The Importance of Sibling Clustering for Efficient Bulkload of XML Document Trees

Carl-Christian Kanne
Guido Moerkotte
cc|moer@pi3.informatik.uni-mannheim.de

July 15, 2005

Abstract

In an XML Data Store (XDS), importing documents from external sources is a very frequent operation. Since a document import consists of a large number of individual node inserts, it is essentially a small bulkload operation. Hence, efficient bulkload support is crucial for XDSs.

Essentially, XML bulkload is the transformation of an XML parser's output into the XDS's persistent storage structures. This involves two major subtasks: (1) Partitioning the documents' logical tree structure into subtrees smaller than a disk page in a way that is both space-efficient an suitable for later processing. (2) Mapping the subtrees to the XDS's internal page representation. In enterprise-scale environments with very large documents and/or very many parallel bulkloads, task (1) is particularly challenging, as not only disk space consumption, but also CPU and main-memory usage are important factors.

In this article, we (1) discuss requirements for an XML bulkload module, (2) examine existing algorithms for tree partitioning with respect to their applicability as XML bulkload algorithms, (3) derive a new tree partitioning algorithm, (4) present the design and implementation of the bulkload module used in our Natix XDS, and (5) evaluate the implementation.

1 Introduction

Loading of large amounts of data which is already available in an external format is called a *bulkload* operation. In conventional DBMS, bulkloads are often used to initialize a database, for example when introducing an application to DBMS usage, or when importing data from a different DBMS or storage format.

In contrast, an XDS needs to support document imports as a regular operation which is used very frequently by applications. Hence, bulkload becomes a core functionality whose performance is a determinant of overall system performance. However, we could find only very few publications that discuss efficient XML bulkload in large-scale XML data stores.

This article attempts to mitigate this deficit by presenting the design and implementation of the bulkload component for the Natix XDS, a native XML data store developed at the University of Mannheim [7]. Besides requirements analysis and interface design, our main focus is the design of an efficient tree partitioning algorithm that decomposes the logical XML document tree into subtrees, or *clusters*, that fit on a disk page. Such a clustering algorithm is needed not only in Natix, but in every XDS that provides native tree storage, such as IBM's System RX [2].

Of particular concern is the number of clusters generated. When accessing the stored documents, inter-cluster navigation is much slower than intra-cluster navigation, often by several orders of magnitude. Even if access reordering techniques are used [8], the number

of clusters is a crucial factor for query performance. Hence, a bulkload algorithm must minimize the number of generated clusters.

Our main contributions are as follows:

1. A detailed requirements analysis for XML bulkload components.
2. An analysis of existing tree clustering algorithms.
3. A novel linear time tree clustering algorithm that generates up to 30% less clusters than the best known algorithms.
4. A description of the design and implementation of a concrete XML bulkloader, the Natix Bulkload Component.
5. An evaluation of the Natix Bulkload Component.

The outline of the article is as follows. In Sec. 2, we give a brief overview of native tree storage in Natix. In Sec. 3, we analyze the requirements an XML bulkload component must meet. In Sec. 4, we discuss existing tree clustering algorithms with respect to these requirements. Sec. 5 describes the interface and implementation of the Natix Bulkload Component. This section also covers our novel tree clustering algorithm. Sec. 6 contains experimental results, including a comparison with the bulkload performance of other XML storage systems. Sec. 7 concludes the article.

2 XML Storage

The requirements for bulkload components are strongly influenced by characteristics of the desired target format.

A suitable format for an enterprise-level XDS must efficiently support bulkloads, incremental updates, synchronization and recovery. In this section, we briefly review the Natix incarnation of such a format (for details, refer to [7, 9, 10]). Similar formats are also used in other XDSs, such as IBM DB2 (System RX) [2].

2.1 Logical Document Object Model

The individual documents are represented as ordered trees in which nodes are labeled with a symbol taken from an alphabet Σ_{Tags}. In the current implementation, we use the set of integers $0 \ldots 2^{16} - 1$ as Σ_{Tags}. Leaf nodes can, additionally, be labeled with arbitrarily long strings over a different alphabet. In the current implementation, leaf nodes may be labeled using Unicode strings.

2.2 Mapping XML to the Logical Object Model

A small wrapper module is used to map the XML model with its node types and attributes to the simple tree model and vice versa. A sample tree for a document fragment is shown in Figure 1.

2.2.1 Mapping XML Document Nodes to Logical Nodes

Elements are mapped one-to-one to tree nodes of the logical model. Attributes are mapped to child nodes of an additional *attribute container* child node, which is always the first child of the element node the attributes belong to. Attributes, PCDATA, CDATA nodes and comments are stored as leaf nodes, using reserved integer values as node label.

External entity references are expanded during import, while retaining the name of the referenced entity as a special inner node.

