Efficient storage of XML data [*]

Carl-Christian Kanne
cc@informatik.uni-mannheim.de

Guido Moerkotte
moer@informatik.uni-mannheim.de

Lehrstuhl für praktische Informatik III
Universität Mannheim
Germany

December 13, 1999

Abstract

We introduce NATIX, an efficient, native repository for storing, retrieving and managing tree-structured large objects, preferably XML documents. In contrast to traditional large object (LOB) managers, we do not split at arbitrary byte positions but take the semantics of the underlying tree structure of XML documents into account.

Our parameterizable split algorithm dynamically maintains physical records of size smaller than a page which contain sets of connected tree nodes. This not only improves efficiency by clustering subtrees but also facilitates their compact representation.

Existing approaches to store XML documents either use flat files or map every single tree node onto a separate physical record. The increased flexibility of our approach results in higher efficiency. Performance measurements validate this claim.

1 Introduction

The concept of semistrucured data has gained wide acceptance, especially in the incarnation of XML (*extensible markup language*, [1]). XML is being adopted as an easy-to-write, easy-to-parse language to exchange semistructured data in commercial, financial, medical, scientific and other applications.

With the advent of standardized APIs for semistructured data like the *document object model* (DOM, [2]), the mere exchange of textual representations with generating at one end and parsing at the other will not be sufficient for all applications. Direct access to and manipulation of the documents' tree structure and, as a consequence, efficient storage managers for tree-structured data will be required. We encountered the need for efficient, updateable storage of XML data while designing a data repository for

[*]Part of this work was supported by the German Ministry of Education and Research (BMBF),FKZ 08C58532

an electronic version of the "Biochemical Pathways Atlas" [3], a large collection of chemical reactions annotated with textual descriptions, WWW references, and multimedia data.

The existing approaches to store semistructured data or structured documents can be divided into three categories:

Flat Streams In this approach, trees are serialized into byte streams, for example by means of a markup language. For large streams, some mechanism is used to distribute the byte streams on disk pages. The mechanism can be a file system, or a BLOB manager in a DBMS.

This method is very fast when storing or retrieving whole documents or big continuous parts of documents. Accessing the documents' structure is only possible through parsing [4].

A Web server's HTML file tree, stored in the file system, is a simple example.

Metamodeling A different method is to model and store the documents or data trees using some conventional DBMS and its data model [5, 6, 7, 8].

In this case, the interaction with structured databases in the same DBMS is easy. On the other hand, scanning a whole document or parts of a document, as needed for example when reconstructing a textual representation, is slower as in the previous method; creation of just one typical web page from its abstract syntax tree requires retrieval of maybe thousands of database objects. Other representations requires complex mapping operations to reproduce a textual representation [9], even duplicate elimination may be required [10].

In general, this approach introduces additional layers in the DBMS between the logical data and the physical data storage, slowing down query processing.

Mixed There are several attempts to merge the two "pure" methods above.

Redundant To get the best of both worlds, data is held in two redundant repositories, one flat and one metamodeled [11]. Updates are propagated either way, or only allowed in one of the repositories. This allows for fast retrieval, but leads to slow updates and incurs significant overhead for consistency control.

Hybrid In hybrid approaches, a certain level of detail of the data is configured as "threshold". Structures coarser than this granularity live in a structured part of the database, finer structures are stored in a "flat object" part of the database [12].

NATIX, our XML repository at Mannheim University is similar to the hybrid systems, with two extensions: First, our "flat" parts of the database are not completely flat, but clustered groups of tree nodes treated as atomic records by the lower levels of NATIX. Second, the "threshold" need not be statically configured, but can be a dynamic value, adapting to the size and structure of documents at

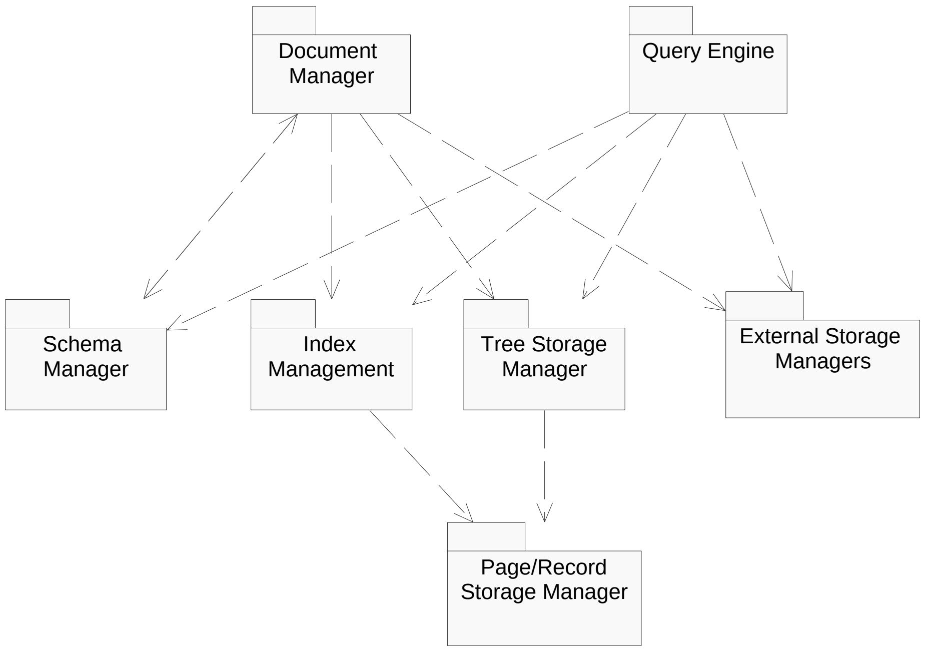

Figure 1: NATIX architectural overview

runtime. As subtrees of the document are changed, clustered nodes can become records of their own or again be merged into clusters. To satisfy special application requirements, clustering of certain node types can be enforced or forbidden by a configuration matrix.

We consider our enhancements to the hybrid systems significant and introduce a new category: We call this kind of storage organization **Native** XML Repository.

The remainder of the paper is organized as follows. Section 2 describes a tree model of the data we want to manage and the physical organization used to store that data. Section 3 describes the methods used to dynamically maintain this physical organization. Section 4 gives some performance results. Section 5 reviews related work and shows how it fits into our model. Section 6 concludes the paper.

2 Preliminaries

Let us first present a a brief NATIX architectural overview, and describe which part of NATIX this paper is about. The logical data model and our model for physical storage organization are also detailed in this section.

```
<SPEECH>
<SPEAKER>OTHELLO</SPEAKER>
<LINE>Let me see your eyes;</LINE>
<LINE>Look in my face.</LINE>
</SPEECH>
```

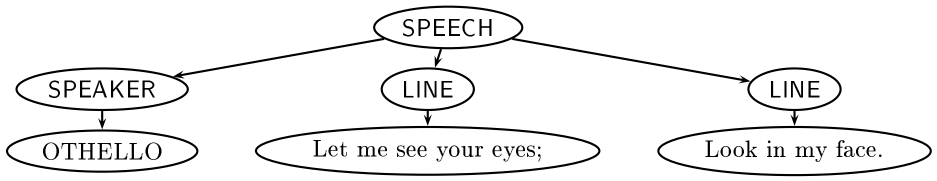

Figure 2: A fragment of XML with its associated logical tree

2.1 Natix Overview

Figure 1 shows the main modules of Natix.

