RC23839 (W0506-130) June 29, 2005
Computer Science

IBM Research Report

Flexible Workload-directed Clustering of XML Documents

Rajesh Bordawekar, Oded Shmueli
IBM Research Division
Thomas J. Watson Research Center
P.O. Box 704
Yorktown Heights, NY 10598

Research Division
Almaden - Austin - Beijing - Haifa - India - T. J. Watson - Tokyo - Zurich

Flexible Workload-directed Clustering of XML Documents

Rajesh Bordawekar[§] Oded Shmueli[†*]
[§]IBM T. J. Watson Research Center, Yorktown Heights, NY 10598
[†]Computer Science Department, Technion, Haifa 32000, Israel
{bordaw@us.ibm.com oshmu@cs.technion.ac.il}

Abstract

We investigate workload-directed physical data clustering in native XML database and repository systems. We present a practical algorithm for clustering XML documents, called XC, which is based on Lukes' tree partitioning algorithm. XC carefully approximates certain aspects of Lukes' algorithm so as to substantially reduce memory and time usage. XC can operate with varying degrees of precision, even in memory constrained environments. Experimental results indicate that XC is a superior clustering algorithm in terms of partition quality, with only a slight overhead in performance when compared to WDFS, a workload-directed depth-first scan and store scheme. We demonstrate that XC is substantially faster than the exact Lukes' algorithm, with only a minimal loss in clustering quality. Results also indicate that XC can exploit application workload information to generate XML clustering solutions that lead to major reduction in page faults for the workload under consideration.

1 Introduction

Current database and repository systems use two main approaches for storing XML documents. The first approach maps an XML document into one or more relational tables [4, 1]. Stored XML documents are then processed via traditional relational operators. The second approach, *Native XML Storage*, views the document as an XML tree. It partitions the XML tree into distinct records containing disjoint connected subtrees [8, 3, 13]. XML records are then stored in disk pages, either in an unparsed, textual form, or using some internal representation. In this paper, we concentrate on the native XML storage approach.

XML documents are often processed using query languages such as XPath, XSLT, or XQuery [2]. These languages use XPath for traversing *paths* of the abstract XML trees. Unlike relational databases where data is queried using *value-dependant* select-project-join (SPJ) queries, XML processing is dominated by *path-dependant* XPath queries. In practice, such traversals are often aided by path indices that reduce the number of path navigations across stored XML records. However, path indices cannot completely eliminate such path traversals as XML query patterns are often complex and it is impractical to maintain multiple path indices to cover all possible paths of the entire XML document. In reality, disk-resident XPath processors employ a *mixed*, i.e., part navigational, part indexed, processing model.

Physical *co-location* or *clustering* of data items has been extensively used in database systems for exploiting common data access patterns to improve query performance. Data clustering was shown to be particularly effective when the underlying data model is hierarchical in nature and supports queries whose execution path traversals remain relatively stable (e.g., IBM's IMS [11]). In native XML database systems, XPath operations result in navigations across stored XML records,

*Work done while the author was visiting IBM T. J. Watson Research Center.

which are similar to those in hierarchical or object-oriented databases. Clustering techniques could also be applied for storing XML documents in native XML databases, where *related* XML nodes can be clustered and stored in the same disk page. Here, two XML nodes are *related* if they are connected via a link and examining one of them is likely to soon lead to examining the other. As disk pages have limited capacity, not all related XML nodes can fit in a single disk page. Therefore, one needs to decide how to assign related XML nodes to disk pages.

1.1 Problem Statement

This paper investigates the following XML clustering problem: Given *a static* XML document[1], the mixed XPath processing model, and associated navigational workload information, identify related XML nodes and efficiently cluster them to disk pages.

1.2 Proposed Solution

We formulate the XML clustering problem as a tree partitioning problem. The tree to be partitioned is a *clustering tree*, namely an XML tree augmented with node and edge weights. Roughly, the edge weights model the XML navigational behavior (higher edge weights mean that the connected XML nodes are more strongly related). Node weights are the (text) sizes of the XML nodes. The problem is to partition the set of nodes of the clustering tree into node-disjoint subsets (called *clusters*) so that each cluster fits into a disk page, and the total of the intra-cluster edges' weights (called the partition's *value*) is maximized. Intuitively, a higher value partition implies fewer disk accesses.

Graph partitioning techniques have been traditionally used for clustering related objects in OODBs [14, 6]. These techniques are not suitable for partitioning trees as they cannot exploit structural aspects of trees for making intelligent partitioning decisions. Furthermore, these techniques are computationally expensive. Amongst the existing deterministic tree partitioning algorithms [10, 7], Lukes' bottom-up dynamic programming solution is the most efficient and practical. At first, it appears obvious to directly use Lukes' algorithm for the XML clustering problem. However, as we demonstrate in Section 5, Lukes' algorithm exhibits excessive memory and time requirements, even while partitioning small XML documents. For example, it took more than 7 hours to partition a 110 KB XMark document. This makes Lukes' algorithm unsuitable for clustering XML documents, even in offline scenarios where execution time is not the main concern.

In this paper, we propose XC, a new tree partitioning algorithm that can operate with varying degrees of precision under space and time constraints. We use this algorithm to partition the clustering tree of an XML document, where edge weights are generated using either statistics, XML document structure or navigational workload. XC is based on LUKES, Lukes' tree partitioning algorithm. While LUKES is an *exact* algorithm, XC is an *approximate* algorithm.

1.3 Main Contributions

The main contribution of this work is a practical solution to the XML clustering problem using a tree partitioning approach which uses XML navigational behavior to direct its partitioning decisions.

A key contribution of this work is XC, which has the following advantages over LUKES:

- First, XC exploits intrinsic structural regularities in Lukes' dynamic programming algorithm, namely, *ready* clusters (Section 4.1.1), to improve its memory consumption and running time,

[1] We don't consider inter-document clustering.

without affecting its precision. With this optimization, XC acts as an efficient version of the precise LUKES algorithm.

- Second, XC implements a parametric approximation of the dynamic programming procedure that allows it to tradeoff precision for time. This enables XC to exhibit linear-time behavior without significant degradation in quality over the precise solution.

- Third, under memory constraints, XC *continuously*, *on-the-fly* adapts its precision by selectively eliminating dynamic programming options such that the memory limitations are not violated. This allows XC to tradeoff precision for memory.

These optimization techniques are orthogonal and can be used either independently or in conjuction with each other. These capabilities make XC a flexible tree partitioning algorithm that can partition a tree with different precision depending on the memory and time availability.

The other main contribution of the paper is the experimental validation of XC using a prototype XML clustering system, **XCS**. Within a single pass, this system scans an XML document, assigns edge weights according to the workload, partitions the XML document into node-disjoint clusters, and then assigns clusters to disk pages. In **XCS**, edge weights of a clustering tree are computed by a hypothetical XPath processor for a workload consisting of XPath queries. We used **XCS** to compare XC against an alternative tree partitioning scheme, workload-directed depth-first scan & storage (WDFS). WDFS is a natural clustering algorithm that traverses the clustering tree in depth-first manner and uses workload information to decide which children nodes should be stored with their parent [14]. The experiments evaluate the quality of tree partitioning and measure the number of page faults for various XPath query workloads under different paging scenarios using the mixed XPath processing model with prefix path indexes.

Experimental results demonstrate that XC computed *approximate* partitions that were closer in value to the *optimal* partition computed by the precise Lukes' algorithm, at a fraction of the runtime costs. For example, using the least precision, XC partitions the same 110 KB XMark document in 432 ms with an *approximate* partition of value 20244, whereas LUKES took more than 7 hours to compute the *optimal* partition with value 20460! The difference in values between the optimal and approximate partitions was extremely small ($< 2\%$). Similarly, XC computed higher valued partitions than WDFS, with a slight overhead in performance. When clustering using workloads consisting of XPath queries, XC was able to compute a partition that matched the XPath navigation behavior. The physical layout generated by XC caused fewer page faults, sometimes by more than an order of magnitude, than WDFS. For example, in one case, the number of page faults reduced from 75 to 7 when the document was partitioned using XC instead of WDFS. Experimental results also show that XC's precision had very minor impact on the final partition value and the number of page faults. Coupled with these advantages, XC's ability to use different degrees of approximation in practically linear time and its ability to execute under severe memory constraints make it a highly suitable candidate for enabling workload-directed XML clustering for both online and offline scenarios.

This work is the first step towards a comprehensive clustering solution for storing XML documents in native XML databases. Currently, we are extending XC for supporting dynamic scenarios where the XML workload can change and/or the XML documents are updated. We are also planning to apply XC's clustering techniques to complex XPath workloads, such as XSLT templates or XQuery queries.

1.4 Organization

The paper is organized as follows. The tree partitioning problem is formalized in Section 2. LUKES is reviewed in Section 3, and in Section 4, we present XC. The experiments are detailed in Section 5. Related work is summarized in Section 6. Section 7 presents conclusions and future work directions. Appendix A presents a small example illustrating the operations of LUKES. Appendix B presents XC's pseudo-code.

2 The Tree Partitioning Problem

Consider a rooted tree $T = (V, E)$, where V is a set of *nodes* and $E \subseteq V \times V$ is a set of *edges*. A *cluster over* T is a non-empty subset of V. When no confusion arises, we simply use the term *cluster*. A *partition of* T, P^T, is a set of pair-wise disjoint clusters over T whose union equals V, that is $P^T = \{c_1, \ldots, c_k\}$, $k \geq 1$, such that $\bigcup_{i=1}^{k} c_i = V$, and $c_i \bigcap c_j = \emptyset$, for all $i \neq j$.

Each node i of T is associated with a non-negative integer *weight*, w_i. Each edge (i, j) of T is associated with a non-negative integer *value*, v_{ij}. The *size* of a cluster c, $size(c)$, is the sum of the weights of its nodes; formally, $size(c) = \Sigma_{i \in c}\ w_i$. The *value* of a cluster c, $value(c)$, is the sum of the values of its edges; formally, $value(c) = \Sigma_{(i,j) \in E\ \wedge i \in c\ \wedge j \in c}\ \ v_{ij}$. The *value* of a partition P^T, $value(P^T)$, is the sum of the values of its clusters; formally, $value(P^T) = \Sigma_{c \in P^T}\ value(c)$.

Let W, the *cluster weight bound*, be a positive integer. The *tree partitioning problem* is formulated thus. Find a highest value partition, P^T_{opt}, among all the possible partitions of T, such that the size of each cluster in P^T_{opt} does not exceed W. P^T_{opt} is said to be an *optimal partition* of T.[2] So, P^T_{opt}=$\{c_1, \ldots, c_k\}$ such that $size(c_i) \leq W$, for $i = 1, \ldots, k$, and $value(P^T_{opt}) = Max\{value(P^T)$ $|$ P^T *is partition of* T*and* $\forall c \in P^T$, $size(c) \leq W\}$.

3 Lukes' Tree Partitioning Algorithm (LUKES)

Lukes' algorithm, LUKES, addresses the tree partitioning problem. LUKES operates on a tree in a bottom-up manner; it considers a node only after all the node's children have been considered. Consider a partition $P^{T'}$ of a subtree T' of T rooted at node x. The unique cluster in $P^{T'}$ which contains x is called the *pivot cluster* of $P^{T'}$. The weight of a partition $P^{T'}$ is the size of its pivot cluster. LUKES uses dynamic programming on the partition weights as follows: For each subtree, say rooted at a node x, and for *e*ach feasible total cluster size U (i.e., $w_x \leq U \leq W$), LUKES constructs, if possible, an optimal subtree partition in which the pivot cluster is of size U. So, LUKES associates with node x a *set* of partitions, each optimal under the constraint that the pivot cluster size is U. When considering a node x, the partitions that are associated with each child node of x are used to update the collection of partitions, one per each feasible total cluster size, for x. Once the tree root node is processed, the final result, P^T_{opt}, is the highest value partition associated with the root; as Lukes showed, P^T_{opt} has the maximum partition value among all possible partitions of the tree.

LUKES: Lukes' algorithm consists of the following steps:

1. For each leaf node u with weight $w = w_u$, form the partition P_w, with value 0, consisting of a single pivot cluster containing the node u ($P_w = \{\{u\}\}$). Mark this partition as the

[2] In general, there may be more than one optimal partition.

optimal partition for u. For all internal non-leaf nodes v, form similar initial partitions, each containing a single cluster with value 0 and weight $w = w_v$.

2. Arbitrarily choose some node x with weight $w = w_x$, such that all of x's children are now leaf nodes. For the subtree rooted at node x, compute for $w' = w, w+1, \cdots, W$, the optimal partition, if one exists, in which the cluster containing x (i.e., the pivot) has weight w' (recall that W is the weight bound). To find these optimal partitions, perform the following steps:

   (a) Let node i be the first child of node x.

   (b) For each partition P of node x and for each partition Q of node i, generate intermediate partitions I_1 and I_2 as follows:

      i. Let c_x be the pivot cluster of P (containing x) and let c_i be the pivot cluster of Q (containing i).

      ii. **Merging:** If $size(c_x) + size(c_i) \leq W$, then create a new cluster, $c_m = c_x \bigcup c_i$, i.e., merge the two respective pivot clusters. Create a new partition, $I_1 = \{c_m\} \bigcup (P - \{c_x\}) \bigcup (Q - \{c_i\})$, i.e., by "gluing" c_m and the remaining clusters from partitions P and Q.

      iii. **Concatenation:** If Q is marked as the optimal partition for node i, create a new partition $I_2 = P \bigcup Q$, i.e., concatenate clusters from partitions P and Q to form a new partition. Else, $I_2 = \{\{\ \}\}$, with value 0.

   (c) Update the intermediate partitions associated with node x. Let a (resp., b) be the weight of the cluster containing node x in I_1 (resp., node i in I_2). If the current partition of weight a associated with x, P_a, is such that $value(P_a) < value(I_1)$ then discard P_a and replace it with I_1. Then, if the current partition of weight b associated with x, P_b, is such that $value(P_b) < value(I_2)$ then discard P_b and replace it with I_2. If i is the last child of node x, proceed to Step 3, otherwise, let i be the next child of x and go to Step 2b.

3. Mark a partition, with the maximum value among the intermediate partitions associated with node x, as the *optimal* partition, P^x_{opt}, for the subtree with root x. Delete the children of node x from tree T. Node x is now a leaf. If node x is the root then P^x_{opt} is the computed *optimal* partition of T. Otherwise, go to Step 2.

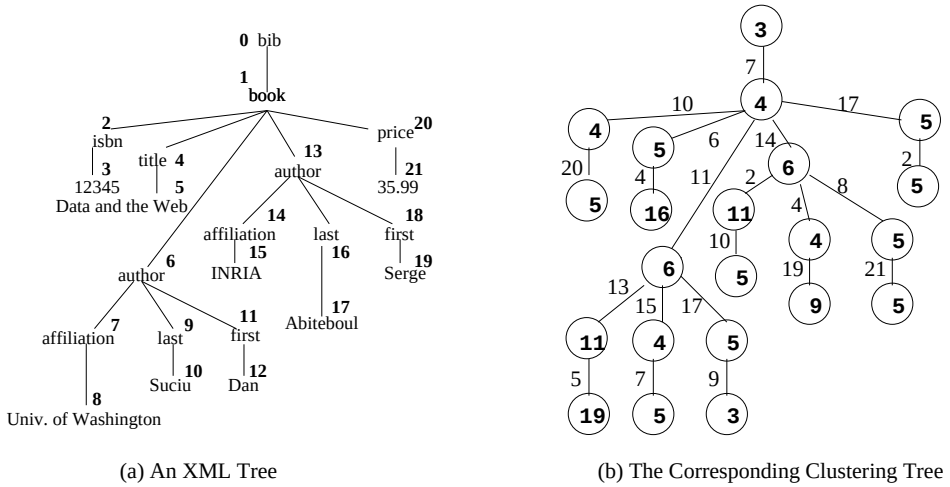
The inner loop (Step 2) is executed for every internal node in the tree. This loop involves combining partitions of the internal node with the partitions of its children (Step 2b). In the worst case, a parent may possess W intermediate partitions after processing its first child (Step 2c); these W partitions can then be combined with W partitions of every remaining child. For an n-node tree, Step 2b can be invoked at most $n-1$ times, leading to an $O(n\,W^2)$ running time.

4 XC: An Approximate Tree Clustering Algorithm

We now present a new approximate tree partitioning algorithm, XC, tailored for clustering large XML documents. We first consider the case of unlimited memory and discuss differences between Lukes' algorithm (LUKES) and XC. We then describe how XC operates subject to application-specified memory constraints. Appendix B presents an outline of the XC.

4.1 Improvements upon LUKES

The key differences between the two algorithms lie in the way intermediate clusters and partitions are generated and used. Both LUKES and XC recursively partition a *clustering tree*. Recall that the *clustering tree* is the XML tree augmented with node and edge weights, both integers. Node weights are the (text) sizes of the XML nodes. Edge weights model the importance of putting the nodes into the same cluster. Once a subtree of the clustering tree is completely partitioned, we call it a *processed subtree* (and its root a *processed node*). For the purpose of discussion, we use the XML tree illustrated in Figure 1(a). Figure 1(b) presents the corresponding clustering tree (with node and edge weights).



(a) An XML Tree

(b) The Corresponding Clustering Tree

Figure 1: XML and Clustering Trees. Node weights match the text sizes of the XML nodes.

4.1.1 Identification of Ready Clusters

While partitioning a clustering tree, LUKES creates new partitions by merging and concatenating clusters from partitions of its processed subtrees. After Step 3 of LUKES is executed, a processed subtree gets partitioned into possibly W partitions, where W is the weight bound. These intermediate partitions are then used for clustering the subtree rooted at the parent of the processed node. As a result, LUKES retains and propagates a larger number of partitions and clusters until the final successful partition is computed, leading to excessive memory usage and runtime costs. XC addresses this problem by aggressively detecting those clusters that are **guaranteed** to be part of *any* final partition. Such clusters are called, *ready clusters*. A cluster in a partition associated with a processed node x is *ready* iff it satisfies the following conditions:

1. It does not contain node x and hence it cannot be involved in any future cluster merge operations, and

2. It is a member of **every** intermediate partition, including the *optimal* partition, associated with node x.

According to condition (1), since the ready cluster is not a pivot cluster of any intermediate partition of a processed subtree, it can not be modified via the merging operation (Step 2b:ii). Condition (2) guarantees that every future generated partition using the intermediate partitions will contain the ready cluster. Hence, the ready cluster can be safely removed from the intermediate

partitions of the processed subtree without affecting the correctness of the final solution. XC detects ready clusters as soon as the subtree rooted at node x is processed and forwards it (along with its XML data) for further processing (e.g., the page assignment sub-system, see Section 5.1, Figure 2). Once the page assignment sub-system assigns a ready cluster to a page, the XML nodes, contained in the cluster, can be deleted along with their XML data. This can lead to significant reduction in memory usage and consequently improves the running time. With only this optimization, XC behaves like an *optimized* version of the *exact* LUKES.

4.1.2 Approximate Dynamic Programming using Weight Intervals

Since LUKES is an exact algorithm, it explores *all* possible relevant alternatives before producing an optimal partition. For a tree with n nodes, for every node n', with weight $w \geq 1$, LUKES examines all possible partitions with weights from w to the weight bound, W. This results in $O(nW)$ space and $O(nW^2)$ time complexities. This is prohibitive for large XML files where values for n can be in the millions (of XML nodes) and W in the thousands (of bytes). To address this problem, XC partitions the $(1, W)$ weight range into equal sized *weight intervals* and retains only one partition for each weight interval. The number of the weight intervals is specified by the application using the *chunk_size* parameter. The value of *chunk_size* can vary from 1 to W and is a divisor of W. The chosen partition of each weight interval has the *maximum value* among the partitions whose weights fall in that interval. So, for XC, following internal node processing, (LUKES, Step 2) there are at most $W_c = \frac{W}{chunk\_size}$ intermediate partitions associated with the node. Similarly, the inner loop (LUKES, Step 2b) iterates over the weight intervals from 1 to W_c. These modifications reduce the space complexity to $O(n\ W_c)$ and time complexity to $O(n\ W_c^2)$. This weight interval approximation technique is used only for *assigning* intermediate partitions in the proper interval. The decision whether to merge clusters (LUKES, Step 2b) is still based on *actual* cluster weights (rather than the interval to which the partition containing the cluster belongs). For example, consider the invocation of LUKES on the clustering tree consisting of nodes 2 and 3. LUKES creates two intermediate partitions. The first, p_4^2, with weight 4 and value 0, is created by concatenating clusters (2) and (3). The partition p_9^2, with weight 9 and value 20, is created by merging clusters (2) and (3). If $W = 10$ and $chunk\_size = 10$ then $W_c = 1$ and both p_4^2 and p_9^2 fall in the single weight interval, so only the higher valued p_9^2 is retained. If $W = 10$ and $chunk\_size = 5$ then $W_c = 2$, and both partitions are retained.

It is instructive to examine the boundary cases. When $chunk\_size = 1$, XC operates exactly as LUKES; it also finds an optimal partition. When $chunk\_size = W$, only one intermediate partition is produced per each node. In other words, when $chunk\_size = W$, XC operates as a simple greedy algorithm. In general, by changing the chunk_size, one can control the precision of XC's approximate tree partitioning algorithm.

4.2 Handling Memory Constraints

In spite of the memory reduction techniques employed by XC, its memory consumption can still be very large, particularly for large XML documents and for smaller chunk_size values. We now describe the technique employed by XC which uses *ready sub-partitions*, to work effectively subject to a memory limit. A *ready sub-partition* is a partition that is associated with the root of an already processed subtree (of the clustering tree), which is a *subset* of the computed best approximate partition for the *whole* clustering tree. If one could identify a ready sub-partition, then the sub-partition's clusters can be forwarded for further processing and the memory of associated data structures could be reclaimed immediately. Observe that the elimination of partitions also

reduces the number of options for future dynamic programming considerations; further reducing the computation and memory usage.

In general, it is difficult to identify a ready sub-partition at a processed node x (a sufficient condition is that in all the partitions associated with x, the pivot cluster has weight W). Instead, when memory limits are approached, we *on-the-fly declare* the highest valued sub-partitions as ready. This approach sharply reduces memory consumption. However, it may affect accuracy of the final solution since there may be some clusters in a ready sub-partition that could have been used in future cluster merge operations (which in turn would have increased the value of the final partition). Therefore, it is important to only declare ready partitions with the highest value such that the final result quality (i.e., value) is not significantly affected.

We now explain the mechanism employed by XC to choose a partition as the next ready sub-partition. Once a subtree is processed, its intermediate partitions are retained until its parent node is processed. For example, for the XML tree presented in Figure 1(a), intermediate partitions for nodes 7, 9, and 11 are retained until the subtree rooted at node 6 is processed. XC maintains a bounded-length (a parameter k, e.g., $k = 8$) list, HL, of the current up to k highest value partitions that are associated with processed nodes (i.e., roots of processed subtrees whose parents are not yet processed). Given a memory limit M, XC computes *high* and *low* "water marks" for managing memory usage (e.g., $low = 0.7M < high = 0.9M < M$). When memory usage crosses the high water mark, corrective actions are triggered. Once it reaches the low water mark, normal operation resumes.

Upon crossing the high water mark, XC chooses a partition, P, with the highest value from HL. XC then marks this partition as a ready sub-partition, forwards its clusters to the page assignment sub-system, and then reclaims the memory of the associated data structures. Observe that P will also form a subset of the final result partition. XC then removes the XML node v_P with which P is associated (i.e., a root of a processed subtree) and discards v_P's link to its parent. This process of eliminating highest value partitions from HL proceeds *continuously* until either the memory usage falls below the low water mark or the optimal partition list is empty. If the memory-usage violation persists and the optimal partition list is empty, then, as soon as any subtree is processed, its optimal partition is immediately marked as ready and is eliminated, in the process described above.

5 Experimental Evaluation

We ran XC with different degrees of precision and compared it against WDFS, a workload-directed depth-first scan and store scheme [14]. WDFS is a natural clustering algorithm that scans an XML document in depth-first manner. In a greedy fashion, it produces XML nodes and uses the workload information (i.e., the edge weights) to store nodes connected by edges with higher weights in the same cluster. The cluster is mapped into disk pages as soon as it is full. A major advantage of WDFS is that it places together XML nodes whose processing completes in temporal closeness. Another advantage of WDFS is that it is ideal for implementation in an online single-pass environment.

For experimental purposes, we used several XML documents from the University of Washington XML repository[3]. We ran our experiments using the prototype XML clustering system, **XCS**. All experiments were run on an x86-based Linux machine with 512 MB main memory and 40 GB disk space. Our implementions of XC and WDFS employ an incremental reference-counting garbage collector to aggressively detect and collect dead objects. Therefore, we used the maximum memory usage during the program execution as a metric of memory consumption. In addition, we used the

[3] `www.cs.washington.edu/research/xmldatasets/`

final partition value, execution times, and the number of page faults during XPath query execution as the main metrics of performance.

5.1 XCS: A Prototype XML Clustering System

Figure 2 illustrates the architecture of **XCS**, a prototype XML clustering system that scans an XML document, partitions it into clusters and assigns the clusters to (disk) pages, all in a single pass. It consists of three distinct sub-systems: edge-weight assigner, tree partitioner and page assigner.

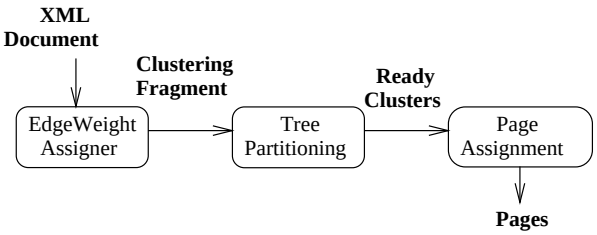


Figure 2: **XCS** Architecture.

5.1.1 Edge-weight Assigner

XCS's edge-weight assigner uses application workload information to assign weights to the clustering tree edges. Currently, the workload information consists of XPath queries. Additional information such as the frequency or importance of specific XPath queries is also provided. In a real-world application, the workload may contain approximations of the actual queries; for example, a parameterized query (i.e., one containing a parameter whose actual values will only be known at runtime when the query is executed) may be approximated with one accessing all instances. **XCS**'s edge-weight assigner uses a simulator that mimics future executions, of a hypothetical XPath processor, for the XPath queries in the workload (this simulator can be replaced by any streaming XPath processor). Currently, the simulator supports the following types of XPath queries:

1. Simple path queries, with the XPath `child` as the only axis, starting from the document root.
2. XPath Descendant-or-self ("//") queries.
3. Queries with positional and path predicates.

Additionally, the simulator mimics executions that use a path index on a query's sub-path. The simulator is invoked by appropriate SAX events. The simulator identifies the XML nodes and the connecting parent-child edges that would be traversed by the hypothetical XPath processor during its simulated execution. To this end, the simulator uses a deterministic automaton that makes state transitions, while traversing an XML link, depending on the target node content. Depending on (1) relative importance/frequency of the XPath query, (2) the axis used in the XPath query, and (3) the plan of execution (i.e., with or without using a path index), the corresponding clustering tree edge is assigned a weight increment. The overall weight awarded to a clustering tree edge is the sum of the weight increments due to the workload queries.

Intuitively, the assigned weight models the fact that the nodes connected by that edge are traversed in (temporal) succession by the XPath processor, and using that edge. The higher is the edge weight, higher is the traversal affinity between the connected nodes. This affinity stems from being traversed often and/or by "important" workload queries. Note that, an edge traversal by

a single "important" query can generate a weight increment that is greater than the cumulative weight increment generated by traversals by several less important queries.

5.1.2 Tree Partitioner and Page Assigner

Once the edge-weight assigner assigns the edge weights, the memory resident portion of the clustering tree is handled by the tree partitioning system (which can use either XC or WDFS). As soon as the tree partitioner identifies clusters suitable for storage, such clusters are forwarded to the page-assignment sub-system which maps them to pages. When XC is used as the tree partitioner, the page assignment sub-system uses an online bin-packing algorithm that makes page assignment decisions solely on the basis of weights of the currently available ready clusters (Section 4.1.1). The WDFS uses a greedy strategy to map clusters to pages.

5.2 Evaluation of XC

We first investigate the effect of chunk_size on the quality (in terms of the partition value) and performance of XC. Recall that XC uses dynamic programming over weight intervals rather than over the entire weight range (see Section 4.1). In XC, the weight interval is determined by the chunk_size parameter (which determines the number of dynamic programming intervals). Table 1 illustrates the effects of the chunk_size parameter on the final partition value and the total execution time. In this experiment, an XML file, `xmark.xml` of size 110 KB, and containing 2647 XML nodes, is clustered with a weight bound, W=1 KB. The chunk_size, c, is varied from 1024 to 1. Clustering tree edges are randomly assigned weights. As illustrated in Table 1, as chunk_size decreases, the value of the final partition solution increases. As the chunk_size decreases, the number of weight intervals used, $\frac{W}{c}$, increases and XC becomes more precise. However, as XC's precision increases, the amount of computation grows (recall that the time complexity of XC is $O(n\ (\frac{W}{c})^2)$), leading to an increase in memory usage and running time. When the chunk_size is 1, XC behaved like the *exact* LUKES, and the required running time was more than 7 hours. The difference in solution values for the chunk_sizes of 1024 and 1, is very small ($< 2\%$), but the execution times varied drastically (432 ms. for 1024 vs. more than 7 hours for 1). This indicates that XC's strategy of dynamic programming over weight intervals and retaining the highest valued partition per weight interval, works well in practice. As the results demonstrate, XC with chunk_size of 1 (equivalently, LUKES), exhibits unrealistic memory and runtime requirements and for varying chunk_sizes, XC obtains good solutions with reasonable memory and runtime usage.

We also measured the impact of identifying ready clusters on the quality of results. We clustered 5 XML documents with and without using the ready clusters. The edges were assigned random weights. As Table 2 illustrates, for all 5 documents, the memory usage increased when the ready clusters were not identified. In many cases, that resulted in a significant degradation in the overall running time. Ready cluster identification reduces the number of data structures that need to be retained and propagated during the clustering process. It doesn't reduce the number of iterations (which is determined by the chunk_size) but it reduces the amount of work per iteration as the amount of data to be handled decreases, and improves overall memory behavior by reducing memory usage and memory footprints of key data structures. It should be noted that for all cases, regardless of the ready cluster identification, the final solution value was the same.

We also ran an experiment where we clustered an XML document using a clustering tree that didn't store the XML data values. We found that the overall running time was similar to when the clustering tree stored the data values. As stated earlier, XC uses an incremental garbage collector to detect and remove dead objects quickly. This, along with the ready cluster identification, makes

Table 1: Effect of Dynamic Programming over Weight Intervals on XC using xmark.xml of size 110 KB. The weight bound was 1 KB and the chunk_size is varied from 1024 to 1. When the chunk_size was 1, XC acted as an optimized version of LUKES.

chunk_size	Value	Memory Usage (bytes)	Time (ms)
1024	20244	49904	432
512	20355	62581	423
256	20430	85603	590
128	20442	116393	1171
64	20460	176740	3040
32	20460	289407	10687
16	20460	523074	44974
8	20460	1012078	195832
4	20460	1910695	960775
1 (LUKES)	20460	7625395	28610786

Table 2: Effect of ready clusters on XC's execution time. 5 XML documents were clustered with (RDY) and without (NRDY) using the ready cluster identification.

Document size (bytes)	Memory (bytes)		Time (ms)	
	RDY	NRDY	RDY	NRDY
xmark 1.1 MB	612956	2133569	3382	3654
mondial 1.97 MB	2227547	6059854	41107	156919
SigmodRecord 478 KB	674173	1182802	3475	4034
uwm 2.33 MB	441110	811964	11204	12577
orders 5.37 MB	1195171	13393376	41242	716419
partsupp 2.24 MB	561892	5198494	13276	96412

Table 3: Evaluating Memory-Constrained Execution of XC using chunk_size of 4 KB on mondial.xml. As the available memory is reduced, maximum memory utilization and running time improve without significant changing the final solution value.

Memory Limit (bytes)	Memory Usage (bytes)	Value	Time (ms)
∞	1425850	8388200	32511
1000000	900576	8353100	37305
500000	452095	8207900	32070
100000	91264	8138600	17794
50000	47247	7996000	13164
30000	25815	7817300	9049

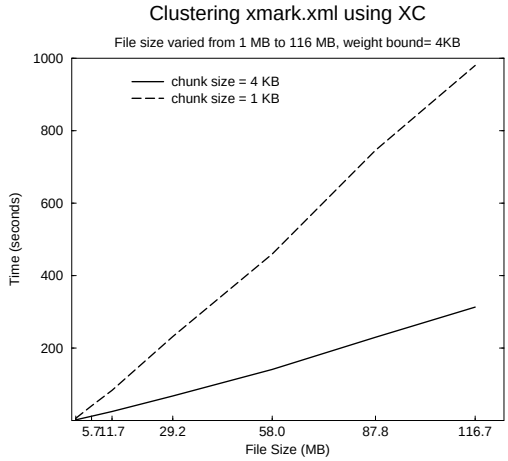


Figure 3: Nearly Linear behavior of XC as the XMark document size is increased. The weight bound was 4 KB.

the clustering application computationally bound. Hence, once the ready cluster identification is used, the number of iterations has the most impact on the overall running time.

Table 3 presents the behavior of XC under various specified memory usage limits. We clustered an XML file, `mondial.xml`, of size 1.97 MB and containing 154855 nodes, using random edge weights. We used the weight bound of 4 KB and ran the experiment, first using a chunk_size of 4096 and without any memory limits. We then repeated the experiment with smaller memory limits (we varied the amount of available memory from 1 MB to 30 KB). The results demonstrate that XC performs well even under extreme memory constraints. Furthermore, as the amount of available memory is reduced, XC degrades gracefully (values of the resultant partitions for the two extreme cases only differ from the best approximate solution by 6%). With tighter memory limits, execution time drops from 32.5 seconds to 9.049 seconds.

Figure 3 presents the behavior of XC as the size of the XML file is increased (which, in this case, implies that the number of input XML nodes is also increased). For this experiment, we used the XMark XML document generation toolkit [12] to generate xmark documents of various sizes (26 KB to 116 MB). For this experiment, we assigned edge weights depending on the *types* of the connected nodes. An element-element edge was assigned weight 3, an element-attribute edge was assigned weight 5, and an element-text edge was assigned weight 9. Each xmark document was

Table 4: Partitioning mondial.xml using workload-based edge weights. XC computes better partitions than WDFS, but at a higher runtime cost.

chunk_size (bytes)	Value	Time (ms)
WDFS		
4096	279000	11540
XC		
4096	410500	15233
1024	412800	17608
512	415600	21124
256	423700	30640
128	423800	53818
64	432900	132149
32	423900	534455

Table 5: Comparing resultant value and time WDFS and XC algorithms for 5 XML documents (weight bound and chunk_size are 4 KB). XC requires more time but generates a better quality partition (higher value) than WDFS.

Document	Value		Time (ms)	
	WDFS	XC	WDFS	XC
xmark	201158	211939	1913	2365
mondial	971513	993988	12486	22687
SigmodRecord	157607	163603	1770	2820
uwm	556935	567444	7605	6779
orders	1600458	1623266	16731	18864
partsupp	473046	481895	4118	6544

clustered using a weight bound of W=4 KB and chunk_sizes of 4 KB and 1 KB. For clarity, Figure 3 presents running times for xmark documents of sizes that vary from 1 MB to 110 MB. As Figure 3 illustrates, XC's runtime is nearly linear in the size of the file.

These experiments demonstrate the XC is a flexible tree partitioning algorithm that can cluster XML documents using different precision and under varying memory limits, in nearly linear time.

The next experiment compares the effectiveness of XC with that of another tree partitioning algorithm, WDFS. We first describe clustering results obtained based on workload-based edge weight assignment. In this experiment, we consider the extreme case in which the workload consists of a single XPath query on `mondial.xml`:
`/mondial/country/province/city`. We considered the execution of this query by a hypothetical XPath processor. The edges of the XML tree that would be traversed during such an execution were assigned weight 100. The document was then partitioned by WDFS and XC using various chunk_sizes. The results are illustrated in Table 4. We can see a clear advantage for XC over WDFS in terms of partition values.

Table 5 presents results of clustering five different XML documents using XC and WDFS. For this experiment, we assigned edge weights based on connected nodes types, in a manner identical to that of the experiment of Figure 3. Note that this setup is skewed against XC as all edges are

assigned weights and the edge weights don't vary significantly (an edge weight can be 3, 5 or 9). We used a weight bound of 4 KB and executed XC with a chunk_size of 4 KB and with infinite memory. As indicated in Table 5, for all documents, XC generated better clustering. However, in many cases, the time required by XC was greater than the time used by WDFS. Even in the most beneficial configuration (weight bound equals chunk_size), XC performs more work than WDFS and uses more memory. This causes XC to be slower than WDFS.

5.3 Evaluating using the Prototype Clustering System, XCS

While the previous experiments examine XC's usefulness as a clustering algorithm which produces high value partitions in reasonable space and time, this sub-section investigates the usefulness of the resulting clustering for applications. To this end, we tested how the resulting page layouts affect query processing.

The testing was performed in the context of the prototype clustering system, **XCS** (Section 5.1). For these tests, we considered the following query workload on `mondial.xml`, consisting of four XPath queries:

(Q1) `/mondial/country/province[2]/city` 150
(Q2) `/mondial/country[//enthicgroups]` 100
(Q3) `/mondial/country/province/city[longitude or lattitude]` 150
(Q4) `/mondial/organization/members` 100

Queries Q1 and Q3 were assigned the weight, 150, and queries Q2 and Q4 were assigned the weight, 100. Queries Q1 and Q2 were executed using a path index on `/mondial/country` and the query Q3 was executed using a path index on `/mondial/country/province`. We clustered `mondial.xml` in **XCS** using XC and WDFS as the tree partitioning algorithms. We first used the combined workload consisting of 4 queries and then repeated the experiments for individual workloads consisting of a single query. We used a weight bound of 4 KB for all experiments and for XC, we used a chunk_size of 1 KB.

We simulated the execution of each query by a hypothetical XPath processor under different clustering scenarios. We generated traces of XML nodes traversed while executing the XPath queries with and without path indices. These traces, along with the page layouts for different clusterings, were used for evaluating the paging behavior. We modified two important parameters to vary the paging scenarios. The first, *buffer size* is the number of disk pages that can be held in memory. The second, *prefetch quantity*, is the number of contiguous disk pages that are fetched in a single disk access (starting at the page addressed and including consecutive pages). The first scenario used a buffer size of 1 page with a 1 page prefetch (denoted by 1/1); the second scenario considered infinite buffer size (i.e., a fetched page is retained in memory for the duration of the execution) and 1 page prefetch (denoted by Inf/1); the final scenario simulated a realistic situation of using a buffer with 64 pages with a 2 page prefetch (denoted by 64/2). This scenario used a simple LRU scheme for discarding in-memory pages. In all cases, we assumed a page size of 4 KB.

Figure 4 represents the page fault measurements for WDFS- and XC-based clustering using combined (namely, the clustering was based on all four queries) and individual (namely, the clustering was based on a single query) workloads. As demonstrated by the results, in all cases, XC caused fewer page faults than WDFS. While WDFS was able to exploit local parent-child traversal affinities, XC was able capture and exploit traversal affinities over *paths* of the abstract XML tree. Although the best paging performance was observed when the document was clustered solely based on the single workload query, the paging performance of individual queries didn't degrade substantially

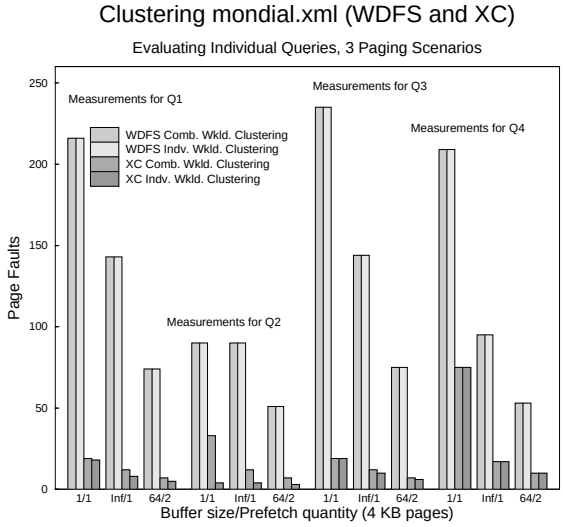


Figure 4: Page faults for WDFS- and XC-based Clustering for combined and individual query workloads. The weight bound was 4 KB and the chunk_size was 1 KB.

when the document was clustered using the combined query workload. The biggest gain was observed for the query, Q2, which was assigned the weight 100. When the combined workload was used, XC generated a clustering that favored the queries Q1 and Q2, which had heavier weights. In contrast, query Q4 exhibited no improvement when the individual query workload was used. Query Q4 did not traverse the same region as the remaining three queries and hence, it solely determined the partitioning of its region, even under the combined workload. This result demonstrated that XC was able to compute the final solution that matched both the *combined* traversal pattern and *individual* traversal patterns.

We also studied the impact of XC's precision on the clustering behavior. We clustered `mondial.xml` using the combined workload with the weight bound, W= 4 KB, and reran the experiment by varying the chunk_size from 4096 to 32. We found that 32 was the smallest chunk_size for `mondial.xml` for which XC could compute the final solution in a reasonable amount of time. Figure 5 shows the impact of XC's chunk_size on the number of page faults for individual queries and Figures 6(a) and (b) illustrate the impact of XC's chunk_size on the final partition value and the overall execution time.

As Figure 5 illustrates, for all queries, as the chunk_size was decreased from 4096 to 32, the number of page faults didn't change significantly (Figure 5). We believe the small fluctuations observed in the number of page faults are artifacts of the online bin-packing algorithm used by **XCS**. As XC's precision is increased, it computes larger clusters, which increases the page fragmentation, leading to an increase in the number of page faults for higher precision. As Figure 6(a) shows, apart from the combined and individual Q4 workloads, the chunk_size had no effect on the final partition value. The query Q4, `/mondial/organization/members`, traverses a region of the XML tree that is closer to the root. Since XC is a bottom-up algorithm, subtrees closer to the root are processed in the later stages of the algorithm. Hence, a more precise version of XC is needed to exploit local affinities in those subtrees that could be processed later in the algorithm. We also studied the impact of XC's precision on the paging behavior of query Q4 when the document was clustered using the workload consisting of only query Q4. We found that the number of page faults remained constant as the chunk_size was reduced from 4096 to 32.

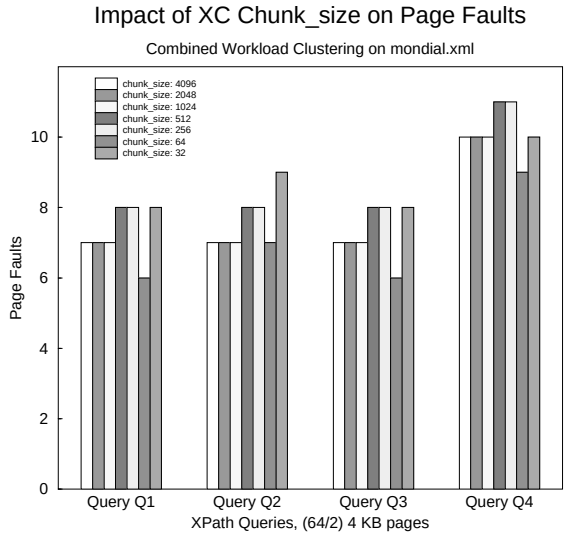


Figure 5: Impact of XC's chunk_size on the page faults for clustering using the combined query workload.

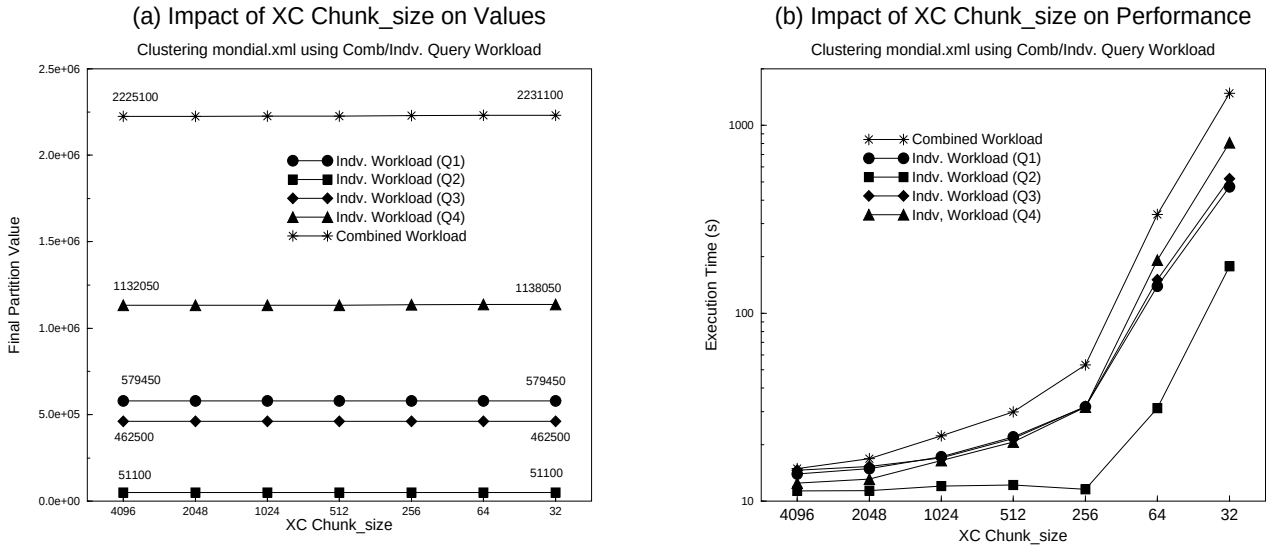


Figure 6: Impact of XC's chunk_size of the final partition value (a) and the overall execution time (b). `mondial.xml` was partitioned using the combined and individual workloads. The weight bound was 4 KB.

As shown in Figure 6(a), the difference in values corresponding to the maximum and minimum chunk_sizes (i.e., 4096 and 32) was extremely small ($< 1\%$). However, as the chunk_size was reduced, XC's running time increased by an order of magnitude (Figure 6(b)). These results illustrate that for smaller chunk_sizes (i.e., at higher precision), XC can compute a better final partition (as shown for the query Q4), but at higher precision, XC always requires large running times. These results also indicate that clustering at higher precision may not always lead to an improvement in the number of page faults.

6 Related Work

Determining if there exists a tree partition with a value at least v is NP-complete in the ordinary sense (Problem ND15, acyclic partitioning) [5]. Johnson and Niemi propose two algorithms for this problem [7]. Both exhibit higher running times than LUKS. Kundu and Misra [9] proposed a $O(n)$ tree partitioning algorithm; unfortunately, the $O(n)$ running time is possible for only for unit edge weights and the algorithm required the entire tree to be in memory.

Since early days of database development, physical clustering of related data items has been examined for exploiting data access patterns. An early reported application of clustering for physical database design was in hierarchical databases [11]. Object-oriented databases (OODBs) have used clustering of logically related objects for achieving better physical object placement [14, 6]. Object clustering in OODBs has not been widely accepted primarily due to the complexities of object-oriented programming (e.g., dynamically changing object access patterns) and problems with effectively partitioning the resultant object graph. OODBs traditionally use graph partitioning algorithms for partitioning a clustering graph whose node weights represent object sizes and edge weights denote access behavior. Such algorithms exhibit large space and time requirements. Furthermore, these graph partitioning techniques are not suitable for partitioning trees as they can't exploit structural aspects of trees for making partitioning decisions. Also, these algorithms assume the entire graph to be generated before partitioning and need *multiple in-memory* passes over the generated graph. Interestingly, [6] uses Lukes' tree partitioning algorithm when the clustering graph is a tree and concludes that it is not useful in real applications due to its large memory usage.

In [3], Fiebig et al. describe Natix, a native XML system that splits XML documents into multiple subtrees (records) such that each subtree record can fit into a page. Their splitting algorithm uses a split matrix to determine which nodes that should be always stored together. We believe a schema-agnostic workload-directed clustering algorithm like XC can be used to generate tree partitions and these partitions could be then integrated into the Natix-like record structure.

7 Discussion and Future Work

In this work, we presented XC, a new approximate tree partitioning algorithm, that is tailored for clustering static XML documents. We showed that XC improves on Lukes' tree partitioning algorithm, LUKES, in: XC is an approximate tree partitioning algorithm, operating within a memory limit (LUKES is a main memory algorithm). XC used data structures and techniques, most significantly *r*eady cluster identification, to drastically reduce memory consumption. XC exhibited running time that is linear in the size of the input XML file. XC can be used in varying degrees of precision; higher precision could potentially compute better solutions, although with higher memory and runtime costs.

Extensive experimental evaluation demonstrated that workload-directed clustering of XML document works well in practice. Furthermore, clustering of XML documents using XC is superior to clustering based on the WDFS, an alternative workload-directed clustering scheme. This validates our formulation of the XML clustering problem as a tree partitioning problem. XC was able to capture and exploit XML navigational workload affinities over paths of the abstract XML tree. Even at a lower precision, XC computes clustering solutions that are close in value to those computed by the precise LUKES algorithm, but at a significantly smaller runtime costs. XC's flexibility and efficiency makes it a good candidate to be used in workload-directed, off-line or online, clustering of XML documents.

Our current work involves extending these techniques for clustering XML documents in scenarios where either the workload is changing dynamically or the XML document is structurally updated. We also plan to apply XC's clustering techniques to complex XPath workloads, such as XSLT templates or XQuery queries. We also plan to extend XC to take into account sibling edges and the general case of XML documents with references (i.e., where the abstract XML data representation is a graph rather than a tree).

References

[1] P. Bohannon, J. Freire, P. Roy, and J. Simeon. From XML Schema to Relations: A Cost-Based Approach to XML Storage. In *Proceedings of the 18th IEEE International Conference on Data Engineering*, pages 64–80, 2002.

[2] World Wide Web Consortium. W3C Architecture Domain. `www.w3c.org/xml`. Online Documents.

[3] T. Fiebig, S. Helmer, C. Kanne, J. Mildenberger, G. Moerkotte, R. Schiele, and T. Westmann. Anatomy of a Native XML Database System. Technical Report, University of Mannheim, 2002.

[4] D. Florescu and D. Kossmann. Storing and Querying XML Data using an RDBMS. *IEEE Data Engineering Bulletin*, 22(3):27–34, 1999.

[5] M. S. Garey and D. S. Johnson. *Computers and Intractability.* W. H. Freeman and Co., 1979.

[6] C. A. Gerlhof, A. Kemper, C. Kilger, and G. Moerkotte. Partition-Based Clustering in Object Bases: From Theory to Practice. In *Proceedings of the Foundations of Data Organization and Algorithms, FODO'93, Chicago, Illinois, October 13-15*, pages 301–316, 1993.

[7] D. S. Johnson and K. A. Niemi. On Knapsacks, