The importance of sibling clustering for efficient bulkload of XML document trees

C. C. Kanne
G. Moerkotte

In an XML Data Store (XDS), importing documents from external sources is a very frequent operation. Because a document import consists of a large number of individual node inserts, it is essentially a small bulkload operation, and thus efficient bulkload support is crucial for the performance of the XDS. The bulkload operation is in essence a mapping of an XML parser's output into the storage structures of the XDS. This involves two major subtasks: (1) partitioning the document's logical tree structure into subtrees that can be stored on a page in a way that is both space-efficient and suitable for later processing and (2) mapping the subtrees to the internal representation of the XDS for paging. In enterprise-scale environments with very large documents and many parallel bulkload operations, the first task is particularly challenging, as not only disk space consumption, but also CPU and main-memory usage are important factors. In this paper, we discuss the requirements for an XDS bulkload component and examine existing algorithms for tree partitioning and their applicability to the bulkload operation. We derive a new tree-partitioning algorithm for use in the bulkload operation and present the design of the bulkload component for the XDS Natix. Finally, we evaluate the performance of the bulkload component and compare our results with previous work.

INTRODUCTION

Loading large amounts of data which is already available in an external format is called a *bulkload* operation. In conventional database management systems (DBMSes), 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 (XML Data Store) needs to support document imports as regular operations that are used very frequently by applications. Hence, the bulkload becomes a core function whose performance is a determinant of overall system performance; however, we could find very few publications that discuss efficient XML bulkload in large-scale XML data stores.

This paper 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.[1] Besides requirements analysis and API (application programming 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, intercluster navigation is much slower than intra-cluster navigation, often by several orders of magnitude. Even if access reordering techniques are used,[3] the number of clusters is a crucial factor for query performance. Hence, the bulkload algorithm must minimize the number of clusters generated.

Our main contributions are

1. a detailed requirements analysis for XML bulk-load components,
2. an analysis of existing tree clustering algorithms,
3. a novel linear-time tree-clustering algorithm that generates up to 30 percent fewer clusters than the best known algorithms,
4. a description of the design and implementation of a concrete XML bulkloader, the Natix bulkload component, and
5. an evaluation of the Natix bulkload component.

The rest of this paper is organized as follows. In the next section, "XML storage," we give a brief overview of native tree storage in Natix. Next, in the section "Requirements," we analyze the requirements an XML bulkload component must meet. Then, in "Tree-clustering algorithms," we discuss existing tree-clustering algorithms with respect to these requirements. In the section "Natix bulkload component," we describe the interface and the implementation of the bulkload component. This section also covers our novel tree-clustering algorithm. In the section "Evaluation," we provide our experimental results, including a comparison with the bulkload performance of other XML storage systems. Our concluding remarks are presented in the last section.

XML STORAGE

The design of the bulkload component is strongly influenced by the format used for storing XML data in the XDS. 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 format for storing XML data (for details, refer to References 1, 4, and 5). Similar formats are also used in other XDSes, such as IBM's System RX.[2]

Logical tree model

The interface to our core storage engine uses a simple, general logical tree model with only two node types and no XML-specific constructs, such as attributes. This model enables a simple engine design and is easily mapped to concrete XML models such as the Document Object Model (DOM) or the simple API for XML (SAX), as explained in the next subsection.

The documents are represented as ordered trees in which nodes are labeled with symbols taken from an alphabet Σ. In the current implementation, we use the set of integers from 0 to $2^{16} - 1$ as Σ. Leaf nodes can additionally be labeled with arbitrarily long strings over a different alphabet (in the current implementation, this alphabet consists of the set of Unicode characters).

Mapping XML to the logical object model

A small wrapper module maps a concrete XML representation (such as DOM, SAX) 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*.

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 to which the attributes belong. Attributes, PCDATA (including whitespace-only data), CDATA nodes, and comments are stored as leaf nodes.