The core of the system is a "classical" physical record manager which is responsible for disk memory management and buffering. It accesses raw disks or file system files and provides a memory space divided into segments, which are a linear collection of equal-sized pages. Pages can be as large as 32K. Each page can be a plain page (for indices and user-defined structures), or holds one or more records. Pages are organized as slotted pages, records are identified by a pair $(pageid, slot)$ (called record ID or *RID*).

On top of the record manager operates a tree storage manager that, maps the trees used to model documents (see section 2.2 below) into records. The methods used in this tree storage manager are the topic of this paper.

Additional modules of Natix exist, but are not detailed here. They include index management, the (not yet implemented) query engine, the schema manager, and the document manager. The document manager allows application access to documents on node and document granularity. It checks schema consistency, called *document validation* in the XML world, performs necessary index updates and integrates document fragments from other sources into a single document view for the user. The schema manager maintains the system catalog data needed by the document manager, which includes the Document Type Definitions (logical XML schema information) and physical schema information and statistics. The system catalog itself is stored as a collection of XML documents inside the system.

2.2 Logical Model

A popular and useful model for XML documents is the labelled tree [2, 13]. General graphs, which are often used to model semistructured data, are represented in XML using special IDREF attributes[1], and XLinks [14], for intra- and inter-document references, respectively.

We use ordered trees in which each non-leaf node is labelled with a symbol taken from an alphabet Σ_{DTD}. Leaf nodes can also be labelled with arbitrarily long strings over a different alphabet (Σ^*). Figure 2 shows an example of an XML fragment and its associated tree. Note that the shown XML document is missing the schema, called *document type definition* (DTD). Details of XML schema descriptions do not concern us here, for our purposes, the DTD is just a way of specifying the node alphabet Σ_{DTD}. Additionally, the DTD can place constraints on how node labels can be combined.

Note that our data model is very similar to an abstract syntax tree and can easily be generated by an XML parser. It also captures all information present in the textual representation of a document, most notably the order of child elements.

2.3 Physical Model

We now elaborate on our physical data model. Besides explaining our own physical tree representation, we hope to provide a general terminology for the description of storage formats for tree-strucured data, which can later be used to compare different approaches (see section 5).

The logical data tree is materialized as a physical data tree, which is built from the original logical nodes and additional nodes needed to manage the physical structure of large trees. Large trees are trees that cannot be stored on a single disk page.

The following sections describe three ways to classify the physical nodes we use to store the logical tree.

Note that, in the following, we use the terms *node* and *object* synonymously. On the other hand, a *record* is something different: It may contain a set of nodes/objects, as explained below.

2.3.1 Object Content

One classification we use for physical nodes is based on their content:

Aggregate nodes are inner nodes of the tree. They contain their respective child nodes.

Literal nodes are leaf nodes containing an uninterpreted stream of bytes, like text strings, graphics, or audio/video sequences.

Proxy nodes are nodes which point to different records. They are used in the representation of large trees, as detailed in section 2.3.3.

2.3.2 Node representation

Instead of storing each tree node in a separate record, we store whole documents (or subtrees of documents) together in one record. This record is treated as atomic by the underlying record manager. Each record contains exactly one subtree. The root nodes of each record's subtree are called *standalone objects*. Other nodes are called *embedded objects*. This is a second way of classifying our physical nodes.

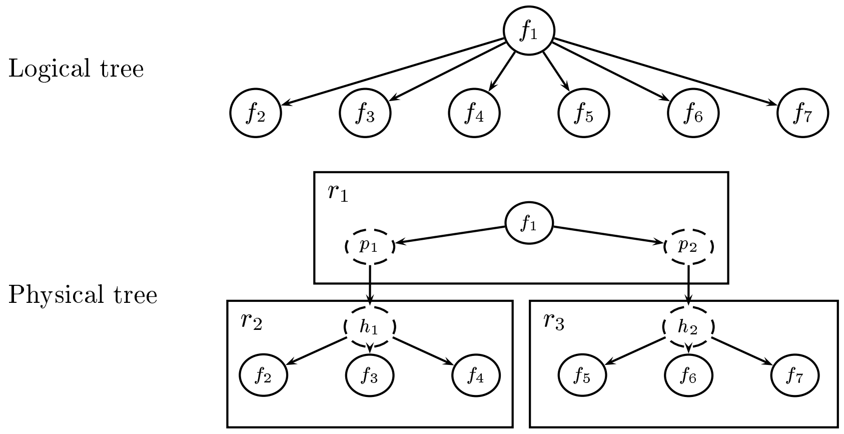

Figure 3: One possibility for distribution of logical nodes on records

The record size has an upper limit, the page size. This raises opportunities to optimize the subtree representation inside the records. Since our algorithm and its presentation do not depend on a specific record representation, we present the low-level details only in Appendix A.

2.3.3 Large Trees

Typical data trees may not fit on a single disk page. So our physical object model must provide a mechanism for distributing data trees over several pages.

One method often used in document management systems is to store a "flat" representation (like the one described in Appendix A) as a BLOB (*binary large object*) and use a mechanism for managing large byte collections inside the storage manager (see [15, 16, 17]). We feel that this approach wastes the available structural information about the data, because treating the representation as a BLOB regards all bytes as equal:

A certain amount of insertions, removals and updates of objects stored in this way would lead to an unfavorable distribution of the data. Some part of even a small object would reside on one page, and the remainder on a different page.

To avoid this, we semantically split large objects based on the underlying tree structure. We partition the data tree into subtrees, and store each subtree in a single record less than a page in size. Connected subtrees residing in other records are represented by *Proxy objects*. Proxy objects consist of the RID of the record which contains the subtree they represent. Substituting all proxies by their respective subtrees reconstructs the original data tree.

A sample is shown in figure 3. To store the given logical tree (which, say, does not fit on a page), the physical data tree is distributed over three records r_1, r_2 and r_3. To achieve this, two proxies (p_1 and p_2) are used in the top level record. Two helper aggregate objects (h_1 and h_2) have also been added to the physical object tree. They are needed to group the children below p_1 and p_2 into records.

This leads to our third classification dimension: Physical objects drawn as dashed ovals like the

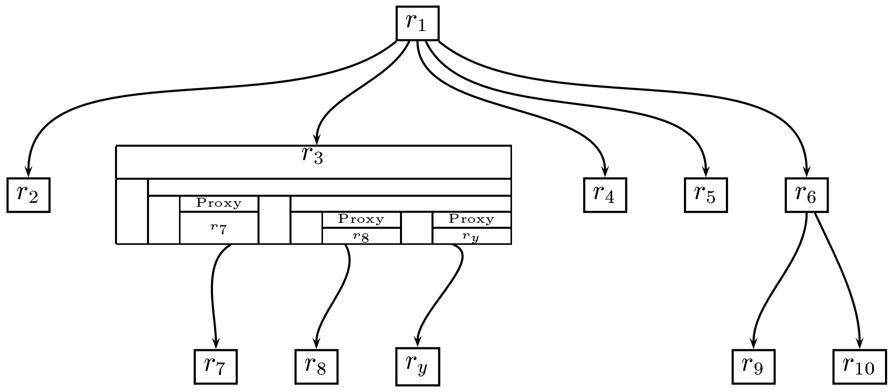

Figure 4: Multiway tree representation of records

proxies p_1, p_2 and the helper aggregates h_1, h_2, needed only for the representation of large data trees, are called *Scaffolding objects*, while objects representing logical nodes (f_i) are called *Facade objects*.

Note that the sample physical tree is only one possibility to store the shown logical tree. There are more, since more of the logical tree's edges could be represented by proxies. The creation and maintenance of such physical trees in our XML repository NATIX, is described in the remainder of this paper.

