Multidimensional Mapping and Indexing of XML

Michael G. Bauer, Frank Ramsak, Rudolf Bayer
Institut für Informatik, TU München
Boltzmannstr. 3, D-85747 Garching bei München, Germany
{bauermi, ramsak, bayer}@in.tum.de

Abstract: We propose a multidimensional approach to store XML data in relational database systems. In contrast to other efforts we suggest a solution to the problem using established database technology. We present a multidimensional mapping scheme for XML and also thoroughly study the impact of established and commercially available multidimensional index structures (compound B-Trees and UB-Trees) on the performance of the mapping scheme. In addition, we compare our multidimensional mapping to other known mapping schemes. While studying the performance we have identified projection and selection to be fundamental parts of a typical query on XML documents. Our measurements show that projection and selection are orthogonal and require special multidimensional index support to be processed efficiently.

1 Introduction

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 archiving data in XML. With the growing need to deal with large collections of XML documents as well as with the rapid increase in the document sizes there is a strong demand to store and query XML data in databases. This ranges from the development of new, efficient index structures for XML to mapping schemes for XML to relational and object-oriented database systems. As relational database management systems (RDBMS) hold the largest market share there was and still is intensive research going on to efficiently store XML in these systems. Several mapping schemes have been proposed in the literature [FK99, CSF$^+$01] but to our knowledge there has never been an extensive analysis of a mapping scheme in conjunction with index structures.

In our work we discuss a multidimensional approach for indexing XML. We propose a multidimensional mapping scheme for XML to relational DBMS and discuss the performance of UB-Trees and compound B-Trees for the indexing of this mapping scheme.

The rest of the paper is structured as follows. At first we motivate the problem of XML indexing (Section 2) in RDBMS. Then we present a modelling of the XML document in a multidimensional universe (Section 3). We describe an implementation of our multidimensional approach by using Multidimensional Hierarchical Clustering (MHC) and propose a database schema and a technique for query rewriting on this schema (Section 4). We also present detailed performance experiments using the UB-Tree and various compound B-Trees as multidimensional index structures in Section 5. We conclude the paper with a short summary in Section 7.

2 The Problem: Storing XML in RDBMSs

Many approaches have been made to store XML in relational database systems. All approaches use a similar concept though. First the XML document is split into parts of a previously defined granularity. These parts are stored in the RDBMS. Queries on the 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 documents. The projection is either done via methods like XSLT or the projection results are assembled directly from the database. Our approach especially takes care of both cases.

Besides storing XML data, querying large amounts of XML data is 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. Throughout this paper we instead use a form of Pseudo-SQL to present queries on XML documents. We have chosen Pseudo-SQL as it is more similar to SQL dialects available in RDBMS. It is also easier to point out important requirements when querying mapped XML data with SQL. We do not consider Pseudo-SQL to be a full-fledged query language for XML, although it is of course possible to formulate the queries of this paper in XQuery (or XPath) without any loss of semantics. Due to the above restrictions the way vice versa is apparently not true.

We have identified two fundamental parts of a query for XML. Consider the following query in Pseudo-SQL which should retrieve the value of a tag (tag1) in a document containing the tag with value ABC: **select <tag1> from xmlbase where tag2='ABC';**. This query can be (similarly to queries in the relational case) split up into two steps. The selection part identifies the document(s) which contain(s) the pattern matching a predicate s. The projection part in contrast returns only those part(s) of the document(s) which are stated right after the **select** statement as a set of paths. 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 I 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 I that fullfill 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 resulted 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.

3 XML - A Multidimensional Model

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.

Example 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 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 defi ne 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 identifi ers group paths for one document and are the results in the above mentioned selection. We assume that this identifi er 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.

Summing up our results we can defi ne a set of XML documents which are stored in our database as follows.

Definition 3 *Let P be a set of paths, $<_{path}$ the order of the paths, V a set of values and id a document identifier.*

$$\textit{XMLdocs} = \bigcup \{(P, <_{path}, V, id) | id = docid\}$$

For better visualization the three dimensions can be presented as a cube (Figure 1). 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 threedimensional 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 identifi ers which match path expression(s) and value(s).

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

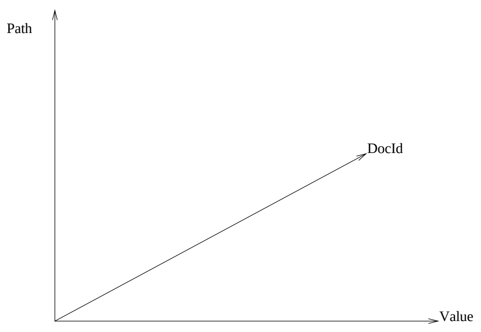

Figure 1: XML documents in a three dimensional cube

4 Implementation

4.1 Multidimensional Hierarchical Clustering

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. 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]. 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. 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 slightly different tree representation as it is usually common. Identical paths which occur several times within a subtree are rewritten by using repetition numbers to preserve uniqueness and order. The paths from example 1 are rewritten to **<author>[1]<FN>[1], <author>[1]<LN>[1],**
<author>[2]<FN>[1], <author>[2]<LN>[1] .

In the following we treat the repetition numbers as an own 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., $ord_m : children(m) \longrightarrow \{1, \ldots, |children(m)|\}$. 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

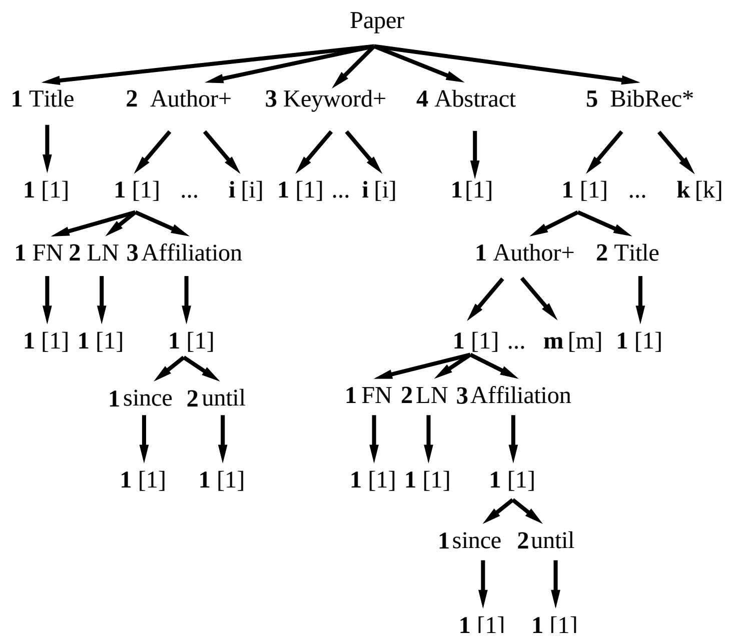

Figure 2: XML document hierarchy and MHC surrogates

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 2 *Using Figure 2 the path* `<author>[1]<FN>[1]` *is transformed into the compound surrogate 2.1.1.1*

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

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

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

Besides the compact, prefix-free and order preserving representation it is also possible to evaluate path expressions 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. It can be shown that all 13 location steps which are defined in XPath 1.0 and XPath 2.0 can be implemented as simple expressions on compound surrogates.

4.2 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 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 2). The table does not contain any information about paths on a per document basis but only stores complete paths to leafs. This reduces the size of **typedim** significantly, so **typedim** is typically very small compared to **xmltriple**.

did	val	surr
1	Rudolf	0010 0001 0001 0001 0000 0000 0000 0000
1	Bayer	0010 0001 0010 0001 0000 0000 0000 0000

Table 1: Relation `xmltriple`

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

Table 2: Relation `typedim`

For our measurements we compared four different indexing methods for our proposed mapping scheme. Since we claim that XML indexing is a multidimensional problem we chose the UB-Tree (threedimensional) and three variants of B-Tree compound indexes for the **xmltriple** relation.