Mapping XML tags to tree labels

As alphabet Σ, the storage engine uses non-negative integers, which are called DeclarationIDs. The module that maps XML to the logical tree model also performs the mapping from tag names, attribute names, and special node labels (such as PCDATA) to

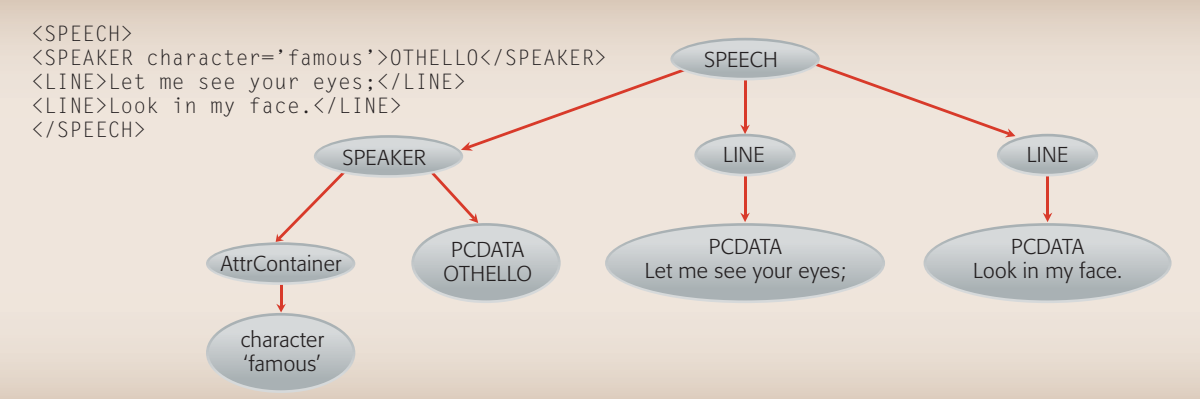

Figure 1
An XML fragment and the corresponding instance of its logical tree model

DeclarationIDs. All the documents in an 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.

Physical tree model

Natix organizes physical storage in units known as *XML segments*. An XML segment provides a storage area and an interface for storing and accessing a collection of logical tree instances. A physical tree is the mapping of a logical tree to physical storage and consists of records, each of which is smaller than a disk page and contains a subtree of the logical tree. In this section we use the term *record subtree* (or simply subtree) to refer to any subtree of the logical tree that is stored in a record. The records can be addressed using *record IDs* (RIDs).

A record subtree contains three types of nodes. *Aggregate nodes* represent the inner nodes of the tree and are labeled with non-negative integers (over the alphabet Σ). *Literal nodes* are leaf nodes that in addition to a label over the alphabet Σ can also have content in the form of a string. A *proxy node* is a special leaf node that points (refers) to a record subtree that is stored in a separate record and corresponds to a connected subtree of the logical tree. Substituting all proxies with the referenced subtrees reconstructs the original logical tree. Although proxies use RIDs to refer to the target subtree, this is an implementation detail and not a requirement for applying our techniques (one could use logical references, such as found in System RX[2]).

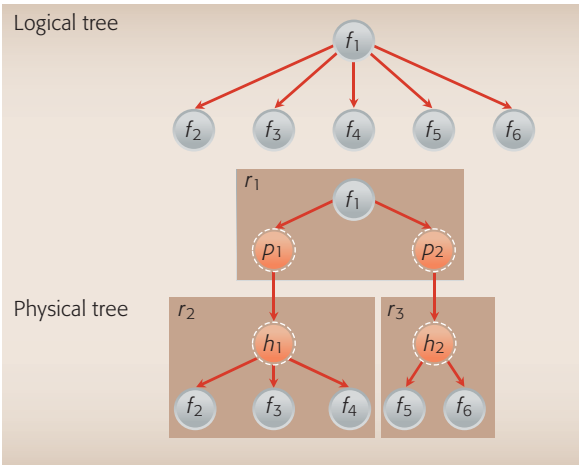

Figure 2
One possible way of mapping logical trees onto records

Every record subtree has two attributes: the *parent* RID, which points to the parent subtree (if it exists), and the *logical document ID* field, which determines the document to which this subtree belongs.

An example of a logical tree and its mapping onto records is shown in ***Figure 2***. To store the given logical tree (assumed not to 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 with dashed lines. They are only needed to link together subtrees contained in different records and

are called *scaffolding nodes*. Nodes drawn with solid lines 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 possible way of storing the sample logical tree. Additional possibilities exist as a proxy can be created on any edge of the logical tree. The maintenance of the physical tree during incremental updates is described in Reference 1. The initial creation of a physical tree for a newly imported document is the main function of the bulkload component described in this paper.

REQUIREMENTS

The requirements for a bulkload component are based on four goals, all of which are related to performance:

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 reasonable to assume that the data to be bulkloaded is delivered as XML parser output. Otherwise, changing the data representation before or during the bulkload operation would require expending additional resources.

2. *The bulkload operation 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 storage layout for imported XML documents should be optimized for performance for typical workloads (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, which involves translating the XDS internal representation to a document. 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 that adequately supports preorder traversal.
   B. Given a set of children, we assume that the access frequency to sibling nodes decreases with their distance from the parent node in the order of traversal. In Natix, for example, to reach a child node, all its left siblings have 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 intercluster borders is much more expensive than intracluster traversal. Hence, fewer clusters imply higher query performance.

4. *The storage required for imported XML documents should be minimized*. This also implies a minimal number of clusters, because each cluster induces storage overhead in the form of proxies and helper aggregates. As previously mentioned, the bulkload component maps the logical object model to the physical object model. Thus, the main task for the bulkload algorithm is to determine which subtrees of the logical model should be stored as physical subtrees, that is, where to locate the scaffolding nodes.

In the next section we examine existing tree clustering algorithms and discuss how closely they fulfill the preceding goals.

TREE-CLUSTERING ALGORITHMS

There are efficient clustering algorithms (applicable to weighted tree structures) that partition the nodes of a tree in a way that minimizes the number of generated clusters. However, the clusters generated by the existing algorithms 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), and (2) all nodes of a cluster are connected. In our case, a cluster occupies a physical record, 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 problem is slightly more complicated than mere assignment of logical nodes to

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 adjacent siblings into a single cluster that does not contain their parent node, thus creating unconnected subtrees within the same cluster (sibling clustering).

Note that these issues apply to many other conceivable tree storage structures because (1) any storage scheme must deal with the entire 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. As explained earlier, our bulkload algorithm has to solve a more general problem than that solved by existing tree-clustering algorithms. The fundamental objective for a tree-clustering algorithm for bulkload operations is to find a minimal number of weight-limited clusters. Hence, in the remainder of this section, we review existing tree-clustering algorithms to find a good starting point for a new bulkload algorithm.

Workload-directed algorithms

Depth-first search[6] applied to (weighted) graphs assigns nodes to the current cluster in a "greedy" manner. New clusters are created whenever the current cluster cannot include the additional node. The resulting clustering is not compatible with our storage structure, as the preorder traversal may cause unconnected 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. This reordering requires, in the worst case, that the entire document be kept in main memory.

Lukes[7] presents a linear-time algorithm that incorporates edge weights and finds an optimal clustering, that is, one that maximizes the total weight of all edges that do not cross clusters. For unit edge weights, the algorithm finds the clustering with the smallest possible number of clusters. However, the algorithm has very large resource requirements; its running time is $O(nk^2)$ where n is the number of nodes and k is the weight limit. In Reference 8, running times of several hours on modern PCs for very small documents (~100K) are reported. The algorithm uses dynamic programming and creates and maintains a large number of intermediate clusterings, which can take up more memory than the original document. In addition, it does not consider sibling clusterings and does not take into account costs for cut edges.

Bordawekar and Shmueli[8] extend Lukes by introducing several techniques to limit memory usage and improve running time. This breaks the optimality but achieves clusterings whose values are quite close to the optimum. Again, cut edges and sibling clusterings are not considered. As we will see in the section "Evaluation," the performance of the algorithm is inferior to the 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[9] 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. Although this may be applied to join-based XML query processing, it does not solve our problem of finding weight-limited clusters.

The algorithm by Kundu and Misra

As the starting point for our own bulkload algorithm, we have chosen the algorithm by Kundu and Misra,[10] 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 facilitate the description of our own algorithm in a later section, we now provide a brief description of the original algorithm and discuss its suitability as a bulkload algorithm in more detail.

The Kundu and Misra algorithm pursues a bottom-up approach, successively assigning clusters to nodes. A node is processed only after its children have been processed. Having processed node x ensures 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 that have not been assigned to a cluster. As long as the subtree weight is larger than k, new clusters are created for

children of x, each containing the subtree including the children and all descendant nodes that are not yet assigned to a cluster. Partitions are created for the children in descending order of their subtree weight. Once the subtree rooted at x has a weight less than k, the processing of x is complete. When this algorithm has reached the root node of a tree, the clusters produced are smaller than k, and a minimum number of clusters containing connected subtrees has been generated (see Reference 10 for a proof).

Suitability as bulkload algorithm

The Kundu and Misra algorithm is easily applied to the clustering problem for the bulkload operation. 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 clusters in a bottom-up manner by constructing optimal clusterings for higher levels of the tree by combining optimal clusterings of subtrees. This prepares preorder traversals of document fragments, as required for document export or when traversing such subtrees for the purpose of evaluating queries.

In addition, however, a bulkload algorithm for Natix has to address issues such as those mentioned earlier:

1. Keeping the entire document tree in memory should be avoided.
2. There is overhead associated with a physical record because the stand-alone header and the proxy node in the referring record require storage space.
3. Neighboring siblings can be assigned to the same physical record, amortizing the overhead 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 because the algorithm's bottom-up approach does not 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.

We refer to the weight limit for a cluster as the *cluster limit*. A cluster limit smaller than the capacity of a disk page may be used to avoid fragmentation. Because the actual cluster size can vary with the tree structure and the size of text nodes, the cluster size is often less than its limit, and thus, many pages are underutilized. In Natix, the cluster limit is set by default to a quarter of the disk page size. This allows several clusters to share a page and thus improves space utilization.

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.

Interface

Figure 3 shows the internal bulkload interface for XML collections. Natix internally organizes storage in so-called segments, hence the identifier XMLSegment.

As input, the bulkload component expects a document tree in the form of a sequence of "visit events" resulting from a depth-first traversal of the tree. The entity that uses the bulkload interface signals these events to the bulkload component by calling appropriate functions each time a node is visited.

The bulkload interface corresponds directly to parser interfaces such as SAX[11] or libxml.[12] These generate parsing events that correspond to a depth-first search of the abstract syntax tree. Clients need to register callbacks with the parser, and these callbacks are invoked when the associated event occurs. Each SAX event can be directly translated into a single call of the bulkload interface (attributes are an exception because they are delivered as a list, together with the parent element). 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 a size hint for the document to be specified. For small documents, this allows the document to be fitted into a matching gap on a partially filled page.

When visiting nonliteral nodes (`beginInternalNode()`) for the first time, the caller may specify how

```
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);
  void abortBulkload(BulkloadContext *context);
[...]
};
```

Figure 3
XML bulkload API

many children the internal node has, if known. After all descendants of the node have been added, `endInternalNode()` is called. When leaf nodes that are labeled with strings are visited, `addLiteralNode()` is called.

Bulkload algorithm

We now describe the variant of the Kundu and Misra[10] 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.

Extending the Kundu and Misra Algorithm

As previously mentioned, three remaining issues need to be addressed by our algorithm: (1) the overhead weight associated with a physical record, (2) the ability to cluster siblings in order to reduce this overhead, and (3) ensuring that the leftmost siblings have a higher probability of being clustered with their parent than other siblings. The issue of overhead is dealt with in the detailed algorithm description below.

The possibility of sibling clustering introduces another degree of freedom when nodes are processed. Instead of choosing the "heaviest" child first when creating new subtrees, it is now possible to create an "artifical" heaviest child by grouping consecutive siblings together into one physical record. This also can be used to address our remaining issue: make clustering of leftmost children with their parent more likely. We can now store some of the rightmost children together in a separate physical record and, at the same time, keep a heavier child further to the left in the same cluster as its parent.

More precisely, instead of choosing the heaviest child to be assigned to a separate cluster from the parent, the bulkload algorithm 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 being clustered with the parent node. Unfortunately, these changes break the optimality assurance of the original algorithm. This demotes the Natix algorithm to a heuristic with respect to the minimum number of records generated. It is not clear how the bottom-up algorithm can be modified to address the issues above and still retain global optimality. In particular, whereas sibling clustering is desirable with respect to the number of generated clusters, it increases the search space of possible clusterings. We have not yet been able to find a linear-time algorithm that produces an optimal solution.

Since efficiency is of great importance for document import, we consider a slightly suboptimal clustering acceptable, as it can be done in linear time. The heuristic algorithm explained next generates very good clusterings in all observed cases. In particular, it outperforms the optimal solution without sibling clustering.

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

Figure 4
The beginInternalNode( ) function

```
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
The endInternalNode( ) function

Detailed description of the Natix algorithm

The algorithm maintains a main-memory tree that consists of nodes that have not yet been assigned to a cluster. 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 that 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, that is, the maximal path length from the root node to a leaf node in the document. This property is ensured by keeping, on each level, only as many nodes as fit within a certain configured memory limit, which is an integer multiple of the cluster limit.

The bulkload operation starts with an empty main-memory tree. Every call to the interface functions to construct the document adds a new main-memory node. Whenever the main memory-tree exceeds a memory limit, a cluster of main-memory nodes is transferred to a record on secondary storage.

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

The `beginInternalNode()` function simply adds the new node to the main-memory tree (***Figure 4***). 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 ensure 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, known as *memory limit*, is the cluster limit multiplied by an integer m (*memory factor*). In the section "Evaluation," we show that for memory factor values greater than 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 the stand-alone record header is larger

```
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
The pruneCurrentCluster( ) function

```
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
The clusterChildren( ) function

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.

During pruning of the tree, physical records are created that 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.

We 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. The `findClusterBoundRight()` function 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.

The `findClusterBoundLeft()` function moves further right, starting from the rightmost node of the

new partition. It includes nodes in the interval as long as they satisfy the preceding predicate, and while the closed interval of subtrees bounded by `firstsplit` and `lastsplit` still fits into a physical record.

The `createRecord()` function 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.

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

Memory management

The main-memory representation consists of many small objects, including literals (we should point out that literals are 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, and memory is released for entire subtrees at the same time. These two facts can be exploited in the following memory management technique. The 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. This way, the subtree induced by a node is stored on consecutive blocks. When a subtree's main-memory structure is no longer used, the sequence of blocks that contain nodes only of this subtree can be deallocated in a per-block fashion, without processing the individual nodes on the blocks.

Terminating import operations prematurely

A document import may be terminated prematurely, for example, because XML document validation fails halfway through a document. For such cases, the bulkload interface provides an `abortBulkload()` method. A call to this routine removes both the nodes still in main memory and the partially stored document on disk storage.

Deallocating the nodes in main memory is done in the same way as removing subtrees. However, there are two approaches to removing the on-disk structures, depending on whether Natix recovery code is enabled.

With enabled recovery, a transaction savepoint is created during `beginBulkload()`. Upon `abortBulkload()`, the transaction is rolled back to that savepoint, and removal of the data structures on disk is automatically handled by the recovery subsystem's rollback routines.

Without recovery support, the bulkload component first scans the main-memory structure for proxy nodes and deletes the referenced records, recursively descending into further proxy nodes if present. After the subtree records are removed from disk, the main-memory tree is deallocated.

EVALUATION

We evaluated the performance of the Natix bulkload component and present now the experimental results. These results show the effect of sibling clustering, the scalability of our design with respect to document size, and a comparison of the performance of Natix with other XDSes.

Document collections

Experiments were performed using three document collections. The first was the XMark benchmark,[13] using scaling factors of $n \times 0.2$ with $n \in \{1 \ldots 5\}$. The second was a synthetic document collection generated using the ToXgene data generator.[14] The Document Type Definition (DTD) as well as the generator template file are listed in Reference 15. The smallest document contained 50 employees, 100 students, 10 lectures, and 30 exams. We generated six documents. With each document we quadrupled these numbers, so that the biggest document contained 51,200 employees, 102,400 students, 10,240 lectures and 30,720 exams. This led to document sizes between 59 KB and 43 MB.

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. It was equipped with 512 MB RAM, a Pentium** IV CPU with 2.4 GHz, and an Ultra Wide SCSI hard disk. The operating system was a SUSE** Linux 9.3 with kernel version 2.6.11. Machine `OLD` was used to reproduce the environment from Reference 13 and had 512 MB 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 entire document was used. The times do not include system startup time (about 0.1 seconds), 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,[16] we used the Monet Database Server with the Pathfinder module as publicly distributed (MonetDB 4.8.0, Pathfinder 0.8.0). We present the import times reported by the Monet console.

Algorithms

For Natix, we implemented the bulkload algorithm by using a default value of $m = 5$. A disk page size of 8 KB was used, and the cluster limit was set to 2 KB to avoid fragmentation (see the section "Suitability as bulkload algorithm").

We also implemented a modified variant of the Kundu and Misra algorithm to compare our approach to optimal partitioning without sibling partitions. We had to modify the Kundu and Misra algorithm to incorporate the fact that the weight of a cluster is modified by the additional proxy nodes. This involves three modifications. First, for processing a node, the weight of proxies is added to the node. Second, nodes whose weight is smaller than or equal to a proxy node are 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 in which 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 use 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 that are referenced by a proxy in the parent's cluster (see the `clusterChildren()` function). As the experimental results show, this rarely occurs.

Results

The experimental results cover several aspects of Natix performance: importance of sibling clustering, scalability, comparison with XC,[17] and comparison with other published results.

Table 1 Number of clusters for XMark with a scaling factor 0.2

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

The importance of sibling clustering

The first series of experiments is intended to illustrate the importance of sibling clustering. Hence, we took the XMark document with scaling factor 0.2, producing a document about 20 MB in size, and bulkloaded it using the modified Kundu and Misra algorithm and the Natix algorithm. For the Natix algorithm, we used different values for the m parameter.

The number of clusters generated are shown in *Table 1*. The modified Kundu and Misra algorithm produces about 50 percent 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 had to be created for the Kundu and Misra 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 and Misra algorithm because it degenerates to a nonoptimal single-child clustering.

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.

Table 2 Import times (seconds) for Natix and MonetDB

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

Scalability

In this experiment, we evaluate the scalability of our approach and compare it to the scalability of a nonclustering approach.

We import the two document collections into Natix and MonetDB/Pathfinder.[16] MonetDB is a relational main-memory DBMS that stores XML as binary relations in which the nodes are stored in preorder, that is, in the order delivered by the parser. In such a format, no clustering is required, but only a preorder traversal is supported as an efficient access path, and updates may be costly.

The results from *Table 2* show that the Natix bulkload algorithm exhibits a running time linear to the document size. For the XMark documents, MonetDB is about twice as fast and also scales linearly. For the "uni" documents, the Natix behavior does not change, the scalability and bulkload speed remain similar to the XMark case. MonetDB, however, shows a different behavior 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, its performance comparable with a nonclustering approach.

Table 3 Comparing import times (in seconds): XC Versus Natix

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

Comparison with XC

XC is an XML clustering algorithm developed at the IBM Thomas J. Watson Research Center in Yorktown Heights, New York.[17] Its optimized version of the Lukes algorithm is a workload-directed algorithm that generates good clustering tailored to previously configured workloads. However, it does not have acceptable performance for online bulkloads. We show some of the results for XC 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 2.4 GHz machine NEW. The results show running times for Natix that are faster by about an order of magnitude. This difference is clearly beyond the difference in processor speed. In addition, the XC heuristic algorithm performs only single-child clustering, which is inferior to sibling clustering with respect to the number of clusters, as demonstrated earlier.

Comparison with other published results

There are few published results for bulkload performance of XDS systems. The only comparable results we could find were obtained using the XMark benchmark by Schmidt et al.[13] They compare bulkload performance for an XMark scaling factor of 1 on various anonymous mass-storage systems. We display some of their results in *Table 4*.

We limit our comparison to disk-based systems, omitting their numbers for main-memory-only systems, as we do not know whether the main-memory-only systems perform logging or checkpointing and whether the numbers reflect the corresponding overhead. The remaining systems are relational DBMSes, identified as System A, System B, and System C in this paper. No details about the

Table 4 XML bulkload times for various systems

System	Bulkload Time (seconds)
System A (from Reference 15)	414
System B (from Reference 15)	781
System C (from Reference 15)	548
Natix	215

employed mappings from documents to relations are given, except that Systems A and B do not require a DTD, whereas System C requires that a relational schema be manually generated 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.,[13] except that it has less main memory (512 MB compared to their 1 GB), and a slightly faster processor (600 Mhz compared to their 550 Mhz).

Although Natix outperforms the relational systems by factors between 1.9 and 3.6, little 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.

CONCLUSION

In this paper we discuss the bulkload component of Natix, a module of the Natix XDS that efficiently converts external documents to the Natix storage format.

In our requirements analysis, we argue that a bulkload component for XML must address three important issues. First, the processing of documents must be efficient 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 conversions. Third, the layout for storing the documents on persistent storage devices must be of high quality in this sense: for tree-structured data such as XML, the number of generated clusters should be minimal. Clusters represent subtrees 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 set of document nodes that is connected through parent-child and sibling relationships.