3 Dynamic maintenance of an efficient storage organization

We will now present the online algorithm used by our NATIX repository for dynamic maintenance of physical trees. The principal problem adressed is that a record containing a subtree can grow larger than a page if a node is added or grows.

In this case, the subtree contained in the record has to be partitioned into several subtrees which can subsequently be distributed on one or more additional records and pages. Scaffolding nodes (proxies and maybe aggregates) have to be introduced into the physical tree to link the new records together.

To describe our tree storage manager and our split algorithm, it is useful to view the partitioned tree as an associative data structure for finding leaf nodes. We will first explain this metaphor, and afterwards use it to detail our algorithm. Possible extensions to the basic algorithm and a flexible configuration mechanism to adapt it to special applications conclude this section.

3.1 Multiway tree representation of records

A data tree that has been distributed over several records can be viewed as a multiway tree with the records as nodes, each record containing a small part of the logical data tree. In the example in figure 4, r_3 is blown up, hinting at the flat representation of the subtree inside record r_3. The references to the child records are proxy objects.

1. Determine the record r into which the node has to be inserted.
2. If there is not enough space on the page, try to move r. If the record still does not fit, split the record:
   (a) Determine the separator by recursively descending into the r's subtree
   (b) Distribute the resulting partitions onto records
   (c) Insert the separator into the parent record, recursively calling this procedure
3. Insert the new node

Figure 5: The Tree Growth Procedure

If viewed this way, our partitioned tree resembles a B-Tree-structure, as often used by traditional large object managers, with the particularity that the "keys" are not taken from a simple domain like integers or strings. Instead, they are based on structural features of the data tree.

This analogy gives us a familiar framework with which we can describe the algorithms used to maintain the clustering of our records.

3.2 Algorithm for Tree Growth

Figure 5 shows the basic structure of our algorithm for adding nodes to a tree. The following subsections will explain the steps in detail.

3.2.1 Determining the Insertion Location

In order to insert a new node f_n into the logical data tree as a child node of f_1, it must be decided where in the physical tree the insert should take place. In the presence of scaffolding nodes, there may exist several possibilities, as shown by the dashed lines in figure 6 (the nodes drawn as dashed ovals are scaffolding nodes); the new node f_n can be inserted into r_a, r_b, or r_c. In our system, this choice may be

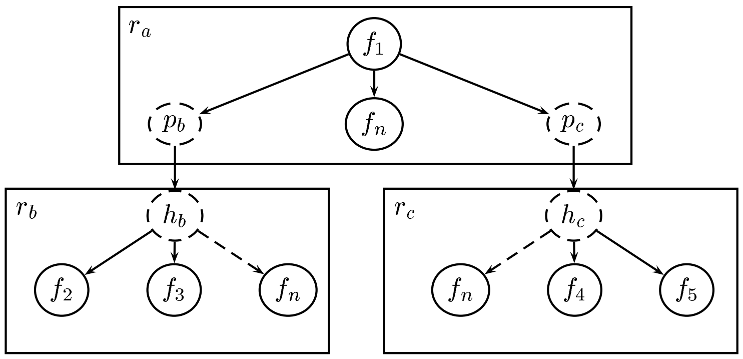

Figure 6: Possibilities to insert a new node f_n into the physical tree

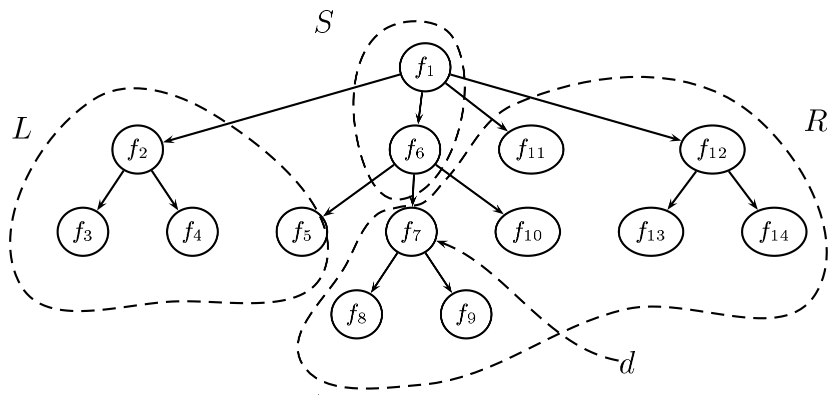

Figure 7: A record's subtree before a split occurs

determined by a configuration parameter, as explained in section 3.3.

3.2.2 Splitting a record

Having decided on an insertion location, it is possible that the designated record's disk page is full. In this case, the system tries to move the record to a page with more free space. If this is not possible, because the record as such exceeds the net page capacity, the record has to be split.

Determining the separator Suppose that, in figure 6 we want to add f_n to record r_b, which cannot grow. Hence, r_b must be split into at least two records r_b' and r_b'', and instead of p_b in the parent record r_a, we need a *separator* with proxies pointing to the new records to indicate where which part of the old record was moved.

In B-Trees, a median key that partitions the data elements into two subsets is chosen as separator. In our tree storage manager, the data in the records is not one-dimensional, but tree-structured. It follows that our separator has to be tree-structured as well.

In fact our algorithm slices a small subtree off the old record's root. This small subtree then servers as as separator. The remaining forest of subtrees is the data that has to be distributed onto the new records.

Figure 7 shows the subtree of one record just before a split. It is partitioned into a left partition L and a right partition R, and the separator S. This separator will be moved up to the parent record, where it indicates into which records the descendant nodes were moved as a result of the split operation.

Already a single node d uniquely determines this partitioning (in the example, $d = f_7$): The separator S consists of all the nodes on the path from d to the subtree's root (in the example, $S = \{f_1, f_6\}$), excluding d. The subtree below d, the subtrees of d's right siblings, and all subtrees below nodes that are right siblings of nodes in S comprise the right partition (in the example, $R = \{f_7, f_8, \ldots, f_{14}\}$), the remaining nodes comprise the left partition (in the example, $L = f_2, \ldots, f_5$).

Hence, our split algorithm must find a node d, such that the resulting L and R are of equal size.

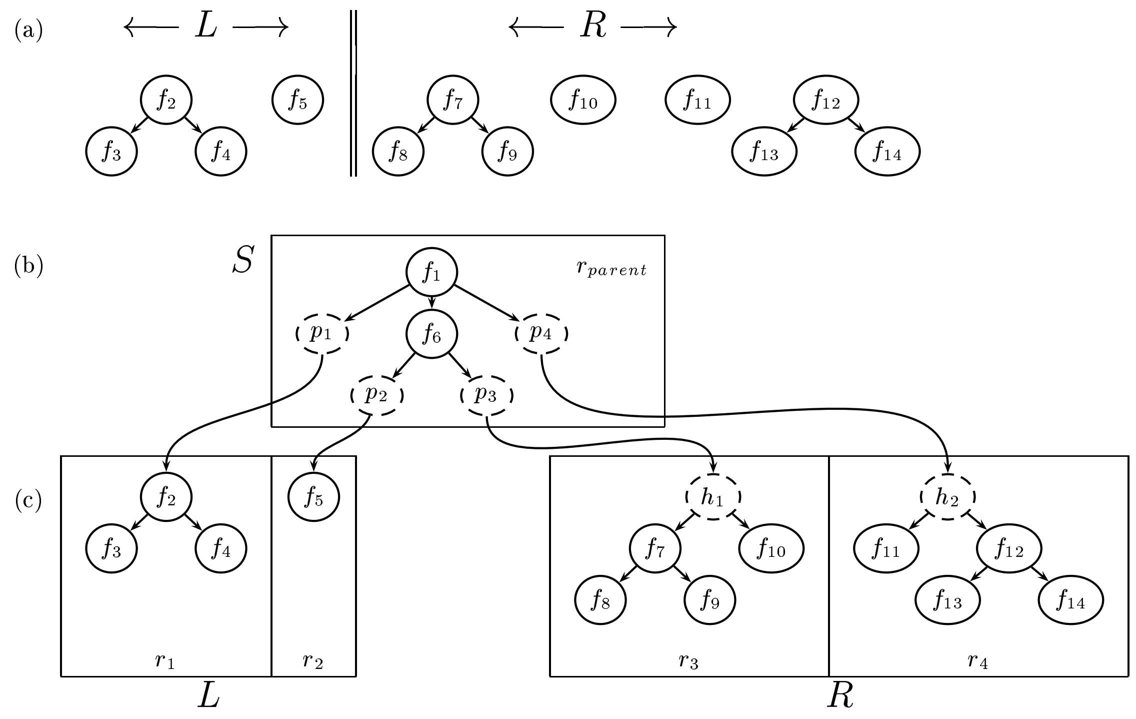

Figure 8: Record assembly for the subtree from figure 7

Actually, the desired ratio between the sizes of L and R is a configuration parameter (the *split target*), which can, for example, be set to achieve very small R partitions to prevent degeneration of the tree if insertion is mainly on the right side (as when creating a tree in pre-order from left to right). Another configuration parameter available for fine-tuning is the *split tolerance*, which states how much the algorithm may deviate from this ratio. Essentially, the split tolerance specifies a minimum size for the subtree of d. Subtrees smaller than this value are not split, but completely moved into one partition to prevent fragmentation.

To determine d, the algorithm starts at the subtree's root and recursively descends into the child whose subtree contains the physical "middle" (or the configured split target) of the record. It stops when it reaches a leaf, or when the subtree size in which it is about to descend is smaller than allowed by the split tolerance parameter.

In the example in figure 7, the size of the subtree below f_7 was smaller than the split tolerance, otherwise, the algorithm had descended further and made f_7 part of the separator.

Distributing the nodes on records After determining the partitioning, the contents of the record has to be distributed onto new records.

Consider a partitioning as implied by node $d = f_7$ in figure 7. The separator is removed from the old record's subtree, as in figure 8(a). In the resulting forest of subtrees, root nodes in the same partition that were siblings in the original tree are grouped under one scaffolding aggregate. In figure 8(c), this happened at nodes h_1 and h_2. Each resulting subtree is then stored in its own record. These new records $(r_1, \ldots, r_4)$ are called *partition records*.

Inserting the separator The separator is moved to the parent record and inserted instead of the proxy which referred to the old, unsplit record, figure 8(b). The edges connected to the nodes in the partition records are replaced by proxies p_i. Since children with the same parent are grouped in one scaffolding aggregate, for each level of the separator a maximum of three nodes is needed, one proxy for the left partition record, one proxy for the right partition record, and one separator node.

To avoid unnecessary scaffolding records, the algorithm considers two special cases: First, if a partition record would consist of just one proxy, the record is not created and the proxy is inserted directly into the separator. Second, if the root node of the separator is a scaffolding aggregate, it is disregarded, and the children of the separator root are inserted in the parent record instead.

To ensure that the parent record contains enough space to hold the separator, the insertion procedure is recursively called for the parent record using the separator as the node to be inserted. If the old record had no parent record, a new root record for the tree is created which contains just the separator.

3.2.3 Inserting the New Node

Finally, the new node is inserted into its designated partition record.

The splitting process operates as if the new node had already been inserted into the old record's subtree, for two reasons. First, this ensures enough free space on the disk page of the new node's record. Second, it also results in a preferable partitioning since it takes the space needed by the new node into account when determining the separator.

3.3 The Split Matrix

When designing the storage manager for the biochemistry database mentioned in the introduction, it quickly became evident that it is not always desirable to leave full control over data distribution to the algorithm. Special application requirements had to be considered. In general, it should be possible to benefit from knowledge about the application's access patterns.

If parent-child navigation from one type of node to another type is frequent in an application, we want to prevent the split algorithm from storing them in separate records. In other contexts, we want certain kinds of subtrees always to be stored in a separate record, for example to collect some kinds of information in their own physical database area.

To express preferences regarding the clustering of a node type and its parent node type, we introduce a *Split Matrix* as parameter to our algorithm:

The Split Matrix S consists of elements $s_{ij}, i, j \in \Sigma_{\mathrm{DTD}}$. The elements express the desired clustering behaviour of a node x with label j as children of a node y with label i:

$$s_{ij} = \begin{cases} 0 & x \text{ is always kept as a standalone} \\ & \text{ record and never clustered with } y \\ \infty & x \text{ is kept as long as possible} \\ & \text{in the same record with } y \\ \textbf{other} & \text{the algorithm may decide} \end{cases}$$

The algorithm as described in section 3.2 acts as if all elements of the Split Matrix were set to the value **other**. It is easily modified to respect the Split Matrix:

When moving the separator to the parent, all nodes x with label j under a parent y with label i are considered part of the separator if $s_{ij} = \infty$, and thus moved to the parent. If $s_{ij} = 0$, such nodes x are always created as standalone object and a proxy is inserted into y. In this case, x is never moved into its parent as part of the separator, and treated for splitting purposes like the root record.

We also use the Split Matrix as the configuration parameter for determining the insertion location of a new node (see section 3.2.1): When a new node x (label j) shall be inserted as a child of node y (label i), then if $s_{ij} = \infty$, x is inserted into the same record y. If $s_{ij} = \textbf{other}$, then the node is inserted on the same record as one of its designated siblings (wherever there is more free space). If $s_{ij} = 0$, x is stored as a standalone node and treated as described above.

In the future, we plan to enrich the semantics of the Split Matrix to support even more adaptable algorithms. For example, the **other** values could contain traversal frequencies gathered from profile traces.

It should be noted that, the Split Matrix is an optional tuning parameter: It is not needed to store XML documents, it only provides a way to make certain access patterns of the apllication known to the storage manager. The "default" split matrix used when nothing else has been specified is the one with all entries set to the value **other**.

As a side effect, other approaches to store XML and semistructured data can be viewed as instances of our algorithm with a certain form of the Split Matrix, as described in section 5.

4 Performance Results

In this section we will present some results concerning the storage and retrieval of a large document collection with NATIX' tree storage manager.

4.1 Environment and test data

The implementation of the record and tree storage managers was done in C++. Measurements were taken on a Pentium-II 333Mhz machine with 128MB RAM under Windows NT 4.0, using an IBM DCAS 34330W disk. Direct disk access and no operating system buffering was used by the record manager. As document collection served an XML markup version of Shakespeare's plays [18]. The total size of the documents is about 8 MB, their tree representations contain about 320000 nodes total.

4.2 Configuration

We compare two configurations of our system: First, we configure the Split Matrix with all elements set to $s_{ij} = 0$. This emulates the approach of storing each tree node in a separate record together with a list of child references. In this case, records are never split, unless the list of children does not fit onto a single page. The record manager was told to store parent with children and sibling nodes on the same page if possible. In the following sections, this configuration is called the *1:1* configuration.

