run-time layer is formed by the implementation of tools for PDF manipulation and viewing. Popular tools are the Acrobat Reader and the Acrobat package, both provided by Adobe but also freely available tools like the Ghostscript package [Gho] and programming libraries like The Big Faceless PDF Library [Fac] which allow the manipulation of PDF.

Summary

In a final summary overview one must say that many parts of the Dexter Model appear in PDF. In many cases the Dexter Model of course is very abstract and as PDF is an implementation this level of abstraction is lost to a certain extent.

7.5 The Authoring Tool

The authoring tool is a key component of the OMNIS/2 system. It enables users of the system to work with documents as they are used to with common paper documents or books.

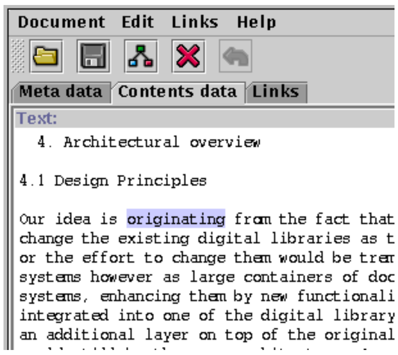
While OMNIS/2 is a server application it is useful to separate the authoring tool from the core system and design it as a client application. As users access OMNIS/2 with the help of a web browser it is also appropriate to use this existing access point for the authoring of documents.

There are several techniques available which allow a web browser to be enhanced by the required functionality. Apparently a simple HTML interface does not satisfy the demands as it does not offer a flexible way to manipulate documents. In addition the only way to communicate with the server is by web formular which is also too restricting. JavaScript enhances HTML by the functionality of a client side scripting language. It does not offer though graphical capabilities which are important for manipulating graphical documents and it does not offer network functionality for communication with the OMNIS/2 server application. While ActiveX would be another candidate to use for the authoring tool it was rejected due to its proprietory properties (only available on Microsoft based platforms) which would have severely narrowed down the usability. It was finally decided to implement the authoring tool as a Java applet which meets exactly the requirements. It offers graphical support for user interfaces, allows network communication and is available on a very large amount of different platforms.

The communication of the Java applet and the core system of OMNIS/2 is performed by exchanging serialized Java objects via HTTP. Before choosing HTTP several other methods to exchange the required information were examined. Using sockets for the communication would have led to a efficient bidirectional communication. Implementing sockets would have created a very complex and time consuming development especially on the server side. In addition sockets usually suffer from restrictions due to firewalls in real-world scenarios. The last argument also targets RMI (Remote Method Invocation) and CORBA (Common Object Request Broker Architecture) based solutions but could be omitted by using HTTP explicitly for this purpose. Both approaches were not an option though as they would have led to a strong link between the client side and the server side and would have made it difficult to add extensions independently from one or the other in later versions.

As mentioned above a communication based on HTTP and serialized objects was used instead. The original object-oriented document model which was presented in Section 7.2 proved to be very valuable in this context. It needed some extensions though as the original design only allowed one access method to the document which output the content of the document in XML for further processing. It was necessary to enhance access to the data of a document and make it more flexible. Otherwise repeated single accesses to information in the document (e.g. author information, title information, etc.) would have resulted in several XML parsing cycles of the same document. To circumvent this, new access methods were introduced into the document model which now allow access to the above mentioned parts of information. In parallel to this extension it was also necessary to separate a document into several units which can be accessed independently from each other. This results in a strong separation between metadata and content data. The newly introduced units are called *media items*. Media items carry four attributes. **Name** is a label for the data in the media item (e.g. author or title for metadata or picture for images). **Type** defines the media type of the item. **Value** holds the actual media content data while **Id** is unique identifier which allows to distinguish media items of a document. With this separation it was also necessary to enhance the anchor and linking model. The basic separation into different anchor types remains but the anchors are no longer tied to different document types but to media items. This allows the linking of both metadata and content data which was not available in the former version of the anchor and document model.

Another extension was required for the content of documents. External documents retrieved their content on demand whenever the document was displayed. On the client side this original concept is very problematic as a loading of arbitrary data from other sources than the originating server is forbidden by the security restrictions of Java Virtual Machines in browsers. To comply with the demands of the authoring system documents (media items) now carry their complete information and do not provide a skeleton which is later filled on demand when the document is displayed.



```
  4. Architectural overview

4.1 Design Principles

Our idea is originating from the fact that
change the existing digital libraries as t
or the effort to change them would be trem
systems however as large containers of doc
systems, enhancing them by new functionali
integrated into one of the digital library
an additional layer on top of the original
```

Figure 7.13: Screenshot of the authoring tool with marked linking area

Figure 7.13 shows a screenshot of the authoring tool with a marked linking area. The authoring tool supports mouse based working with icons and pull-down menus. It also offers an undo feature.

7.6 Going towards an OpenArchives Compatible System

7.6.1 The OpenArchives Initative

The goal of the OpenArchives Initative (OAI) [LdS01, OAI] is to establish interoperability among digital libraries. The OpenArchives Initiative emerged at Carnegie Mellon University as a proposal of Carl Lagoze and Herbert van de Sompel. The initiative wants to open digital library systems so that the offered data is accessible by other systems, which would allow external usage of the data. The first proposals of OAI exclusively considered e-print archives (grey literatur, etc.). Due to the success of the first proposals the content of the systems was continously extended and now includes various sorts of digital content.

The OAI uses a strict separation into *data providers* and *service providers*. Data providers are systems which store data and offer access to this data conforming to the OAI. In contrast to this, service providers offer services by utilizing the offered data in some form. Already offered services include searches over various archives and filtering and recommendation of records [OAI]. To be able to process the data the service providers collect the metadata of existing data providers (s.c. harvesting) and store the metadata locally (s.c. mirroring). The harvested data can then be searched locally and services on this data can be offered.

The similarities to OMNIS/2 are apparent. Both proposals utilize existing data to offer services for end users. Both approaches use XML as an exchange format. There are also differences though. The OAI is targeted towards metadata harvesting while for OMNIS/2 the data still resides in the original systems and is retrieved on demand at run-time. The basic focus of both systems is significantly different though. The OAI only proposes standards for the interoperability of existing systems but does not favour an implementation. OMNIS/2 in contrast is a specific implementation which is similar to OpenArchives compatible systems.