4.3 The Query-Rewriting

As mentioned already in Section 1 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

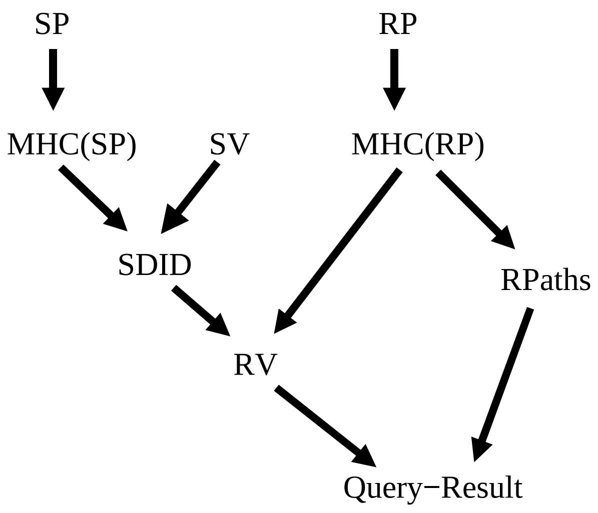

Figure 3: Steps to rewrite the query into SQL

in Figure 3.

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 scientifi c papers that were written by a certain author. The same query will later be used in our measurements.

Example 3
```
select titel, keywords
from xmldata
where /author[FN = "Michael" and LN = "Bauer"]
```

The initial query is seperated 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 3. 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. 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 signifi cant facts about the nature of XML queries.

Example 4 *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
```

5 Measurements

5.1 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 [XML]. 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 not many 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.

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 the threedimensional 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 explained and discussed in detail in Section 5.4.

We rewrote the query from Section 4.3 to SQL as shown above and measured both, the elapsed time and the number of pages that were accessed for answering the SQL queries.

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

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 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 3).

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

Table 3: Number of tuples returned for each processing step

5.2 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 (due to reasons of visibility the fi gure 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.

5.2.1 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-fi lling Z-curve. As the query is processed by repeatingly stabbing through the threedimensional 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 benefi ts 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 and Table 5.

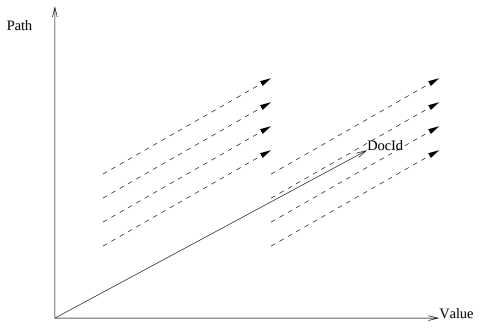

Figure 4: Point restrictions for selection query

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: Page numbers for the selection

5.2.2 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 shows that the query reads almost all pages of the database. The query is slowest among all the others (Table 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.

5.2.3 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).

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 5: Running times for selection and projection queries

5.2.4 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 5 and Table 4) although the performance of the *SurrValDid* index is of course not reached.

5.3 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 5 sketches the scenario with three lines due to the sake of clarity.

5.3.1 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 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 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 being 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 6: Page numbers for the projection

5.3.2 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.

5.3.3 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 fi ltering the retrieved pages. The retrieved 463 data pages carry approx. $463 * 100 = 46300$ tuples. The post fi ltering has to be done for each of the 104 tuples that resulted 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.

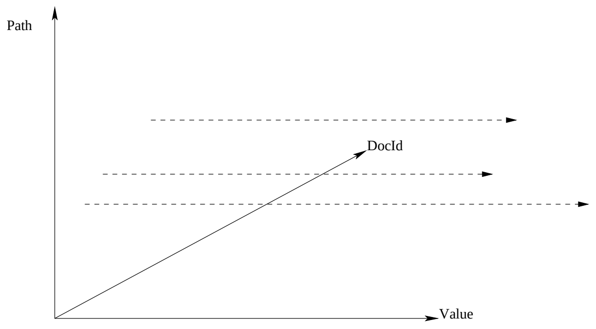

Figure 5: Point restriction for projection query

5.3.4 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.

5.4 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 modifi ed version of the Edge mapping from [CSF$^+$01]. The mapping consists of two tables *roots(id, label)* and *edge (parentid, childid, label)*. The *roots* table contains a tuple for every document with an *id* which identifi es the document (document identifi er) and *label* for the root tag. The *edge* table contains a tuple for every nesting relationship. Non leaf nodes contain the id of the *parent* node in the parent 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.

Example 5 *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.

For our measurements we indexed the root table with a primary compound B-Tree on the attributes (*parentid*, *childid*) (*Edge_Compound* in Tables 4-6). In a second variant we created secondary indexes on each of the attributes *parentid*, *childid*, and *label* (*Edge_Secondary* in Tables 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).

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.

6 Related Work

Storing, indexing and retrieving XML in database systems got a lot of attention in research over the last years. Among the mapping schemes especially the work of Florescu and Kossmann[FK99] was very influential. Their approach to explicitly store the graph of the XML documents in relations (the Edge mapping) became one of the benchmarks that almost all other mappings (including ours) have to compete with. Besides other static mapping approaches (e.g. [STZ+99]) there were efforts to extract the schema from the data itself using data mining algorithms (e.g. as in the STORED project[DFS99]). The schema is then used to store the data in a relational database. Some proposals for storage though neglect the order of paths in XML documents. Order was only recently discussed in detail[TVB+02]. The proposed Dewey order is very similar to our MHC technique and is originally used for the classification of library items.

Most index structures for XML and semistructured data concentrate on the efficient enconding of paths. Cooper, et al.[CSF^{+}01] present a solution based on fast text enconding using patricia tries (which they evolved into a balanced index structure for secondary storage). Besides other research work (T-index [MS99]) there are also commercial systems available which claim to store XML data "natively" but omit a detailed description [Tam, XIS]. Widom, et al. have published significant work on systems storing semistructured data, XML and query languages [MAG^{+}97, AQM^{+}97]. The Lore system which is based on the OEM model uses different indexes on values and paths to speed up query processing [MWA^{+}98]. We use a similar approach for the indexes of our relational XML mapping.

Recently a technique to speed up XPath location steps using a numbering scheme based on pre and post order of the XML document graph was published and benchmarked using R-Trees and compound B-Trees [Gru02]. In contrast to our MHC proposal which is onedimensional this approach is multidimensional.

7 Summary

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-Trees) and compared them to the well-known edge mapping. In the overall running times of the queries we were faster in every case than edge mapping and had a up seven times smaller database size. Additionally we have described the orthogonal nature of typical XML queries.

Due to the special orthogonal requirements for selection and projection currently a combination of the two compound B-Trees seems to be 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 research on the multidimensional model, the orthogonal requirements for selection and projection, and the importance of choice of indexes has shown that typical XML queries carry restrictions on values and structure. Depending on data and queries the performance of indexes may strongly vary with the selectivity of value and structure attributes. This has to be especially taken care of when chosing indexes in database schemas for storing XML.

Acknowledgements

Part of this work is funded by DFG (German Research Foundation) within the research initiative "V3D2" ("Distributed Processing and Delivery of Digital Documents").

References

[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.

[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.

[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.

[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.

[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.

[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.

[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.

[MWA$^+$98] Jason McHugh, Jennifer Widom, Serge Abiteboul, Qingshan Luo, and Anand Rajaraman. Indexing Semistructured Data. Technical report, February 1998.

[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.

[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.

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

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