We also evaluate the extent to which a number of existing algorithms fit our requirements. Even the best candidate, the tree-clustering algorithm by Kundu and Misra, fails to address all requirements, in particular because it keeps the entire document in memory and because it does not cluster siblings.

We extend 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 percent fewer clusters than an optimal single-child clustering. The algorithm has linear complexity with respect to the document size and uses main-storage space proportional to the document tree height.

We present experimental results that demonstrate the competitiveness of our bulkload component. Specifically, we show that sibling clustering is superior to single-child clustering and that our algorithm scales linearly with the document size while the multiplying constants are small. Moreover, 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. Finally, 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 plan to improve our heuristics for sibling clustering. We also intend to incorporate information about access patterns into our algorithm without compromising bulkload performance.

CITED REFERENCES

1. 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**, No. 4, 292–314, (2002).
2. K. Beyer, R. J. Cochrane, V. Josifovski, J. Kleewein, G. Lapis, G. Lohman, B. Lyle, F. Ozcan, H. Pirahesh, N. Seemann, T. Truong, B. Van der Linden, B. Vickery,

and C. Zhang, "System RX: One Part Relational, One Part XML," *Proceedings of the ACM SIGMOD Conference* (2005), pp. 347–358.

3. C.-C. Kanne, M. Brantner, and G. Moerkotte, "Cost-Sensitive Reordering of Navigational Primitives," *Proceedings of the ACM SIGMOD Conference* (2005), pp. 742–753.
4. C.-C. Kanne and G. Moerkotte, *Efficient Storage of XML Data*, Technical Report TR-1999-008, Department for Mathematics and Computer Science, University of Mannheim (June 1999).
5. C.-C. Kanne and G. Moerkotte, "Efficient Storage of XML Data," *Proceedings of the 16th International Conference on Data Engineering (ICDE)*, IEEE Computer Society (2000), page 198.
6. M. M. Tsangaris and J. F. Naughton, "On the Performance of Object Clustering Techniques," In *Proceedings of the 1992 ACM SIGMOD International Conference on Management of Data, San Diego, California, June 2–5, 1992*, Michael Stonebraker, Editor, ACM Press (1992), pp. 144–153.
7. J. A. Lukes, "Efficient Algorithm for the Partitioning of Trees," *IBM Journal of Research and Development* **18**, No. 3, 217–224 (1974).
8. R. Bordawekar and O. Shmueli, "Flexible Workload-Aware Clustering of XML Documents," *Database and XML Technologies, Proceedings of Second International XML Database Symposium, XSym 2004*, Lecture Notes in Computer Science 3186, Springer, New York (2004), pp. 204–218.
9. M. Schkolnick, "A Clustering Algorithm for Hierarchical Structures," *ACM Transactions on Database Systems* **2**, No. 1, 27–44 (1977).
10. S. Kundu and J. Misra, "A Linear Tree Partitioning Algorithm," *SIAM Journal on Computing* **6**, No. 1, 151–154 (March 1977).
11. D. Megginson, *SAX: A Simple API for XML*, Technical Report, Megginson Technologies Ltd. (2001).
12. D. Veillard, *The XML C Parser and Toolkit of Gnome* (2002), http://www.xmlsoft.org/index.html.
13. A. Schmidt, F. Waas, M. Kersten, M. J. Carey, I. Manolescu, and R. Busse, "XMark: A Benchmark for XML Data Management," *Proceedings of the 28th VLDB Conference* (2002), pp. 974–985.
14. D. Barbosa, A. Mendelzon, J. Keenleyside, and K. Lyons, "ToXgene: a Template-Based Data Generator for XML," *Proceedings of the ACM SIGMOD Conference* (2002), pp. 616–616.
15. C-C. Kanne and G. Moerkotte, *The Importance of Sibling Clustering for Efficient Bulkload of XML Document Trees*, Technical Report TR-2005-009, Department of Mathematics and Computer Science, University of Mannheim (November 2005).
16. P. A. Boncz, T. Grust, S. Manegold, J. Rittinger, and J. Teubner, *Pathfinder: Relational XQuery over Multi-Gigabyte XML Inputs In Interactive Time*, Technical Report INS-E0503, CWI, Amsterdam, Netherland (March 2005).
17. R. Bordawekar and O. Shmueli, *Flexible Workload-Aware Clustering of XML Documents*, IBM Thomas J. Watson Research Center, Yorktown Heights, NY (May 2004) (available from the author).

Accepted for publication November 8, 2005.
Published online May 3, 2006.

Carl-Christian Kanne
Department for Practical Computer Science III, University of Mannheim, 68131 Mannheim, Germany (cc@informatik.uni-mannheim.de). Dr. Kanne is a researcher at the University of Mannheim. He received a Masters (Diplom) degree in computer science from Rheinisch-Westfälische Technische Hochschule (RWTH) Aachen in 1998, and a Ph.D. degree from the University of Mannheim in 2003. His current work focuses on Natix, a native XML data store developed at the University of Mannheim.

Guido Moerkotte
Department for Practical Computer Science III, University of Mannheim, 68131 Mannheim, Germany (moerkotte@informatik.uni-mannheim.de). Dr. Moerkotte studied computer science at the University of Dortmund, the University of Massachusetts at Amherst, and the University of Karlsruhe. From the University of Karlsruhe he received a diploma in 1987, a doctoral degree in 1989, and his habilitation in 1994. He is currently a full professor at the University of Mannheim. His research interests include all aspects of database management systems. He is especially interested in system design and implementation. He is a co-author of more than 100 publications. ■