7.6.2 Implementation of the OpenArchives Concepts

Due to the strong similarities in the basic concepts we have decided to extend OMNIS/2 by the additional functionality of an OAI service provider [SB03]. To be able to harvest OAI compatible data providers we used the OAI harvester of Jeff Young (OCLC) [ALC]. The OAI harvester allows to arbitrarily process the harvested XML data before storage. In our case we prepare the XML data for storage in the relational database system of OMNIS/2. All queries on OAI data are then performed locally on the harvested data. Accessing OAI data through OMNIS/2 is identical to the techniques which are presented in the previous chapters. The data is transformed into the internal OMNIS/2 XML format and enriched with links and annotations before it is presented to the user. The OAI and OMNIS/2 format are very similar as they are both based on Dublin Core. The structure of OAI XML data also is very flat, which makes it similar to the data from DBLP (Section 4.5). When storing the data with our presented method then the considerations about flat documents are important.

Chapter 8

The Dualism of Associative and Navigational Search

8.1 Introduction

The purpose of this chapter is to discuss the differences and similarities between different methods of exploring a hypermedia system. We will especially concentrate on problems which occur when following links (s.c. navigational browsing) and show some approaches to solve the problem of changing and moving documents in a hypermedia application like the World Wide Web. While the WWW acts as a good example, most of the results are also valid for hypermedia systems in general. In the final section in this chapter we will summarize a possible application of the proposed techniques in OMNIS/2. In future versions OMNIS/2 as a hypermedia meta system should be able to handle changing and moving documents.

The discussion about design and implementation details of OMNIS/2 showed subtle differences between following links from one web document to another and searching in web based digital library systems. There apparently seems to be a remarkable difference between the two methods of exploring hypermedia systems which also results in different implementations for the two approaches.

Following links from one node to another is the classical way of exploring hypermedia systems. This is commonly known as navigation in hyperspace. As users are looking for information in hyperspace we have decided to call this method of finding information *navigational search*. We call the second method of searching *associative search*. It is associative because users formulate predicates by giving keywords, content or similarity measures. Sometimes the predicates are formulated implicitly by filling out fields in a formular, etc. Search facilities (search engines, etc.) then return result lists with links to documents which are associated with the input of the user. The result lists are of course based on the data of the search facilities and also based on their algorithms for answering queries. Search facilities are especially important when the hypertext system exceeds the size of a browsable structure (e.g. millions of documents). In these cases they are quite often the only approach to find valuable information in the hypermedia system.

Valuable information though can also be found by using navigational search. For this purpose large collections of links are created and grouped and ordered by topic. This task is usually done by humans with experience in cataloging similar to the task of librarians. Accessing the collection usually starts from a root entry point and users continue to navigate through the collection (following paths created by the collectors) until they have found the information they were looking for.

8.2 Associative and Navigational Search in Detail

8.2.1 Associative Search

Associative search is mainly based on techniques from information retrieval. As information retrieval is not a topic of this work we will concentrate in the following chapter on navigational search and will explain associative search briefly. We will see later though that associative search can be helpful if the link structure of a hypermedia system is no longer intact.

Offering associative search requires a collection of data in a first step. In a WWW based environment the data is collected by crawlers, web spiders or harvesters using navigational search. After harvesting, the collected data then has to be stored and indexed. The storage and indexing approaches differ depending on the algorithms which are used in the retrieval stage later. The stage of collecting data though is mostly identical for different search engines. Some optimizations can be made to speed up this step as we will see when we discuss navigational search. The quantity and quality of the collected material directly influences the results of associative search but the main differences among different search facilities come from the used retrieval and ranking techniques on the collected data. In the following we will concentrate on text and text phrases and neglect other media formats. The most widely used methods currently exist for text while other media formats are still a basis for intensive research.

A very important feature which is required for associative search is *ranking*. Especially when handling large result sets it is not sufficient to return the results one by one. The results have to be ordered according to a metric which reflects the relevance of the results for the user. The suitabilty of a ranking method usually differs from user to user but there are some general ranking methods which proved valuable and are mentioned here. A very intuitive and simple ranking would return documents at the top of a result list which contain a search term with a high frequency. It was shown though that a ranking which is based on linking is more appropriate as linking implicitly gives information about the importance of a page or document in a hypermedia system. If many links point to a page or document, then the page (or document) is considered to be important. The PageRank algorithm used by the Internet search engine Google [BP98] is based on the linking information which is inherently available on the World Wide Web. The PageRank algorithm is more complex though. Besides the occurence of the search phrase on a web page, it also takes into account if a page which links to a page is itself considered to be important. Links from important pages to a page are considered to be more relevant than links from unimportant pages. The importance of a page in the PageRank algorithm is also increased if a page is often updated and if the link does also contain the search phrase. While methods which manipulate frequency schemes are very apparent by adding words to a page, attacks on the PageRank algorithm are also possible. Adding a link to page X (containing a certain phrase) on many different pages which change fast enough and are also linked from important pages (in respect to PageRank) leads to an increased ranking for a desired phrase on page X. This method is also known as *Google Bombing* and severly declines the quality of search results if widely used.

8.2.2 Navigational Search

Following links seems to be very intuitive as it is the classical method of exploring hypermedia systems. When understanding the limitations and obstacles of navigational search it is necessary to see linking a little different from the publicly accepted viewpoint of following a link to a new document or document position. As mentioned earlier in Section 7.4 the Dexter model explains links as link relation which binds a source anchor and a target anchor. Both, source anchors and target anchors, define a position. In HTML a link is expressed with a tag similar to the following `<A HREF="target.html#pos">Source anchor</A>`. The source and target anchors are hereby given in the HTML statement. The position of the source anchor is defined by the position of the `<A HREF="">` element in the HTML document. In the above example the position of the target anchor is referenced via a target HTML document and a position `pos`. The position of the target anchor is an HTML element named with a fragment identifier (`pos` in this case) [HTM].

The positions of source and target anchors are fundamental for the integrity of a hypermedia system. This is especially important for dynamic documents and will be further discussed in Section 8.3.