Second, we set all elements of the Split Matrix s_{ij} to the value **other**, giving our algorithm full control over the distribution of the nodes on records. In the following sections, this configuration is called the *1:n*, or *native XML* configuration.

Note that, a "one record" configuration with all matrix elements set to ∞ does not work because we could not store any document larger than a page.

In both cases, the split target was set to $\frac{1}{2}$ to produce two partitions of equal size in each split. The split tolerance was set to $\frac{1}{10}th$ of a page. The buffer size was 2 MB, enough to hold at least one document in any representation. The page size was varied between 2K and 32K. The buffer was cleared at the start of each operation.

4.3 Operations Measured

For storage, we used an XML parser written in C and inserted the document tree in two different insertion orders. First, in pre-order, to represent a "bulkload" of or consecutive *append*s to a textual representation. Second, we traversed the binary tree representation of the document tree (in which each node has its first child as left binary child and next sibling as right binary child [19]) with breadth-first search to insert the nodes, resulting in an *incremental update* pattern where inserts occur distributed over the whole document.

Four kinds of retrieval operations were performed: A full pre-order tree traversal, and three pattern matching queries. The first query retrieves all speakers in the third act and second scene of every play, which means it accesses all leaf nodes of a certain type in one selected subtree of the document. The second query recreates the textual representation of the complete first speech in every scene, hence

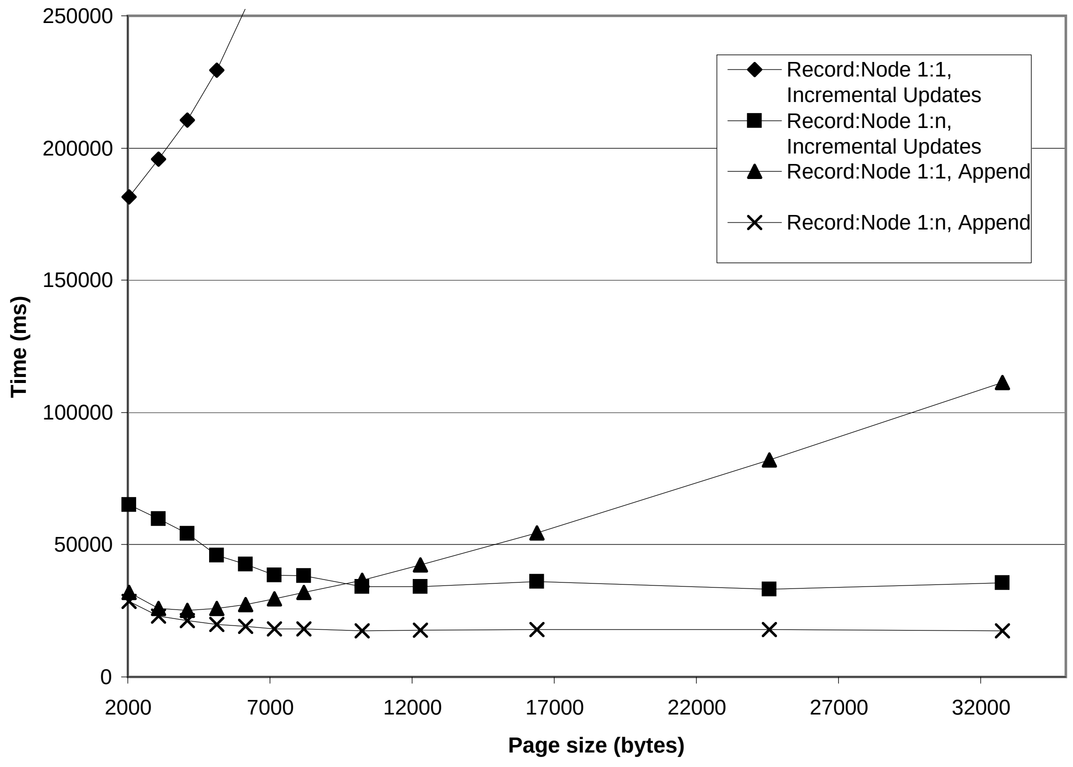

Figure 9: Insertion

reading a lot of small continguous fragments of each document. Last, a simple path query was evaluated by reading only the opening speech of each play.

4.4 Results

Figures 9 to 14 show the results. The operation times result form averaged series of measurements and are given in milliseconds, and are shown as a function of the page size; it can be seen that page size has a significant influence on performance.

4.4.1 Update

Update is faster when the algorithm has full control over the distribution of nodes on records. If updates are scattered all over the data, the difference can be almost an order of magnitude. This is probably a result of the very localized access pattern when nodes are grouped into records, both in terms of main and secondary memory.

Interestingly, our approach benefits from larger page sizes, while the ”traditional” approach of storing each node in a separate record performs best when using ”traditional” small 2-4K pages.

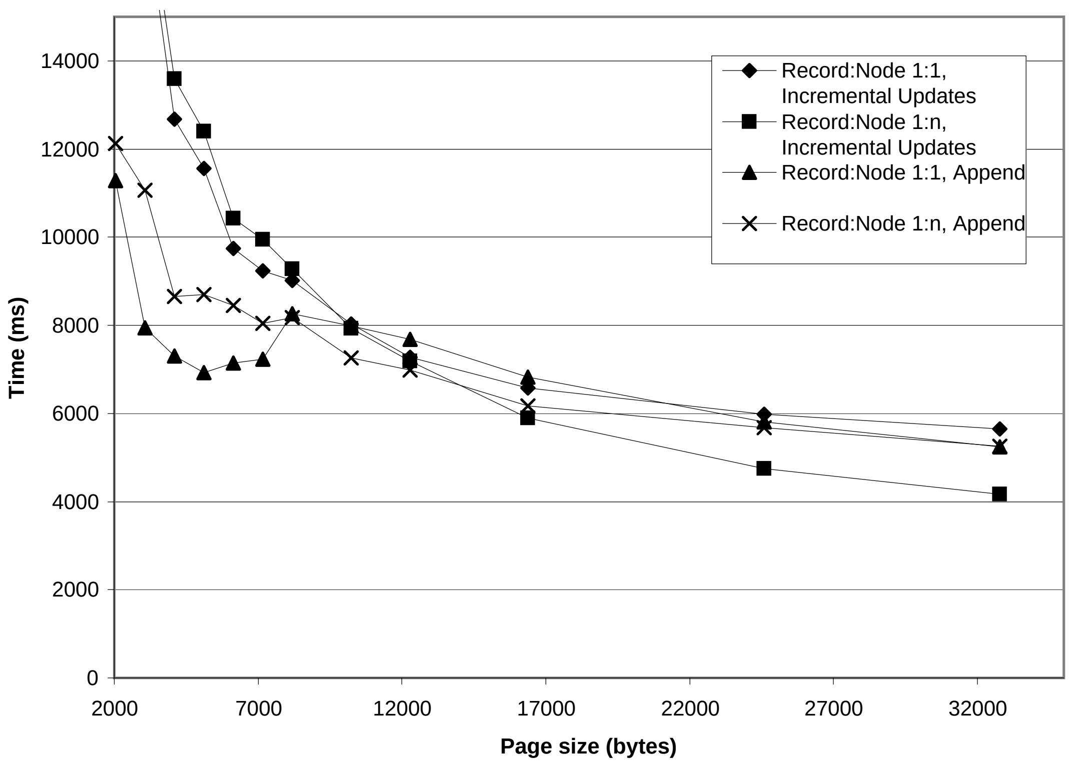

Figure 10: Full tree traversal

Note that, the best result (4K pages) of the "1:1"-approach is more than 50% slower than the best result (32K pages) of the native format when using pre-order insert.

To permit better resolution for the faster runs, the update times for incremental updates and 1:1 format are only shown for pages up to 6K in size. After that, they increase almost linearly to above $1000000ms$ for 32K pages. Hence, incremental updates are faster by at least a factor of three when using the native format.

4.4.2 Full tree traversal

When traversing the full tree in preorder, again the best result is achieved by using the native format with a large page size. It is faster by 20% than the best result for the single record approach. In this case, large page sizes are good for everyone, because all data has to be read anyway, and loading large continguous chunks is faster than loading small ones.

4.4.3 Query 1

When retrieving leaf nodes of a certain type in a selected subtree of each document, the best behaviour is again shown by the native format after incremental updates. Incremental updates do not produce a good clustering for the single-record approach. The resulting random access pattern slows query evaluation

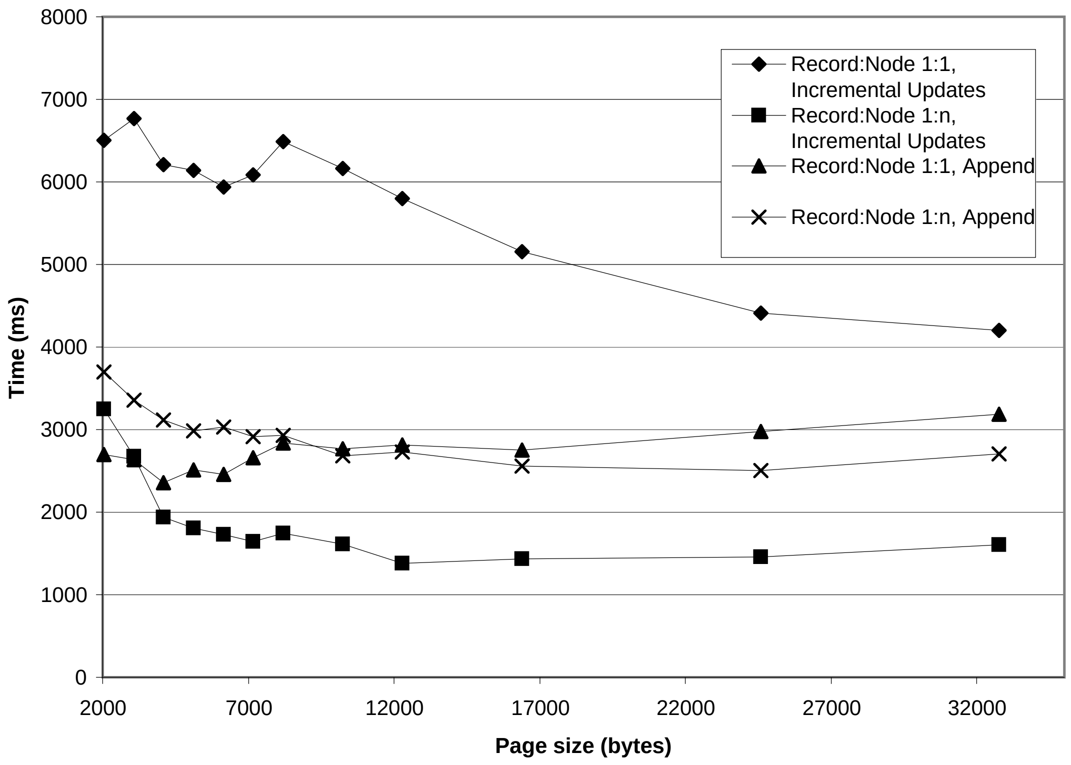

Figure 11: Selection on leaf nodes of document subtree (Query1)

after incremental updates.

The native format after pre-order insert also does not perform well, because the physical tree is linearly degenerated. To reach the specified nodes, nearly the whole document data is loaded into memory.

4.4.4 Query 2

In this query, small contiguous fragments of the documents are retrieved. Hence, small pages result in better access times for all storage formats. In the native format after incremental updates, the nodes belonging to each fragment usually are clustered within the same record, so query time is halved compared to the other formats.

4.4.5 Query 3

In this query, again the balanced tree after incremental updates on the native format results in the best performance. The physical record tree has only a depth of 2, so just two disk accesses are needed to reach each requested speech.

The degenerated tree after pre-order insert again leads to unnecessary reads for the native format.

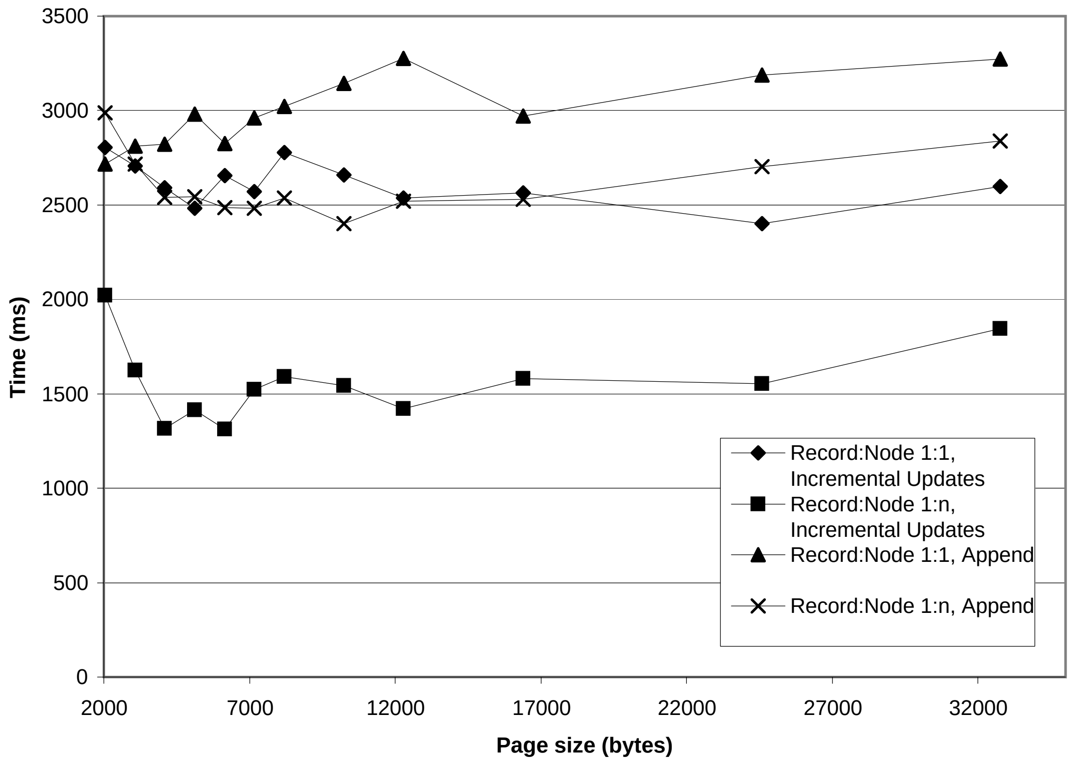

Figure 12: Small continguous fragments (Query2)

4.4.6 Space Requirements

The space requirements given is the total number of bytes on disk used to store the documents.

As expected, the native format has a much better space utilization due to the compact subtree representation inside the records (Appendix A). The single records for each node carry a lot of overhead, most notably big parent and child references, and slot information for each record.

The space utilization is better for larger pages, since less per-page administration space is needed. For the native format, less splits and proxies are needed for large pages, which further reduces the space overhead.

The reduced space requirements give an additional opportunity for faster query processing: If the logical structure and order of the nodes is irrelevant for a query (e.g. scan all elements of a given type), a simple scan of the documents requires time proportional to the amount of I/O required, which is significantly less if the native format is used.

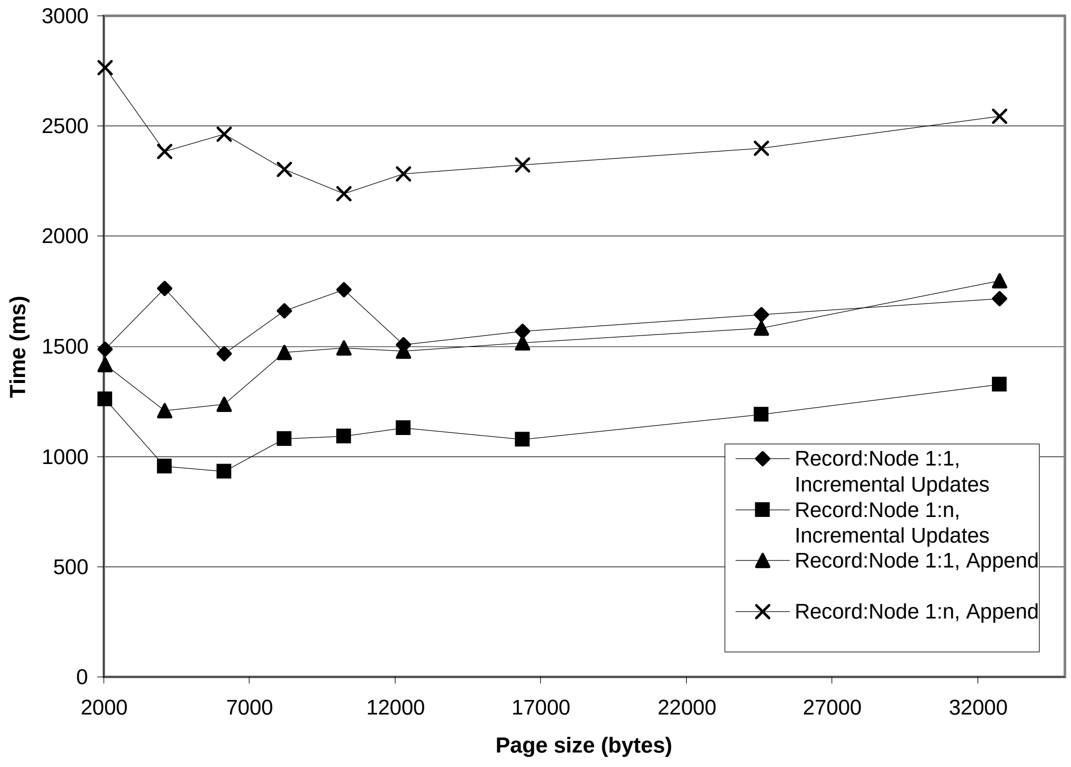

Figure 13: Single path for each document (Query3)

5 Related Work

Other work on efficient storage for (binary) large objects [15, 16, 17] also uses trees to organize the physical distribution of the data, but does not exploit the internal semantic structure of large objects. Objects are split at arbitrary byte positions.

Not much work on efficient storage organization for semistructured data currently exists. There are other proposed repositories for semistructured data, not focussing on storage organization, as detailed in the following.

Flat files are studied closely by Abiteboul et al. [4]. There, a parser is used to gain access to the document structure and evaluate queries.

Metamodeled systems in the categorization of section 1 are already commercially available. POET (POET Content Management system [6]) and ObjectDesign (Excelon [8]) each use their object-oriented database systems (POET Server and ObjectStore, respectively) to store and model SGML/XML documents as trees. They use a separate record for each tree node, which in the terminology of section 2 means that each facade node is a standalone node, and all aggregates contain exclusively proxies. This is equivalent to a configuration of our algorithm with all elements of the Split Matrix set to 0 as evaluated

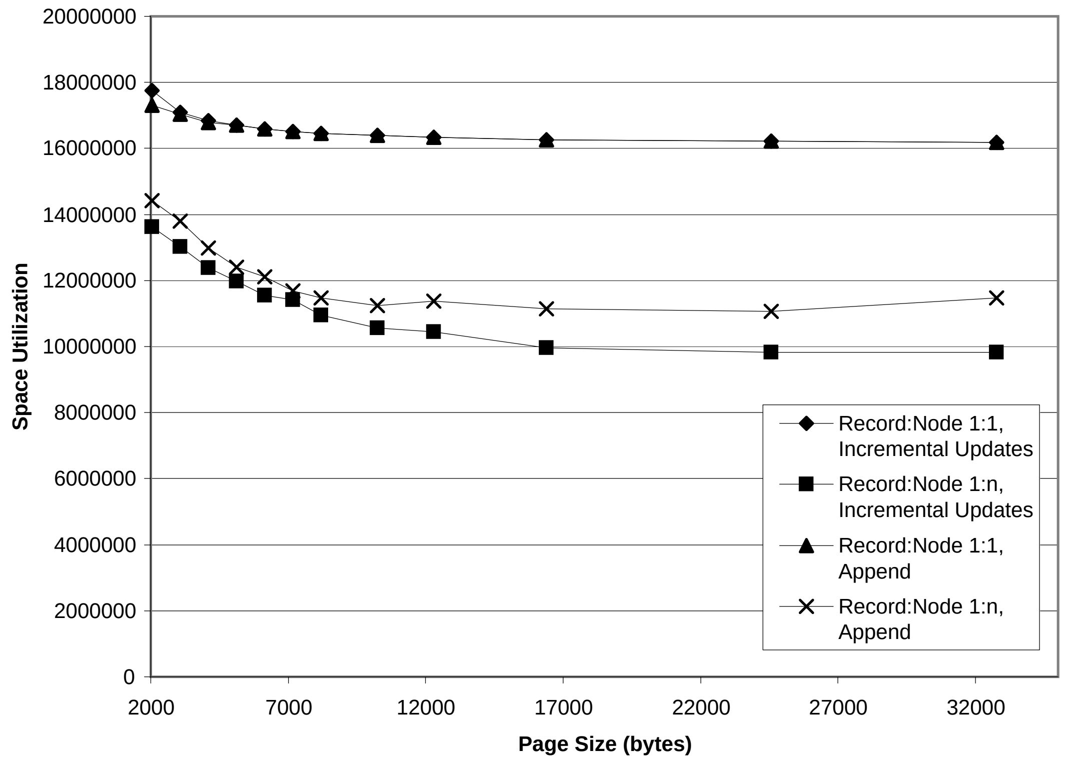

Figure 14: Space requirements

in section 4.

The Lightweight Object REpository (LORE, [5]) uses a graph model as well. Like POET and Excelon, it stores each node in its own record, although the system is not developed on top an existing OODBMS. Their focus is on query processing.

A different metamodeling approach to store XML is STORED [10]. For a class of documents, a relational schema is automatically created and the documents are stored in any existing RDBMS. This automatic schema creation is a complex operation, needed every time a new document type is encountered. Storing XML documents requires pattern matching to map the graph data to relational data, another complex operation, needed every time a document is stored. Incremental updates to the XML tree, and the ordering of elements inside a document are not considered. Retrieving documents requires duplicate elimination, since a single XML node can be stored in more than one tuple. To enable lossless storage of documents that do not fit into the schema, an 'overflow store' is needed. The functionality required from this overflow store is the same as for a full-fledged XML repository.

Another relational approach is taken in the Monet database[7], where XML data is decomposed into binary relations. These relations, one for each tag name, contain the edges of the tree representation. This is similar to the approach of a separate record per node, although efficient main-memory representations for small relations are used. As in STORED, incremental updates to the XML tree, and the ordering of elements inside a document are not considered. The latter is of importance when trying to recreate a textual document representation.

The hybrid HyperStorM system by Neuhold et al. [12] bears the most similarity to our approach. In HyperStorM, the upper levels of the document tree are stored as standalone nodes with proxies as in POET, Excelon and LORE. Certain node types are statically configured to be "flat", which in our terminology means they contain only embedded nodes. Embedded nodes are stored as markup strings and have to be parsed to access the structure. Proxies do not exist, which means that, in the subtree below an embedded node, only more embedded nodes can exist. This is equivalent to our algorithm with a Split Matrix which contains only 0 and ∞ elements and no elements of value **other**. The configuration of HyperStorM is static and done one the type level, while our system dynamically makes splitting decisions and allows configuration not only based on node type, but based on combinations of node type and parent node type. They do not address the problem of splitting large "flat" objects, but leave this to the underlying storage manager, an OODBMS.

6 Conclusion and Future Work

We have presented a method to dynamically maintain efficient physical storage for large tree-structured objects. A flexible model to describe physical storage formats for trees was used to describe our algorithm and related approaches.

In contrast to traditional large object managers or file systems, our storage manager NATIX uses the semantic structure of large objects to make better splitting decisions. Our splitting algorithm is configurable to a degree that allows to simulate other storage formats already in use for tree-structured data. First measurements performed with XML data show the strengths of our approach. Updates and queries can be sped up by a factor of two or more. Space utilization is also better, by a factor of nearly two compared with other approaches.

In the future, besides studying and extending the effect of configuration parameters on the splitting algorithm, query processing operators and index structures that support our storage structure will play a dominant role in our research.

References

[1] Extensible Markup Language (XML) 1.0, W3C Recommendation, February 1998. available at `http://www.w3.org/TR/1998/REC-xml-19980210`.

[2] Lauren Wood et al. Document Object Model (DOM) level 1 specification, version 1.0, October 1998. W3C Recommendation, available at `http://www.w3.org/TR/1998/REC-DOM-Level-1-19981001`.

[3] Gerhard Michal, editor. *Biochemical Pathways*. Spektrum Akademischer Verlag, Heidelberg, 1999.

[4] Serge Abiteboul, Sophie Cluet, and Tova Milo. Querying and updating the file. In *Proceedings of the Conference on Very Large Data Bases (VLDB)*, pages 73–84, 1993.

[5] 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, September 1997.

[6] POET Content Management Suite. `http://www.poet.com/CMSsdk.htm`.

[7] Roelof van Zwol, Peter M. G. Apers, and Annita N. Wilschut. Modeling and querying semistructured data with moa. In *ICDT'99 Workshop on Query Processing for semistructured data*, 1999.

[8] ObjectDesign Inc. Excelon, the ebusiness information server. `http://www.odi.com/excelon`.

[9] J. Shanmugasundaram, H. Gang, K. Tufte, C. Yhang, D. J. DeWitt, and J. Naughton. Relational databases for querying xml documents: Limitations and opportunities. In *Proceedings of the Conference on Very Large Data Bases (VLDB)*, pages 302–314, 1999.

[10] Alin Deutsch, Mary F. Fernandez, and Dan Suciu. Storing semistructured data with STORED. In *Proceedings of the 1999 SIGMOD Conference*, 1999.

[11] Tak W. Yan and Jurgen Annevelink. Integrating a structured-text retrieval system with an object-oriented database system. In *Proceedings of the Conference on Very Large Data Bases (VLDB)*, pages 740–749, 1994.

[12] Klemens Böhm, Karl Aberer, Erich J. Neuhold, and Xiaoya Yang. Structured document storage and refined declarative and navigational access mechanisms in HyperStorM. *VLDB Journal*, 6(4):296–311, 1997.

[13] Extensible Stylesheet Language, W3C Working Draft, April 199. available at `http://www.w3.org/TR/WD-xsl`.

[14] XML Link Language, W3C Working Draft, March 1998. available at `http://www.w3.org/TR/WD-xlink`.

[15] Alexandros Biliris. An efficient database storage structure for large dynamic objects. In *Proceedings of the International Conference on Data Engineering*, pages 301–308, 1992.

[16] Michael J. Carey, David J. DeWitt, Joel E. Richardson, and Eugene J. Shekita. Object and file management in the EXODUS extensible database system. In *Proceedings of the Conference on Very Large Data Bases (VLDB)*, pages 91–100, 1986.

[17] Tobin J. Lehman and Bruce G. Lindsay. The starburst long field manager. In *Proceedings of the Conference on Very Large Data Bases (VLDB)*, pages 375–383, 1989.

[18] UNC Sunsite. XML Sample Documents. available at `http://metalab.unc.edu/pub/sun-info/standards/xml/eg/`.

[19] Donald E. Knuth. *The Art of Computer Programming*, volume 1, chapter 2.3.2. Addison Wesley, second edition, 1998.

Byte Repr.	10 Bytes	6 Bytes	6 Bytes	OTHELLO	6 Bytes	6 Bytes	Let me. . .	6 Bytes	6 Bytes	Look in. . .
	↑	↑	↑	↑	↑	↑	↑	↑	↑	↑
Structure	Header	Header	Header	Contents	Header	Header	Contents	Header	Header	Contents
			String literal node			String literal node			String literal node	
			Contents			Contents			Contents	
		SPEAKER node			LINE node			LINE node		
		Contents								
	SPEECH node									

Figure 15: The tree from figure 2 represented as one record

A Record representation

This section explains the format used to store subtrees in flat records. As a result of the limited page size, we can materialize the parent-child-relationships rather efficiently, saving references pointing to nodes outside the record.

Inside each record, the nodes are stored *within* their respective parent aggregate objects, so the complete subtree rooted at any given object consists of all the objects recursively contained in it. An example for the tree from figure 2 is shown in figure 15.

Each record contains exactly one root node which contains all the other objects in the record (the SPEECH node in the example above). As explained in section 2.3.2, such objects are called *standalone objects*, while objects stored within other objects are called *embedded objects*.

The physical representation of objects on disk starts with a header describing the *content type* (aggregate, literal or proxy) and the *logical type* (e.g. the tag or attribute name for *Facade objects*), the *size of the object* and a *parent pointer*. Literals are typed, currently either string literals, 8/16/32/64-Bit integer literals, float, or URI (Uniform Resource Identifier) literals.

Since on each page typically only a limited set of (content type, logical type) combinations occur, this information is stored in the object header as 2 byte offset into a node type table which is maintained on each page.

Pointers needed to materialize the relationships for the embedded nodes within one record only need 2 bytes, since a page is less than 64K in size. Since the embedded parent pointers are stored as offsets, the byte representation of subtrees in records is location-independent, so that records can be moved around on the page without modification.

Our layout results in a header of only 6 bytes for embedded objects, minimizing the overhead for storing the tree structure. Note that, for example, storing vanilla XML markup with only a 1-character tag name already needs 7 bytes (< `X` > . . . < /`X` >)!

Standalone objects contain their parent record as RID (8 bytes). The size of the object, which is equal to the size of the record, can be obtained from the slot information. Together with the type index, a standalone header usually consumes 10 bytes.