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

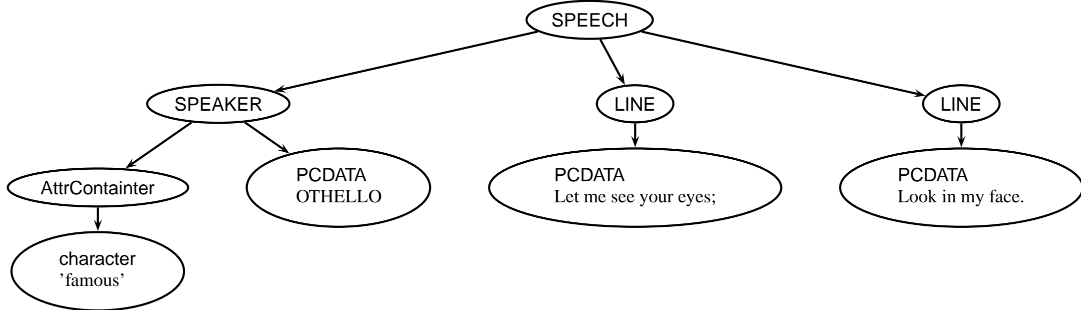

Figure 1: An XML fragment and its logical tree

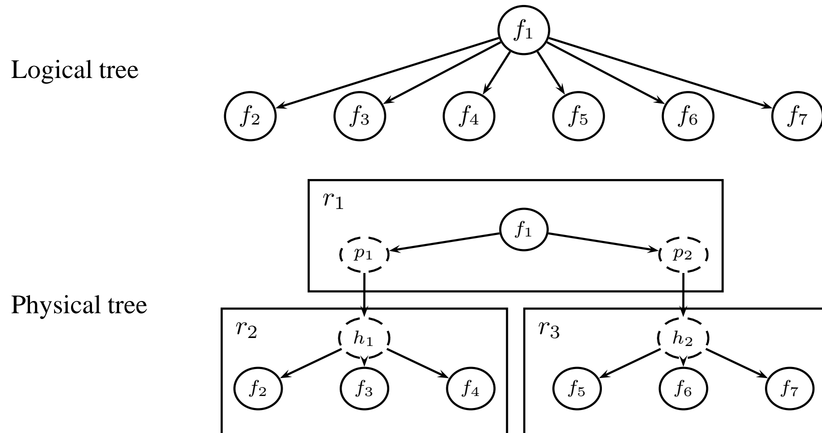

Figure 2: One possibility for distribution of logical nodes onto records

2.2.2 Mapping XML Tags to Tree Labels

The module which maps XML to the internal model uses a separate data structure to map tag and attribute names to integers, which are used as Σ_{Tags}. All the documents in one XML collection share the same mapping, which makes query evaluation simpler and more efficient because the possible integer values for a given tag or attribute name can be resolved once per query and stay the same for all documents in the collection. We call the integer labels `DeclarationID`s.

2.3 Physical Object Model

We partition the tree into subtrees, in which *Proxy nodes* are used to refer to connected subtrees not stored in the same record. Their contents is the RID of the record containing the subtree they represent. Substituting all proxies by their respective subtrees reconstructs the original data tree.

A sample is shown in Figure 2. To store the given logical tree (which, say, does not fit on a page), the physical data tree is distributed over the three records r_1, r_2 and r_3. Two proxies (p_1 and p_2) are used in the top level record. Two helper aggregate nodes (h_1 and h_2) have been added to the physical tree. They group the children below p_1 and p_2 into a

tree. Proxy and helper aggregate nodes are drawn as dashed ovals. They are only needed to link together subtrees contained in different records, and are are called *scaffolding nodes*. Nodes drawn as solid ovals represent logical nodes (f_i), and are called *facade nodes*. Only facade nodes are visible to the caller of the XML segment interface.

The given physical tree is only one possibility to store the sample logical tree. More possibilities exist since any edge of the logical tree can be represented by a proxy. The maintenance of the physical tree during incremental updates is described in [7]. The initial creation of a physical tree for a newly imported document is the core functionality of the bulkload component described in this article.

The following section will explain how the individual subtrees are materialized.

2.4 Physical Subtree Model

Each subtree is stored in a single record and, hence, must fit on a page. Each subtree represents part of a logical tree as defined in Section 2.1. In addition to leaves labelled with strings, physical subtrees also contain another kind of leaf node, which is labelled with references to other subtrees.

Every subtree also has two additional attributes. A *parent record* RID points to the parent subtree (if it exists), and a logical document ID field allows to determine which document this subtree belongs to.

Classified by their contents, there are three types of nodes in subtrees:

Aggregate nodes represent inner nodes of the logical tree.

Literal nodes represent leaf nodes of the logical tree and contain text strings. If a literal is larger than a page, it is split into chunks which are processed in the same way as large logical trees.

Proxy nodes are subtree leaf nodes which contain physical references to other records. They are used to link trees together that were partitioned into subtrees (see 2.3). Proxies also represent a major difference between Natix' format and the System RX format [2], where the proxies contain logical identifiers which may be annotated with metadata about the target subtree. Our bulkload algorithm is not concerned with the representation of the links, and can also be applied to formats that employ logical links.

3 Requirements

We now turn to the requirements for a bulkload component that creates the persistent data structures from an external document representation.

We base our design of the bulkload component on three goals, all of which are performance-related.

1. The interface should closely match the typical output of XML parsers.

   XML parsers are the most common source of imported XML documents, and many XML tools, among them query evaluation components, are able to efficiently deliver results using parser-like interfaces. Hence, it is very reasonable to assume that the data to be bulkloaded is delivered as XML parser output.

   We do not want to waste resources by requiring to change the data representation before or while accessing the bulkload component, in addition to potential representation changes for the actual transfer to the persistent storage format.

2. The mechanism should not require main memory proportional to the document size.

Linear memory usage would prohibit the import of documents larger than available main memory. As a generalization, the total amount of concurrently importable documents would be limited by available physical memory.

3. The produced storage layout should be efficient for typical workloads on documents.

   We identify three subgoals.

   (a) A dominant access pattern for document trees is the preorder traversal of subtrees induced by inner nodes. It is used when exporting documents and document fragments to their textual representation. Query evaluation on XML documents typically also relies on preorder traversals, such as the evaluation of XPath `descendant` and `descendant-or-self` axes. The default bulkload strategy therefore is to create a layout which adequately supports preorder traversal.

   (b) Given a set of children, we assume that the access frequency of sibling nodes decreases with their order. Typically, the leftmost children are accessed more often than the last children. For example, to reach any child by position in its sibling sequence, in Natix storage format, all left siblings of the target node need to be visited. Hence, the likelihood of being stored in the same record as the parent node should be higher for left siblings.

   (c) The number of clusters or subtrees should be as small as possible, because traversal of inter-cluster borders is much more expensive than intra-cluster traversal. Hence, fewer clusters imply higher query performance.

4. The produced storage layout should have minimal space requirements. This also implies a minimal number of clusters, because each cluster induces storage overhead in form of proxies and helper aggregates.

The goals imply some kind of clustering algorithm that partitions a tree into a minimum number of subtrees with limited size, which can then be used as Natix XML subtree records.

4 Tree Clustering Algorithms

There are efficient clustering algorithms applicable to weighted tree structures which consider the problem of creating a clustering of a tree which minimizes the number of generated clusters. Some of them are reviewed in this section.

However, the clusterings generated by the existing algorithms always have the following properties: (1) The weight of each cluster has an upper limit, which is a parameter of the algorithms. The weight of a cluster is the sum of the weight of its nodes. (2) All nodes of a cluster are connected.

In our case, a cluster is a physical record in our subtree model (Sec. 2.4), and the node weight is the size of the node (without its subtree) in bytes. Hence, the upper limit must be a value smaller than or equal to the disk page size.

Unfortunately, our storage format does not match well with the stated constraints on clusters, because in our case (1) the storage cost of a cut edge is not 0, as a cut edge causes overhead in the form of a proxy node and a new physical record header, and (2) it is possible to put several different siblings into a single cluster, creating nonconnected partitions of the tree.

Note that these issues apply to many other conceivable tree storage structures, because (1) any storage scheme must materialize the whole tree structure, not only the uncut edges, and (2) even if efficient sibling clustering is not explicitly supported by a format, it is still desirable to perform implicit clustering of siblings by placing them on the same disk page.

4.1 Workload-Directed Algorithms

(Weighted) Depth-First Search [16] processes a graph using depth-first search, assigning nodes greedily to the current cluster. New clusters are created whenever the current cluster cannot hold the current node. The resulting clustering is not compatible with our storage structure, as the preorder traversal may cause nonconnected subtrees to be clustered together. The cost of cut edges is also not taken into account. In the weighted variant, the algorithm also accounts for edge weights that represent traversal frequencies. Here, the edges to visit are ordered by weight to avoid cutting heavily used edges. The weighted algorithm requires a main-memory representation of the document.

Lukes [12] presents a linear time algorithm that incorporates edge weights and find a clustering of optimal value, e.g. one where the total weight of all edges that do not cross clusters are maximized. For unit edge weights, the algorithm finds the smallest possible clustering. However, the algorithm has very large constants; its running time is $O(nk^2)$ where n is the number of nodes and k is the weight limit. [4] report running times of several hours on modern PCs for very small documents ($\sim$100K). The algorithm also requires a main-memory representation of the document and intermediate results, does not consider sibling clusterings, and also does not observe costs for cut edges.

Bordawekar et al. [4] extend Lukes by introducing several techniques to limit memory usage and improve running time. This breaks the optimality, but achieves approximate clusterings whose value is quite close to the optimum. Again, cut edges and sibling clusterings are not considered. As we will see in Sec. 6, the performance of the algorithm is inferior to Natix' algorithm, even though their measurements only reflect the actual clustering phase, and not the construction of the persistent data structures and associated costs, such as logging.

Schkolnick [14] partitions hierarchical structures based on access patterns. However, the algorithm does not enforce a size limit for clusters, and does not consider nodes of varying weight. The algorithm has a different objective than space-efficient bulkload; it clusters objects into base collections, which can be joined to efficiently answer queries. While this may be applied to join-based XML query processing, it does not solve our problem of finding weight-limited clusters.

4.2 The Algorithm by Kundu and Misra

As a foundation of our own bulkload algorithm, we have chosen the one by Kundu and Misra [11], which creates a clustering of a tree with weighted nodes, where each cluster is connected and has at most weight k, and where the number of clusters is minimal.

To simplify the description of our own algorithm, we first outline the original algorithm, and discuss its shortcomings in more detail.

4.2.1 Outline

The algorithm pursues a bottom-up approach, successively assigning clusters to nodes. A node is processed only after its sons have been processed. *Processing* a node x guarantees that the weight of the subtree rooted at x is smaller than k. The *weight of a subtree* is the sum of all weights of those nodes in the subtree which have not been assigned to a cluster. While the subtree weight is larger than k, new clusters are created for sons of x, each containing the subtree including the son and all descendant nodes that are not yet assigned to a cluster. Partitions are created for the sons in descending order of their subtree weight. Once the subtree rooted at x has a weight less than k, processing of x is finished. When this algorithm has reached the root node of a tree, the resulting clusters are smaller than k, and a minimum number of clusters containing connected subtrees has been generated (Refer to [11] for a proof).

4.2.2 Suitability as bulkload algorithm

Document bulkload is easily translated into a problem instance for the algorithm above. Document tree nodes have a weight proportional to their space usage, clusters are stored as physical records, and the limit for the size of a physical record is the system page size. The algorithm generates physical records in a bottom-up manner, so that subtrees induced by some inner nodes are in as few physical records as possible. This prepares preorder traversals of document fragments, as required when exporting or traversing such subtrees when evaluating queries.

However, a bulkload algorithm for Natix needs to address some additional issues as explained above:

1. We do not want to keep the whole document tree in memory.
2. There is an overhead weight associated with a physical record, because the standalone header and the proxy node in the referring record occupy space.
3. Neighbouring siblings can be assigned to the same physical record, amortizing the overhead weight over several subtrees.
4. The leftmost siblings should have a higher probability of being clustered with their parent.

The first issue can easily be addressed, since the algorithm's bottom-up approach does never change a node's assignment to a cluster. Hence, once a cluster has reached the size limit, it can be stored in a physical record on disk and the constituent nodes need not be retained in main memory.

The weight limit for a cluster is called *cluster limit* in the following. A cluster limit smaller than the capacity of a disk page may be used to avoid fragmentation. Since the actual cluster sizes can vary with tree structure and text node sizes, a cluster does not always closely approach the limit. Hence, many underutilised pages may be created. In Natix, the cluster limit is set by default to a quarter of the disk page size, to allow several clusters to share a page and thus improve space utilisation.

5 Natix Bulkload Component

Based on the requirements stated in the previous section, we now present the design and implementation of the Natix Bulkload Component. We begin with the Bulkload API that is used to import an external document, and then elaborate on our clustering algorithm.

5.1 Interface

Figure 3 shows the internal bulkload interface for XML collections[1].

The document tree to bulkload is ”described” to the segment in form of a sequence of ”visit events” resulting from a depth-first search of the tree. The bulkload user signals these events to the bulkload component by calling appropriate functions each time a node is visited.

This corresponds directly to parser interfaces such as SAX [13] or libxml [17]. These generate parsing events which correspond to a depth-first search of the abstract syntax tree. Clients need to register callbacks with the parser which are invoked when the associated event occurs. Each SAX event can be directly translated into a single call of the bulkload interface[2].

[1] Natix internally organizes storage in segments, hence the identifier XMLSegment

[2] Attributes are an exception, as they are delivered as a list together with the parent element. This is a design error in the SAX interface, avoided in libxml2.

```
class SEG_XMLSegment : public SEG_SlottedPageSegment
{
public:
[...]
  class BulkloadContext;
  BulkloadContext *beginBulkload(const DocumentID &doc, DeclarationID logt,
                                 uint32_t childcount, uint32_t sizehint);
  void beginInternalNode(BulkloadContext *context, DeclarationID lt, uint32_t children);
  void endInternalNode(BulkloadContext *context);
  void addLiteralNode(BulkloadContext *context, DeclarationID lt,
                      uint32_t contentsize, ptr_t content);
  NID endBulkload(BulkloadContext *context);
[...]
};
```

Figure 3: XML bulkload interface

The first visit of the document root node initializes the bulkload (`beginBulkload()`), and the second visit (`endBulkload()`) terminates the bulkload and returns the node identifier of the stored root node. The `beginBulkload()` call allows to specify a size hint for the document. For small documents, this allows to fit the document into a matching gap on an already used page.

When visiting nonliteral nodes (`beginInternalNode()`) for the first time, the caller may specify how many children the internal node has, if known. After all descendants of the node have been added, `endInternalNode()` is called.

When visiting leaf nodes which are labeled with strings, `addLiteralNode()` is called.

5.2 Bulkload Algorithm

We now explain the variant of the Kundu and Misra [11] algorithm used in Natix. After giving a top-level explanation on how to extend the algorithm for our XML storage format, we elaborate on the details, using C++-like pseudocode to specify the routines involved.

5.2.1 Extending the Kundu and Misra Algorithm

As explained in Sec. 4.2, three remaining issues need to be addressed by our algorithm: (1) The overhead weight associated with a physical record. (2) Siblings can be clustered to reduce this overhead. (3) The leftmost siblings should have a higher probability of being clustered with their parent.

The overhead weight is dealt with in the detailed algorithm description below.

The possibility of sibling clustering introduces another degree of freedom when processing nodes. Instead of choosing the heaviest son first when creating new subtrees, it is now possible to create an ”artificial” heaviest son by grouping consecutive siblings together into one physical record. This can also be used to address our remaining issue: Make clustering of leftmost sons with their parent more likely. We can now store some of the rightmost sons together in a separate physical record, while keeping a heavier son further to the left in the same cluster as its parent.

More precisely, instead of choosing the heaviest son to be assigned to a separate cluster from the parent, Natix combines some of the *rightmost*, unassigned, consecutive children of the currently processed node and clusters them into physical records smaller than the cluster limit. This amortizes the record overhead over several nodes. It also increases the likelihood of the leftmost children to be clustered with the parent node.

Unfortunately, the changes described above break the optimality guaranteed by the original algorithm. This demotes the Natix algorithm to a heuristic with respect to min-

```
void SEG_XMLSegment::beginInternalNode(BulkloadContext *context, DeclarationID id)
{
  context->current()->appendNode(new BulkloadNode(id));
}
```

Figure 4: Code for `beginInternalNode()`

imum number of records generated. It is not clear how the bottom-up algorithm can be modified to retain optimality. If at all possible, a combinatorical approach seems likely that would have to select the least costly of all possible partitionings of records into clusters. We were not able to find an algorithm to do this efficiently. Since efficiency is of great importance for document import, and the heuristic algorithm explained below generates very good clusterings in all observed cases, we consider a slightly suboptimal clustering acceptable.

5.2.2 Detailed Description of the Natix Algorithm

The algorithm maintains a main-memory tree which consists of nodes that have not been assigned to a cluster yet. The main-memory tree nodes are stored using native C++ pointers for parent references, and sets of child pointers in each node. The main-memory tree also includes main-memory versions for proxies referencing subtrees which have already been assigned to clusters and moved to physical records. The worst-case size of this main-memory tree is proportional to the height of the document tree, i.e. the maximal path length from the root node to a leaf node in the document. This property is guaranteed by keeping, on each level, only as many nodes as fit within a certain configured memory limit (see below).

In the beginning, bulkload starts with an empty main-memory tree. Every call to the interface functions to construct the document either results in a new main-memory node, or transfers some of the main-memory nodes to the storage representation by assigning them to a cluster, or both.

To simplify the exposition, we only consider treatment of the `beginInternalNode()` and `endInternalNode()` functions. Calls to `addLiteralNode()` can be regarded as calls to `beginInternalNode()` immediately followed by `endInternalNode()`.

The `beginInternalNode()` code simply adds the new node to the main memory tree (Figure 4). The node is buffered in this main-memory tree since it only can be processed until its complete subtree has been described using the bulkload interface.

When `endInternalNode()` is called (Figure 5), the current node's subtree has been completely visited by the depth-first traversal, and it can be processed. The function `pruneCurrentCluster()` is called to guarantee that the node's subtree is smaller than the cluster limit. Then, the parent of the current node becomes the new current node, and its weight is increased by the subtree weight of the node for which `endBulkload()` was called. Finally, if the size of the main memory tree below the current node has reached a certain constant threshold, we start to create physical records to reduce the amount of memory occupied by the bulkload, even if the cluster limit has not been reached. The threshold is called *memory limit*. It is the cluster limit times an integer *memory factor* m. In Sec. 6, we will show that above the Natix default $m = 5$, the performance gains are negligible.

Figure 6 shows the code for pruning the main-memory tree. If the subtree below the current node together with standalone record header is large than the cluster limit, then the children of the node are clustered into physical records until the size of the main memory subtree falls below the cluster limit. The `IGNOREPROXIES` identifier is explained below.

```
void SEG_XMLSegment::endInternalNode(BulkloadContext *context)
{
  BulkloadNode *processed=context->current();
  pruneCurrentCluster(context);
  context->current(processed->parent());
  context->current()->addWeight(processed->weight());
  if(context->current()->weight() > m * clusterLimit())
    pruneCurrentCluster(context);
}
```

Figure 5: Code for `endInternalNode()`

```
void SEG_XMLSegment::pruneCurrentCluster(BulkloadContext *context)
{
   BulkloadNode *current=context->current();

   if(current->weight() + clusterOverhead() > clusterLimit())
      clusterChildren(context, IGNOREPROXIES);

   while(current->weight() + clusterOverhead() > clusterLimit())
      clusterChildren(context, CLUSTERPROXIES);
}
```

Figure 6: Code for `pruneCurrentCluster()`

During pruning of the tree, physical records are created which contain subtrees of the main-memory tree. These main-memory subtrees are replaced with main-memory proxy nodes. Therefore, even after creating clusters and removing the nodes from the main-memory tree, the remaining proxy nodes may still cause the subtree to be larger than the cluster limit. Hence, in the `while` loop the proxy nodes themselves are grouped into clusters and physical records are created for them, possibly in several levels, until the subtree fits into the cluster limit.

The `clusterChildren()` function (Figure 7) determines the cluster boundaries, moves clustered subtrees into physical records, and replaces the subtrees with proxies in the main-memory tree. Note that the grouping of child nodes into clusters proceeds from right to left, making sure that nodes further to the right are more likely to be clustered, as specified in our requirements.

Instead of showing code, we will only briefly describe the lower-level functions required by `clusterChildren()`. The `findClusterBoundRight()` and `findClusterBoundLeft()` functions determine the interval of those children of the current node that are to be included in a new physical record. `findClusterBoundRight()` looks for nodes satisfying a predicate that depends on the `mode` parameter. The search starts at the second argument `lastsplit` and continues to the left siblings. If `mode == IGNOREPROXIES`, then the predicate is true for all non-proxy nodes. Otherwise, any node qualifies.

`findClusterBoundLeft()` moves further right starting from the rightmost node of the new partition. It includes nodes into the interval while they satisfy the same predicate as above, and while the closed interval of subtrees bounded by `firstsplit` and `lastsplit` still fits into a physical record.

`createRecord()` is straightforward and creates new subtree records from the main-memory representations. If main-memory proxy nodes are included in the subtree, they are inserted into the physical record, and their target record's parent pointer is updated to refer to the new physical record.

`replaceWithProxy()` removes the main-memory representation of the subtrees that have been moved to a record and inserts a proxy instead.

```
void SEG_XMLSegment::clusterChildren(BulkloadContext *context, ClusterMode m)
{
  BulkloadNode *current=context->current();
  BulkloadNode *lastsplit=current->lastChild();

  lastsplit=findClusterBoundRight(context,lastsplit,mode);

  while(lastsplit!=0 &&
        current->weight() + clusterOverhead() > clusterLimit() )
  {
    BulkloadNode* firstsplit;
    firstsplit=findClusterBoundLeft(context,lastsplit,mode);
    RID target=createRecord(context,firstsplit,lastsplit,false);
    BulkloadNode* nextsplit=firstsplit->leftSibling;
    replaceWithProxy(context,current,firstsplit,lastsplit,target);
    lastsplit=nextsplit;
    lastsplit=findClusterBoundRight(context,lastsplit,mode);
  }
}
```

Figure 7: Code for `clusterChildren()`

Memory management The main-memory representation consists of a large amount of small objects. In the case of literals, these are even of variable size.

In spite of this, memory management is not expensive during bulkload. Memory is allocated for the nodes during a depth-first traversal. In depth-first preorder, all nodes of a subtree form a consecutive interval of nodes. This makes it possible for the bulkload component to use a special memory management technique. The special memory manager requests memory in blocks of constant size from the operating system, adding nodes to blocks in depth-first preorder as they are delivered to the bulkload component. The order in which the blocks are used is maintained in a list. When a subtree's main memory representation is no longer used, the interval of blocks which only contain nodes of this subtree can be deallocated in a per-block fashion, without regarding the individual nodes on the blocks.

6 Evaluation

This section presents experimental results to assess the performance of the Natix Bulkload Component. We examine the effect of sibling clustering, the scalability with respect to document size, and compare Natix to other XDSs.

6.1 Document Collections

We performed experiments using three document collections. The first is the XMark benchmark [15] using scaling factors of $n \times 0.2$ with $n \in \{1 \ldots 5\}$. The second is a sythetic document collection generated using the ToXgene data generator [1]. The DTD as well as the generator template file are listed in [6]. The smallest document contains 50 employees, 100 students, 10 lectures and 30 exams. We generated 6 documents. With each document we quadrupled these numbers, so that the biggest document contains 51200 employees, 102400 students, 10240 lectures and 30720 exams. This leads to document sizes between 59kB and 43MB.

6.2 Environment

The system used for the experiments ran on two machines.

Machine `NEW` was used for all experiments except for the comparison to the older benchmark results (Sec. 6.4.4). It was equipped with 512MB RAM, a Pentium IV CPU

with 2.4 GHz, and an UltraWide SCSI hard disk. The operating system was SuSE Linux 9.3 with kernel version 2.6.11.

Machine `OLD` was used to reproduce the environment from [15], and had 512MB of RAM, a Pentium III running at 600 MHz, and an Ultra Wide SCSI disk.

Natix was compiled with g++ 3.3.5 using optimization level O3.

The measured times are the total elapsed time to import the document, including full logging and recovery support. A main memory page buffer with sufficient memory to hold the whole document was used. The times do not include system startup time (about 0.1s), and the page buffer was not flushed during bulkload. However, the times do include commit processing and flushing of the log.

For the comparison to MonetDB [3], we used Monet Database Server V4.8.0 with the Pathfinder module as publicly distributed. We present the import times reported by the Monet console.

6.3 Algorithms

For Natix, we implemented the algorithm as explained in this article, using a default value of $m = 5$ except where stated otherwise. A disk page size of 8K was used, and the cluster limit was set to 2K to avoid fragmentation (see Sec. 4.2.2).

We also implemented a modified variant of the Kundu algorithm to compare our approach against optimal partitioning without sibling partitions. We had to modify Kundu to incorporate the fact that the weight of a cluster is modified by the additional proxy nodes. This was done using three modifications. First, while processing a node, the weight of added proxies was added to the node. Second, nodes whose weight was smaller than or equal to a proxy node were always clustered with their parent, because clustering them would not decrease the weight of the parent node. Third, the Kundu algorithm has to deal with the case that the physical representation for a single node with proxies for all its children and small nodes clustered with the parent does not fit into the cluster limit. In this case, and only in this case, we used the same approach as in the Natix Algorithm, namely to partition the proxy nodes and the small regular nodes from right to left by clustering them into ”intermediate clusters” of maximal weight which were referenced by a proxy in the parent's cluster (See `clusterChildren()` in Sec. 5.2.2). As we will see in the experimental results, this rarely occurs.

6.4 Results

6.4.1 The Importance of Sibling Clustering

With the first series of experiments, we wanted to illustrate the importance of sibling clustering.

Hence, we took the XMark document with scaling factor 0.2, producing a document about 20MB in size, and bulkload it using the modified Kundu and the Natix algorithm. For the Natix algorithm, we used different values for the m parameter (see Sec. 5.2.2).

The number of clusters generated are shown in Table 1. The modified Kundu algorithm produces about 50% third more clusters than the Natix algorithm with values $m > 1$. This demonstrates that even a heuristic for sibling clustering can significantly outperform the optimal single child clustering case. Note that the number of nodes for which intermediate clusters (see above) had to be created for the Kundu algorithm was less than 750, and did not significantly distort the results.

For $m = 1$, the Natix algorithm does not perform well. This is expected, because once it reaches that limit, it immediately creates new clusters for any additional node, instead of delaying clustering decisions until more siblings are available. It performs even worse than the Kundu algorithm, because it degenerates to a non-optimal single child clustering.

Method	Clusters
Kundu (Optimal Single Child Clustering)	30198
Natix (Sibling Clustering, $m = 1$)	33929
Natix (Sibling Clustering, $m = 2$)	22852
Natix (Sibling Clustering, $m = 3$)	22117
Natix (Sibling Clustering, $m = 5$)	21895
Natix (Sibling Clustering, $m = 10$)	21779
Natix (Sibling Clustering, $m = \infty$)	21692

Table 1: Number of Clusters for XMark SF 0.2

Document	Size (10^3 bytes)	MonetDB	Natix
xmark 0.2	22514	2.16s	5.34s
xmark 0.4	46693	4.52s	10.76s
xmark 0.6	70322	9.88s	16.46s
xmark 0.8	93560	12.03s	22.74s
xmark 1.0	105264	16.03s	27.98s
uni1.xml	58	0.03s	0.02s
uni2.xml	166	0.04s	0.09s
uni3.xml	673	0.08s	0.19s
uni4.xml	2704	0.31s	0.81s
uni5.xml	11053	3.27s	3.08s
uni6.xml	44360	28.70s	13.67s

Table 2: Import times (seconds) for Natix and MonetDB

For $m > 1$, the number of clusters quickly converges against the best case achievable by the Natix algorithm with unlimited memory, which is shown in the last row.

6.4.2 Scalability

In our second experiment, we wanted to show the scalability of our approach, and compare it to the scalability of a non-clustering approach.

We imported the two document collections into Natix and MonetDB/Pathfinder [3]. MonetDB is a relational main-memory DBMS that stores XML as binary relations in which the nodes are stored in preorder, i.e. in the order delivered by the parser. In such a format, no clustering is required, but only a preorder traversal is supported as efficient access path, and updates may be costly.

The results from Table 2 show that the Natix Bulkload Algorithm exhibits a running time linear in the document size. For the XMark documents, MonetDB is about twice as fast, and also scales linearly. For the uni documents, the Natix behaviour does not change, the scalability and bulkload speed remain similar to the XMark case. MonetDB, however, shows a different behaviour and is slower and scales worse. We were not able to find the cause.

We conclude that the clustering approach employed by Natix performs and scales adequately, and can keep up with a non-clustering approach.

6.4.3 Comparison with XC

XC is a XML clustering algorithm developed at the IBM Watson Research Center in Yorktown heights [5] (See Sec. 4.1). Their optimized version of Lukes is a workload-directed algorithm that generates good clusterings tailored to previously configured workloads. How-

Document	Size	XC	Natix
SigmodRecord.xml	467K	2.82s	0.27s
mondial-3.0.xml	1.8M	22.69s	0.58s
partsupp.xml	2.2M	6.54s	0.49s
uwm.xml	2.3M	6.78s	0.91s
orders.xml	5.2M	18.86s	1.25s

Table 3: Comparison with XC (Import time)

System	Bulkload time (Seconds)
System A (from [15])	414
System B (from [15])	781
System C (from [15])	548
Natix	215

Table 4: XML Bulkload Times for various systems

ever, it does not have acceptable performance for online bulkloads. We show some of their results in Table 3. The table also includes Natix import times for the same documents.

The XC system is written in C++, and the experiments were performed on an x86-based Linux system with 1.7 GHz CPU speed. The Natix results were obtained on our Machine `NEW` with 2.4 GHz. The results show running times for Natix which are faster by about an order of magnitude. This diffence is clearly beyond the difference in processor speed. In addition, their heuristical algorithm only performs single child clustering, which is inferior to sibling clustering with respect to the number of clusters, as demonstrated above.

6.4.4 Comparison with Other Published Results

Published bulkload performance results for XDS systems are rare and far between. The only comparable numbers we could find were from the XMark benchmark by Schmidt et al. [15]. They compare bulkload performance for XMark scaling factor 1 on various anonymous mass-storage systems. We repeat some of their results in Table 4.

We limit our comparison to the disk-based systems, omitting their numbers for purely main-memory based systems, as we do not know whether the main-memory based systems perform logging or checkpointing, and wether the numbers reflect that. The remaining systems are relational DBMS, and called ”System A”, ”System B” and ”System C” in the paper. No details about the employed mappings from documents to relations are given, except that systems A and B do not require a DTD, while system C requires to manually generate a relational schema from a DTD.

Table 4 also includes a measurement of Natix's bulkload performance for the same document. We used our Machine `OLD`, which is very similar to the one described in Schmidt et al. [15], except that it has less main memory (512MB compared to their 1 GB), and a slightly faster processor (600Mhz compared to their 550Mhz).

Although Natix outperforms the relational systems by factors between 1.9 and 3.6, few is known about the exact configurations and techniques used to store XML in the relational systems. Hence, it is unclear to what extent the numbers are comparable.

7 Conclusion and Future Work

This article discusses the Natix Bulkload Component, a module of the Natix XML Data Store that is responsible for efficiently converting external documents into the Natix storage format.

In our requirements analysis, we argued that a bulkload component for XML must address three important issues: First, it must be efficient and limited in its usage of resources such as computing power and memory. Second, the interface to the bulkload component must closely match the format in which external documents are delivered, avoiding expensive representation changes. Third, the generated persistent storage layout must be of high quality.

We clarified that for tree-structured data such as XML, a high quality of the storage layout is equivalent to a small number of generated *clusters*. Clusters represent subsets of the document tree that are closely related with respect to document structure, and that fit on a disk page. In the context of the Natix storage format, and similar approaches, such a cluster is a subset of the document nodes that is connected via parent-child and sibling relationships.

We assessed a number of existing algorithms for our purposes. Even the best candidate, the tree clustering algorithm by Kundu and Misra, failed to address all requirements, in particular because it keeps the whole document in memory, and because it does not cluster siblings.

Hence, we transformed the approach by Kundu and Misra into a novel clustering heuristic, the Natix Bulkload Algorithm. Albeit not optimal, this algorithm uses sibling clustering to produce 30% less clusters than an optimal single-child clustering. The algorithm has linear complexity with respect to the document size, while using space proportional to the document tree height.

We presented experimental results, which demonstrate the competitiveness of our bulkload component on several fronts: (1) We show that sibling clustering is superior compared to single-child clustering. (2) Our algorithm scales linearly with small constants. (3) Compared to highly efficient relational bulkload techniques that materialize the document in preorder as it arrives, the performance penalty that has to be paid for clustering is acceptable. (4) Our bulkload component is faster by at least an order of magnitude than existing workload-directed approaches that derive their clustering decisions primarily from expected access patterns.

In the future, we want to improve our heuristics for sibling clustering. We also want to incorporate information about access patterns into our algorithm without compromising bulkload performance.

References

[1] D. Barbosa, A. Mendelzon, J. Keenleyside, and K. Lyons. ToXgene: a template-based data generator for XML. In *SIGMOD Conference*, 2002.

[2] K. Beyer, R. J. Cochrane, V. Josifovski, J. Kleewein, G. Lapis, G. Lohman, B. Lyle, F. Özcan, H. Pirahesh, N. Seemann, T. Truong, B. V. der Linden, B. Vickery, and C. Zhang. System RX: one part relational, one part XML. In *SIGMOD Conference*, pages 347–358, 2005.

[3] P. A. Boncz, T. Grust, S. Manegold, J. Rittinger, and J. T. er. Pathfinder: Relational XQuery Over Multi-Gigabyte XML Inputs In Intera ctive Time. Technical Report INS-E0503, CWI, March 2005. MonetDB 4.8.0, Pathfinder 0.8.0.

[4] R. Bordawekar and O. Shmueli. Flexible workload-aware clustering of XML documents. In *Database and XML Technologies, Second International XML Database Symposium, XSym*, pages 204–218, 2004.

[5] R. Bordawekar and O. Shmueli. Flexible workload-aware clustering of XML documents. Technical report, IBM T.J. Watson Research Center, Yorktown Heights, May 2004.

[6] M. Brantner, S. Helmer, C.-C. Kanne, and G. Moerkotte. Classification, Translation and Optimization of Nested XPath Expressions. Technical report, University of Mannheim, 2005. http://pi3.informatik.uni-mannheim.de/˜msb/TRB2005CTO.pdf.

[7] T. Fiebig, S. Helmer, C.-C. Kanne, G. Moerkotte, J. Neumann, R. Schiele, and T. Westmann. Anatomy of a Native XML base management system. *VLDB Journal*, 11(4):292–314, 2003.

[8] C.-C. Kanne, M. Brantner, and G. Moerkotte. Cost-sensitive reordering of navigational primitives. In *SIGMOD Conference*, pages 742–753, 2005.

[9] C.-C. Kanne and G. Moerkotte. Efficient storage of XML data. Technical Report Nr. 8, Lehrstuhl für praktische Informatik III, Universität Mannheim, June 1999.

[10] C.-C. Kanne and G. Moerkotte. Efficient storage of XML data. In *ICDE*, page 198, 2000.

[11] S. Kundu and J. Misra. A linear tree partitioning algorithm. *SIAM J. Comput.*, 6(1):151–154, March 1977.

[12] J. A. Lukes. Efficient algorithm for the partitioning of trees. *IBM Journal of Research and Development*, 18(3):217–224, 1974.

[13] D. Megginson. SAX: A simple API for XML. Technical report, Megginson Technologies, 2001.

[14] M. Schkolnick. A clustering algorithm for hierarchical structures. *ACM Trans. Database Syst.*, 2(1):27–44, 1977.

[15] A. Schmidt, F. Waas, M. Kersten, M. J. Carey, I. Manolescu, and R. Busse. XMark: A benchmark for XML data management. In *VLDB Conf.*, pages 974–985, 2002.

[16] M. M. Tsangaris and J. F. Naughton. On the performance of object clustering techniques. In M. Stonebraker, editor, *Proceedings of the 1992 ACM SIGMOD International Conference on Management of Data, San Diego, California, June 2-5, 1992*, pages 144–153. ACM Press, 1992.

[17] D. Veillard. The XML C library for gnome. Project Web Site, 2002.