While following links is the classical method of exploring hypermedia systems, it is also simple enough to be performed automatically by crawlers, harvesters and (web) spiders. These tools are used by search engines and web archives to mirror an existing site and to collect data on a client based approach. Due to the client approach apparently not all data on the servers can be accessed. Especially the “hidden web” which is stored behind search forms in large searchable electronic databases cannot be accessed. While there are approaches to overcome problems of the hidden web by analyzing web pages and filling out forms automatically [RGM00], one also has to be cautious for simple harvesting approaches. Simple harvesting can lead to very scattered results if the harvesting of a site is interrupted because a link is followed to another system and the original site is visited later on. While documents on a web site apparently can not be harvested simultaneously a long delay increases the probability of an inconsistent data base, as documents and link structure usually change over time. This can be avoided by sorting links according to their targets so that links pointing to local targets (relative to the current page) are visited first. When links of already visited pages additionally are removed then this further speeds up the harvesting process. It has to be taken care of though that the load created on the remote server by careless harvesting can be higher by several factors than the load which is created by human users. Especially the simultaneous harvesting of all links pointing to local pages which are found on a harvested page can create high loads on the harvested servers and severely affects performance of the remote server. Although the two goals (fast local harvesting with low server load) seem orthogonal it can be achieved by introducing delays between harvesting pages which lowers the server load significantly. The delay though has to be chosen with care so that harvesting performance is not decreased. Another important aspect is that a harvester also needs to avoid circles in the web structure. A duplicate check on URLs is not a criteria for this as due to a bad link structure it can happen that a URL is appended by identical path fragments and retrieves identical pages from different URLs. Possible approaches are a duplicate detection for documents and for path fragments of an URL. URL length can also be limited to avoid running into circles.

8.3 Advanced Problems of Navigational Browsing

Above we already briefly mentioned problems when documents are changed or when documents are moved to other locations. Since these operations can damage the link structure we show some approaches on how to overcome navigational problems for changed and moved documents.

8.3.1 Changes in Documents

As throughout this work we only considered hierarchically structured documents we continue this tradition in this section on changes. In a first step, it is necessary to identify the different methods of changes on documents which can occur. For us, changes can be split into structural changes and content changes.

1. Structural changes include insertion and deletion of nodes in the hierarchical structure of documents. Insertion of nodes can either mean insertion of one node or insertion of a subtree. In most hierarchical documents (also in XML based documents) single nodes are identified with node identifiers by algorithms. If the hierarchical structure also represents the order of the document (as in XML) and if this order is represented by the node identifiers then the insertion of a node or a subtree can lead to a complete renumbering of node identifiers. If nodes or subtrees are deleted then a renumbering is not necessarily required if the order is still valid. Structural changes can become quite complex when node or subtrees are repositioned within the hierarchy.

2. Content changes are changes which affect the content of nodes. As we restrict content to text content in this chapter, the changes which can occur are insertion and deletion of characters. The position of characters is usually given by their position in text. For both, insertion and deletion, the position of characters in a node following the position of the modification is changed.

To further discuss the problem of changes in documents the notion of source and target anchors is again convenient. When anchors are given in documents as they are with HTML documents then changes to a document also change the position of anchors. For source anchors the position is given by the position of the link element in the HTML source code. The position of target anchors in HTML is twofold. As shown above, the target anchor consists of a target URL and an optional identifier tag which is changed accordingly to changes in the document. If the identifier itself is removed or changed so that a dangling reference is created then the default behaviour in most browsers is to just follow the link to the referenced page. No further positioning in the document is then performed as the position of the anchors is given by syntax and not by the semantic of the underlying documents.

When links are added to documents at run-time as it happens in OMNIS/2 then the exact position of an anchor is very important. The anchors are stored separately from the documents with certain positions or identifiers of positions. The tools and modules which add anchors to documents do this on the basis of the given positions. If the underlying content or structure is changed but the positions in the anchors are not, then depending on the changes the link is placed at the wrong position or can not be positioned at all. Various approaches to make anchor positions robust against changes have been proposed. In the following we will outline some of them which showed to be suitable for OMNIS/2.

8.3.2 Making Anchors Robust Against Changes

In this section we are going to sketch four proposed solutions to make anchor positions robust against changes in documents. While the first approaches are very simple the last ones use quite sophisticated algorithms from bioinformatics. The starting scenario is equivalent for all approaches. We assume an anchor with a position in the original document and a modified document, where the modifications are unknown. The task is to place the anchor at a position in the new (modified) document which is semantically equivalent to the position in the old (original) document. There is a restriction apparently because when the original position is removed it is impossible to find the position, which requires to treat the link with a default behaviour (omiting to display the link or rendering the link in a special frame or window for “lost links”). The four approaches which we present in the following share the same basic principles. They are all based on a position of anchors and a heuristic to find the anchor position again in the modified document. Although sharing similarities, the approaches also vary in the principles as they use different formats for the position of anchors and different heuristics.

1. Storage of Content

Anchors can be stored by just giving the character position in a document. This solution is simple and works as long as the document is not changed. A very simple and intuitive approach to make anchors robust against changes is to additionally store the content at the position where the anchor should be placed. Positioning the anchor for a modified document requires a heuristic to place the anchor especially when the content appears more than once in the document. A local search near the original character position in the document should be favoured of course. The heuristics can be further improved when more (redundant) content information is stored with the anchors position (like text fragements before and after the actual anchor position). A technique similar to this has been proposed in the work on the Clio project (Section 6.4.3 and [LMR99]).

2. Storage of Paths and Content

The above mentioned approach using just content can be further improved by additionally storing the path to the character position of an anchor including the content [PW00]. The path then enables to position the anchor again in a modified document. Changes in the structure of the documents are no problem as long as the path to the anchor position is intact. If additional nodes are inserted with the same tags at the same level in the hierarchy then an heuristics can used to reposition the anchor by examinig the following and preceding siblings. The algorithm can also check on levels above or below the position where the stored path differs from the path in the document.

3. Identifying Changes with Signatures

There has also been an approach to detect changes between XML documents and use this information to reposition anchors. As published [CAM02] signatures are used for nodes and subtrees. In a first step each node in the original document is provided with a persistent unique identifier (the signature) consisting of its content and the signature of its child nodes. Changes are detected by a *diff* algorithm which matches node identifiers from a modified and an original document version. The detected changes (insertions, deletions) are then collected in s.c. *deltas*. The deltas are a representation of the differences between the documents in compact form. The approach especially concentrates on detecting move operations where nodes or content appears in a different subtree. The novelity of the signature approach is that changes can be detected in linear time in contrast to other methods to detect changes in documents. The linearity is hereby achieved by not giving the shortest deltas (i.e. the minimum number of changes in the documents).

If anchor positions are given based on signatures then the relevant nodes can be found easily by the above heuristic which tries to find nodes based on signatures. If paths are used then the generated deltas have to be used to calculate a new version of the path.

4. Identifying Structural Changes with Algorithms from Bioinformatics

This approach is similar to the approach which uses signatures. The algorithms which are used for change detection are very sophisticated though. The approach is based on the fact that hierarchical documents we have been talking about in this work share a remarkable resemblance with secondary structure of RNA (ribonucleic acid). Similar to DNA (deoxyribonucleic acid) RNA is a sequence of nucleotids (bases) over a four letter alphabet and transmits genetic information (Figure 8.1). The used bases are adenine,

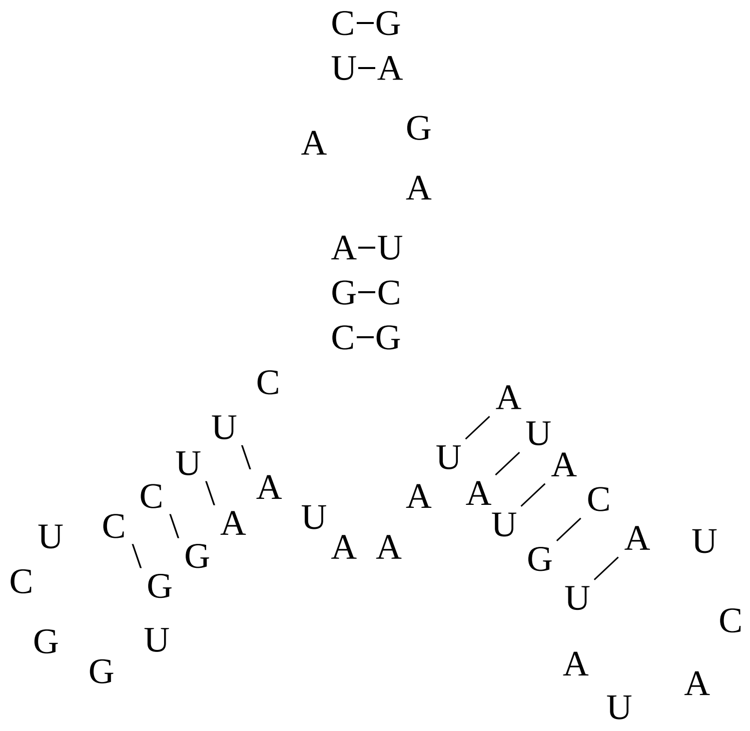


Figure 8.1: RNA secondary structure

cytosine, guanine and uracil. The alphabet consequently is $\{A, C, G, U\}$. The sequence of these bases is called primary structure. The secondary structure is a set of base-pairs which formed bonds between $A - U$ and $C - G$ and is represented as a tree in biology and bioinformatics (Figure 8.2). A research issue in biology and bioinformatics is to find similarities among two RNA secondary structures because the similarities hint a similar biological function of the two RNA sequences [Zha98]. Similarity is hereby defined as minimum cost edit distance (a minimum list of change operations), computing longest common subsequences or calculating a shortest common super-sequence. The resemblance between RNA and hierarchically structured documents is shown in Figure 8.3 where we show how the tree representation of RNA secondary structure can be easily converted into XML with the base-pairs and bases being transformed into tags.

This resemblance allows to use algorithms from bioinformatics for the similarity problem and helps to find the positions of anchors in the changed hierarchical documents. The used methods are euqivalent to the signature based approach. The approach though requires the storage of the original unchanged document and the anchor positions. It is not possible to detect the changes if only a changed document is available.

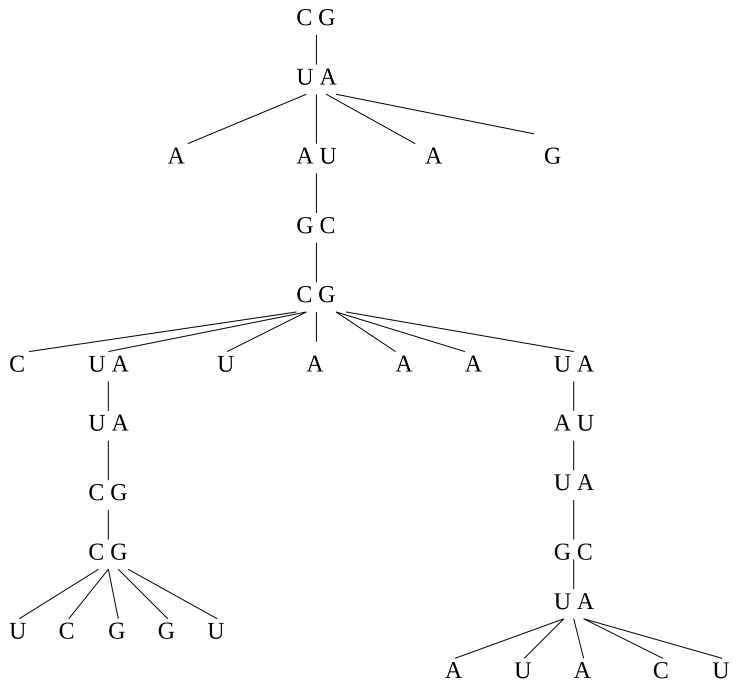


Figure 8.2: RNA secondary structure in a tree representation

```
<CG> <UA>
  <A/>
  <AU> <GC> <CG>
    <C/>
    <UA> <UA> <CG> <CG>
      <U/> <C/> <G/> <G/> <U/>
    </CG> </CG> </UA> </UA>
    <U/> <A/> <A/> <A/>
    <UA> <AU> <UA> <GC> <UA>
       <A/> <U/> <A/> <C/> <U/>
    </UA> </GC> </UA> </AU> </UA>
  </CG> </GC> </AU>
  <A/> <G/>
</UA> </CG>
```

Figure 8.3: RNA tree as a well-formed XML document

8.3.3 Moving of Documents

Besides changed documents another problem are documents which are moved within a hypermedia system. In the WWW just moving a document to a new URL breaks the link structure. Visitors of the old position are usually not automatically forwarded to a new URL if not provided by the document maintainer or server administrator who change links or enable forwarding. This leads to the well-known *404 error (Document not found)* in the WWW. HTML provides a forwarding mechanism based on meta tag (among other solutions). Modern webservers allow URL rewriting where incoming requests are rewritten to new addresses. Systems which use databases can solve the problem of broken links by moved documents when moving a document also updates the target anchors in the database. This is done in the systems OMNIS/2 and Hyperwave which enable link integrity by referential integrity in the database system.

Changing links due to moved documents is also a problem for archives of the World Wide Web. Efforts to build archives of the WWW are developed for some time now. One example is the Internet Archive which is a digital library of Internet sites and other cultural artifacts in digital form [IAr]. The Internet archive regularly harvests selected websites, stores them in its database and makes them searchable. The archive is special as it is not an archive which stores only one version of a webpage but all versions it harvested over time, hereby creating a temporal archive. All stored webpages can be searched and visited as they are available in the archive. Links to other pages which are available in the archive are rewritten to their new address in the archive. To achieve this, it is required to check every page which is inserted into the archive whether the page is referenced from a page in the archive. If this is the case then the link in the archived page has to be rewritten. An efficient implementation requires an index on all external links which occur on pages within the archive. This enables a fast retrieval of the relevant pages (or page ids) and allows to change the links to the newly stored page. Otherwise inserting new documents into the archive might become a bottle-neck of the system as all documents would have to be checked for links. In OMNIS/2 indexes on links are available as links are stored separately from the documents.

8.3.4 Making Documents Findable

Making documents findable does not mean storing documents in a search engine so that they can be found. We understand the term in a way that documents can be found again even if they were moved to a new location in a hypermedia system (e.g. the World Wide Web). In this section we present two methods of making documents findable. The first method discusses persistant identifiers, while the second presents a way to make URLs robust against changes.

1. Persistant Identifiers

In an ideal case documents would carry a unique persistent identifier given to them by a naming authority. A name resolution authority keeps a database of identifiers and location addresses. Retrieving a document can then be performed by querying a name resolution authority with the identifier which then returns the exact address of the document. A fundamental issue apparently is to keep the database of the name resolution authority up to date. It is usually considered that the database can only be kept in an up to date state if the documents have a certain value which is important to be preserved otherwise maintenance will be neglected.

URLs as they are used in the World Wide Web do not conform with the above mentioned procedure as they are only addresses and not identifiers (there is no global naming authority in the WWW). It is possible though to use persistant URLs through an additional layer of indirection. Hereby a special (persistant) URL of a special provider is used which automatically redirects browsers to the desired page. The association between the persistant URL and the URL address is usually maintained by the administrator of the document.

This technique though is usefull mainly to get access to web sites via a main address or single documents and not for large-scale sites where every link has to be stable. A service based on this technique is available with PURL [PUR] from OCLC.

2. Robust Locations

The idea behind robust locations [PW00] is to add a number of well chosen terms to a URL which uniquely identify a document. The terms are called *lexical signatures.* Retrieving the document is then based on two phases. In a first phase a document a retrieved by a client commonly by using the URL (without the signature). If the document can not be returned because of a 'document not found' error then the second phase is performed. In the second phase the client asks a search engine with the given signature. When the terms of the signature carefully chosen, then exactly one document is returned. Choosing the right terms is difficult as certain constraints have to be met. It is usually not sufficient to choose a single term for a document. While it might identify a document in a very short and unique manner, a single term is very sensitive to changes and modifications. Especially when the single term is actually based on a typographic error a correction of the error renders the signature useless. It is stated in the literature [PW00] that according to empirical results approx. five terms are sufficient to uniquely identify a document with the help of a search engine.

It is important to note at this point that associative search (signatures fed into search engines) can be used to ensure navigational search as it can support an intact link structure. And while we have shown that associative and navigational search are contrary, navigational search clearly benefits from techniques developed for associative search.

8.4 Impact on OMNIS/2

OMNIS/2 as a metasystem for existing digital libraries offers all functionalities of a hypermedia system. The scenario of OMNIS/2 is quite complex though. OMNIS/2 sits on top of existing digital library systems and adds links and annotations to documents which reside on external systems. As the external documents are not stored in OMNIS/2 but only referenced it is important that the documents can be found in the external systems. If documents are moved in the external systems or if their unique persistant identifier is changed without notifying OMNIS/2 then the link structure is damaged and the functionality of OMNIS/2 is restricted. The above mentioned methods for moving documents would therefore be very valuable for OMNIS/2. The most promising approach uses lexical signatures but as it is required to get a global view on documents in the WWW a global search engine is required for this.

Links are added to documents by OMNIS/2 at run-time with the anchor information stored separately in the meta database. There are two possibilities of underlying documents, internal and external. As internal documents are stored in OMNIS/2 and can be monitored, the position of anchors can be automatically updated when documents are edited using the authoring tool. External systems, which can not be controlled by OMNIS/2, can change the documents (correcting typographical errors, adding text, etc.) arbitrarily. This also restricts the functionality of OMNIS/2 as anchors are added at wrong positions when the underlying document changes. To avoid this, the above mentioned techniques on robust anchors can be utilized to ensure anchors at correct positions. Utilizing one of the methods would only require minimal local changes to the system as the functionality could be implemented by enhancements in the module for linking documents. From the proposed methods it might be sufficient to store path information and content as the sophisticated methods require storing the original document including the anchor information which causes a large overhead if a high number of documents should be handled.

Chapter 9

Summary and Future Work

9.1 Summary

In the previous chapters of this work we have discussed topics from the development of a modern meta system for existing digital library systems and introduced techniques to store XML in relational database systems. We have seen that digital libraries in many variants have found their way into the daily work of scientists and other users. Modern digital library systems though can be still seen as large containers with powerful query languages. It is not possible for users to work with documents from these systems as they are used to from documents printed on paper. The presented meta system OMNIS/2 enhances the existing systems in a meta layer on top of the existing digital library systems. This design allows to offer features to users like annotations of documents and cross-linking of documents without changing the original systems. All information which is required to enrich the documents is stored in a separate relational database (called meta database) and added at run-time to the documents when they are retrieved from the existing digital library systems to be presented to the users. As OMNIS/2 does not (and can not) control documents which reside in external digital library systems the adding of links to existing documents is sensitive to changing and moving documents. We examined methods which could help to ensure link integrity for source and target documents and gave recommendations.

Besides being a meta system, OMNIS/2 is also a complete stand-alone digital library system. It offers additional storage and indexing of user-defined multimedia documents and harvested documents from external systems.

The usage of XML in OMNIS/2 as a presentation and exchange format for external and internal documents allowed a standardized way for the handling of documents. The necessity to store XML documents (e.g. harvested documents from the OpenArchives Initiative but also user-defined documents) in the meta database requires storage techniques for XML and semistructured data in relational database management systems. We have proposed a numbering scheme for the graph of an XML document based on the technique of simplified Multidimensional Hierarchical Clustering (MHC). MHC allows to store paths in

XML documents in relational database systems in a compact and order preserving method. We have shown that all navigation axes of XPath can be processed on our representation. For some axes (parent, ancestor) our implementation requires User Defined Functions but the calculation can be solely performed in main memory. If an implementation of MHC with bitstrings is chosen then optimizers of relational database systems require knowledge about the efficient processing of prefixes on bitstrings and the hierarchical structure of XML documents which results in handling prefixes as ranges. The proposal of MHC led us to the development of a multidimensional mapping approach to store XML documents in a relational database management system. The multidimensional mapping allows to use simultaneous navigation in documents and the use of multidimensional indexes. Based on a novel classification of XQuery the existence of restrictions on structure and values in typical XML queries showed that it is important that structure and values have to be recognized and considered when chosing indexes for storage of XML in relational database management systems.

9.2 Future Work

The area of XML storage and indexing is still evolving with new proposals for efficient approaches. The separation between native systems (using special index structures) and approaches which use (object-)relational systems will remain in the future. Due to the market share of (object-)relational systems it will be difficult though for special systems which use special index structures to get into the market. Mature systems for XML storage also require techniques for locking when updates are allowed. Locking and updates in the systems are not well discussed in the available literature on special index structures up to now. So it is expected that new proposals of indexes will deal with these special XML problems. Future developments are to be expected in parallel of developments for XML manipulation languages which handle updates, insertions, and deletions[1].

The approaches for XML storage using established database technology do not suffer from these problems as severe as the native systems. Mapping schemes use underlying database technology which already provides locking mechanisms. Some problems might occur though for inserting XML. The problems for numbering schemes are to preserve the properties without renumbering the complete structure which declines performance. As we have mentioned briefly our numbering scheme is more appropriate for inserts than others.

Research on variants of our proposed simplified MHC mapping scheme might also prove to be valuable. Variants might include a bottom-up numbering scheme which might enable a better encoding of equivalent subtrees or different implementations not based on bit strings. Such an implementation requires a different representation of the limits of single surrogates though. As seen already an implemention with variable length surrogates made the required algorithms more complex. It is also important to consider that an implementation has to be supported by the optimizer of the underlying database system.

[1]As of this writing no final recommendations for XML languages dealing with updates, insertions, and deletions are available.

While OMNIS/2 as an implementation of a metasystem for digital libraries can be used as a basis for the implementation of other XML mapping and indexing schemes, it can also be extended towards other future developments in the field. Special predictions are extremely difficult but the system might benefit from developments for the Sematic Web [Sem]. The Semantic Web will make it easier to share and reuse data on the Web. It might also have impacts on methods which try to ensure link integrity for moving and changing documents. Such methods might have a better performance although the distributed nature of the Web will continue to exist.

Bibliography

[ABS99] Serge Abiteboul, Peter Buneman, and Dan Suciu. *Data on the Web. From Relational to Semistructured Data and XML.* Morgan Kaufmann Publishers, 1999.

[ACM] The ACM Digital Library, http://www.acm.org/dl/.

[Ado] Adobe Systems Incorporated. *PDF Reference*, 3rd edition.

[ALC] ALCME, Advanced Library Collection Management Environment, OAIHarvester, http://alcme.oclc.org/.

[Ann] Annotea on Mozilla, http://annozilla.mozdev.org/.

[AQM$^+$97] Serge Abiteboul, Dallan Quass, Jason McHugh, Jennifer Widom, and Janet L. Wiener. The Lorel Query Language for Semistructured Data. *Int. J. on Digital Libraries*, 1(1):68–88, 1997.

[BMMM95] Charles Brooks, Murray S. Mazer, S. Meeks, and J. Miller. Application-specific proxy servers as http stream transducers. In *Proceedings of the 4th International World Wide Web Conference*, Boston, MA, USA, December 1995.

[BP98] Sergey Brin and Lawrence Page. The Anatomy of a Large-Scale Hypertextual Web Search Engine. *Computer Networks*, 30(1-7):107–117, 1998.

[BRB03] Michael G. Bauer, Frank Ramsak, and Rudolf Bayer. Multidimensional Mapping and Indexing of XML. In *BTW 2003, Datenbanksysteme für Business, Technologie und Web, Tagungsband der 10. BTW-Konferenz, 26.-28. Februar 2003, Leipzig*, volume 26 of *LNI*, pages 305–323. GI, 2003.

[BS01a] Michael G. Bauer and Günther Specht. Enhancing Digital Library Documents by A Posteriori Cross Linking Using XSLT. In *Research and Advanced Technology for Digital Libraries, 5th European Conference, ECDL 2001, Darmstadt, Germany, September 4-9, 2001, Proceedings*, volume 2163 of *Lecture Notes in Computer Science*, pages 95–102. Springer, 2001.

[BS01b] Michael G. Bauer and Günther Specht. The Anchor and Linking Concept of a Meta System for Existing Digital Libraries. In *NetObjectDays 2001 - Object-Oriented Software Systems*, pages 260–265, Erfurt, Germany, October 2001.

[BS01c] Michael G. Bauer and Günther Specht. The Object Oriented Document Model of a Meta System for Existing Digital Libraries. In *12th International Workshop on Database and Expert Systems Applications (DEXA 2001), 3-7 September 2001, Munich, Germany*, pages 933–936. IEEE Computer Society, 2001.

[Bus45] Vannevar Bush. As we may think. *Atlantic Monthly*, 176:101–108, 1945.

[CAM02] Gregory Cobena, Serge Abiteboul, and Amélie Marian. Detecting Changes in XML Documents. In *Proceedings of the 18th International Conference on Data Engineering, 26 February - 1 March 2002, San Jose, CA*. IEEE Computer Society, 2002.

[CGMH$^+$94] S. Chawathe, H. Garcia-Molina, J. Hammer, K. Ireland, Y. Papakonstantinou, J. Ullman, and J. Widom. The TSIMMIS Project: Integration of Heterogeneous Information Sources. In *Proceedings of IPSJ Conference, Tokyo, Japan*, pages 7–18, October 1994.

[Cit] CiteSeer.IST, Scientific Literature Digital Library, http://citeseer.ist.psu.edu/cs/.

[CSF$^+$01] Brian Cooper, Neal Sample, Michael J. Franklin, Gisli R. Hjaltason, and Moshe Shadmon. A Fast Index for Semistructured Data. In *VLDB 2001, Proceedings of 27th International Conference on Very Large Data Bases, September 11-14, 2001, Roma, Italy*, pages 341–350, 2001.

[CX00] Josephine M. Cheng and Jane Xu. XML and DB2. In *Proceedings of the 16th International Conference on Data Engineering, 28 February - 3 March, 2000, San Diego, California, USA*, pages 569–573. IEEE Computer Society, 200.

[DBL] DBLP, Uni Trier, http://dblp.uni-trier.de/.

[DFS99] Alin Deutsch, Mary F. Fernandez, and Dan Suciu. Storing Semistructured Data with STORED. In *SIGMOD 1999, Proceedings ACM SIGMOD International Conference on Management of Data, June 1-3, 1999, Philadelphia, Pennsylvania, USA*, pages 431–442. ACM Press, 1999.

[DHW97] Martin Dörr, Hachim Haddouti, and Stephan Wiesener. The German National Bibliography 1601-1700: Digital Images in a Cooperative Cataloging Project. In *4th International Forum on Research and Technology Advances in*

Digital Libraries (ADL '97), May 7-9, 1997, Washington, DC, pages 50–55. IEEE Computer Society, 1997.

[Fac] The Big Faceless Java PDF library, http://big.faceless.org/products/pdf/.

[FK99] Daniela Florescu and 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.

[GHJV94] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: elements of reusable object-oriented software*. Addison-Wesley Publishing Company, Reading, Massachusetts, 1994.

[Gho] Ghostscript, an interpreter for PostScript and PDF, http://www.ghostscript.com/.

[Gru] Personal communication with Torsten Grust.

[Gru02] Thorsten Grust. Accelerating XPath Location Steps. In *SIGMOD 2002, Proceedings ACM SIGMOD International Conference on Management of Data, June, 2002, Madison, Wisconsin, USA*. ACM Press, 2002.

[HBvR94] Lynda Hardman, Dick C. A. Bulterman, and Guido van Rossum. The Amsterdam Hypermedia Model: Adding Time and Context to the Dexter Model. *Communications of the ACM*, 37(2):50–62, February 1994.

[HS94] Frank G. Halasz and Mayer D. Schwartz. The Dexter Hypertext Reference Model. *Communications of the ACM*, 37(2):30–39, February 1994.

[HTM] HyperText Markup Language (HTML), http://www.w3.org/MarkUp/.

[IAr] The Internet Archive, http://www.archive.org.

[KK01] José Kahan and Marja-Riitta Koivunen. Annotea: An Open RDF Infrastructure for Shared Web Annotations. In *Proceedings of the 10th International World Wide Web Conference*, Hong Kong, May 2001.

[LdS01] Carl Lagoze and Herbert Van de Sompel. The Open Archives Initiative: Building a Low-Barrier Interoperability Framework. In *ACM/IEEE Joint Conference on Digital Libraries, JCDL 2001, Roanoke, Virginia, USA, June 24-28, 2001, Proceedings*, pages 54–62. ACM, 2001.

[LMR99] Sarah M. Luebke, Hilary A. Mason, and Samuel A. Rebelsky. Annotating the World Wide Web. In B. Collis and R. Oliver, editors, *EdMedia 99 World Conference on Educational Multimedia, Hypermedia, and Telecommunications*, June 1999.

[MAG$^+$97] Jason McHugh, Serge Abiteboul, Roy Goldman, Dallan Quass, and Jennifer Widom. Lore: A Database Management System for Semistructured Data. *SIGMOD Record*, 26(3):54–66, 1997.

[Mar99] Volker Markl. *Processing Relational Queries using a Multidimensional Access Technique.* PhD thesis, DISDBIS, Band 59, Infix Verlag, 1999.

[Mau96] Hermann Maurer. *Hyper-G now Hyperwave: The Next Generation Web Solution.* Addison-Wesley Publishing Company, Harlow, England, 1996.

[Moz] Home of Mozilla, http://www.mozilla.org/.

[MRB99] Volker Markl, Frank Ramsak, and Rudolf Bayer. Improving OLAP Performance by Multidimensional Hierarchical Clustering. In *Proc. of IDEAS Conf., Montreal, Canada*, 1999.

[MS99] Tova Milo and Dan Suciu. Index Structures for Path Expressions. In *Database Theory - ICDT '99, 7th International Conference, Jerusalem, Israel, January 10-12, 1999, Proceedings*, volume 1540 of *Lecture Notes in Computer Science*, pages 277–295. Springer, 1999.

[OAI] The Open Archives Initiative, http://www.openarchives.org/.

[PGMW95] Yannis Papakonstantinou, Hector Garcia-Molina, and Jennifer Widom. Object Exchange Across Heterogeneous Information Sources. In Philip S. Yu and Arbee L. P. Chen, editors, *Proceedings of the Eleventh International Conference on Data Engineering, March 6-10, 1995, Taipei, Taiwan*, pages 251–260. IEEE Computer Society, 1995.

[PUR] Persistent URL Home Page, http://www.purl.org/.

[PW00] Thomas A. Phelps and Robert Wilensky. Robust Hyperlinks and Locations. *D-Lib Magazine*, 6(7/8), 2000.

[Ram02] Frank Ramsak. *Towards a general-purpose, multidimensional index: Integration, Optimization and Enhancement of UB-Trees.* PhD thesis, Technische Universität München, 2002.

[RDF] Resource Description Framework (RDF), http://www.w3.org/RDF/.

[RGM00] Sriam Raghavan and Hector Garcia-Molina. Crawling the Hidden Web. Technical report, Stanford University, December 2000.

[RMW95] Martin Röscheisen, Christian Mogensen, and Terry Winograd. Beyond Browsing: Shared Comments, SOAPs, Trails, and On-Line Communities. In *Proceedings of the 3rd World Wide Web Conference*, Darmstadt, Germany, May 1995.

[SB00] Günther Specht and Michael G. Bauer. OMNIS/2: A Multimedia Meta System for Existing Digital Libraries. In *Research and Advanced Technology for Digital Libraries, 4th European Conference, ECDL 2000, Lisbon, Portugal, September 18-20, 2000, Proceedings*, volume 1923 of *Lecture Notes in Computer Science*, pages 180–189. Springer, 2000.

[SB02] Dennis Shasha and Philippe Bonnet. *Database Tuning.* Morgan Kaufmann Publishers, 2002.

[SB03] Günther Specht and Michael G. Bauer. Weiterentwicklung von digitalen Bibliothekssystemen zu OpenArchives-Systemen. In *9. IuK-Tagung 2003*, Osnabrück, 2003.

[Sch03] Ulrike Schwin. XML in der Oracle Datenbank “relational and beyond”. In *BTW 2003, Datenbanksysteme für Business, Technologie und Web, Tagungsband der 10. BTW-Konferenz, 26.-28. Februar 2003, Leipzig*, volume 26 of *LNI*, pages 611–619. GI, 2003.

[Sem] W3C Semantic Web, http://www.w3.org/2001/sw/.

[SK00] Günther Specht and Thomas Kahabka. Information Filtering and Personalization in Databases Using Gaussian Curves. In *2000 International Database Engineering and Applications Symposium, IDEAS 2000, September 18-20, 2000, Yokohoma, Japan, Proccedings*, pages 16–24. IEEE Computer Society, 2000.

[SMB96] Matthew A. Schickler, Murray S. Mazer, and Charles Brooks. Pan-Browser Support for Annotations and Other Meta-Information on the World Wide Web. In *Proceedings of the 5th International World Wide Web Conference*, Paris, France, May 1996.

[SOA] Simple Object Access Protocol (SOAP), http://www.w3.org/TR/SOAP/.

[SSK$^+$01] Jayavel Shanmugasundaram, Eugene J. Shekita, Jerry Kiernan, Rajasekar Krishnamurthy, Stratis Viglas, Jeffrey F. Naughton, and Igor Tatarinov. A General Technique for Querying XML Documents using a Relational Database System. *SIGMOD Record*, 30(3):20–26, 2001.

[STZ$^+$99] Jayavel Shanmugasundaram, Kristin Tufte, Chun Zhang, Gang He, David J. DeWitt, and Jeffrey F. Naughton. Relational Databases for Querying XML Documents: Limitations and Opportunities. In *VLDB’99, Proceedings of 25th International Conference on Very Large Data Bases, September 7-10, 1999, Edinburgh, Scotland, UK*, pages 302–314. Morgan Kaufmann, 1999.

[SWK$^+$01] A. R. Schmidt, F. Waas, M. L. Kersten, D. Florescu, I. Manolescu, M. J. Carey, and R. Busse. The XML Benchmark Project. Technical Report INS-R0103, CWI, Amsterdam, The Netherlands, April 2001.

[Tam] Tamino XML Server, SoftwareAG, http://www.softwareag.com/.

[TVB$^+$02] Igor Tatarinov, Stratis Viglas, Kevin S. Beyer, Jayavel Shanmugasundaram, Eugene J. Shekita, and Chun Zhang. Storing and Querying Ordered XML Using a Relational Database System. In *SIGMOD 2002, Proceedings ACM SIGMOD International Conference on Management of Data, June, 2002, Madison, Wisconsin, USA*. ACM Press, 2002.

[Wei02] Felix Weigel. A Survey of Indexing Techniques for Semistructured Documents, Institute of Computer Science, LMU, Munich. Projektarbeit/project thesis, 2002.

[Wik] Wikipedia, the free encyclopedia, http://www.wikipedia.org/.

[WL98] Ke Wang and Huiqing Li. Discovering typical structures of documents: a road map approach. In *ACM SIGIR Conference on Research and Development in Information Retrieval*, August 1998.

[Xal] Xalan, XSL Stylesheet Processor, http://xml.apache.org.

[Xer] Xerces, XML Parser, http://xml.apache.org.

[XIS] XIS XML database, Excelon, http://www.exceloncorp.com/.

[XMLa] Extensible Markup Language (XML), http://www.w3.org/XML/.

[XMLb] Papers about XML by Ronald Bourret, http://www.rpbourret.com/xml/.

[XMLc] XML Generator, IBM Alphaworks, http://www.alphaworks.ibm.com.

[XPa] XML Path Language (XPath) 1.0, http://www.w3.org/TR/xpath.

[XQu] XML Query (XQuery), http://www.w3.org/XML/Query.

[Zha98] Kaizhong Zhang. Computing similarity between RNA Secondary Structures. In *Proceedings of IEEE International Joint Symposia on Intelligence and Systems, Rockville, Maryland*, pages 126–132, May 1998.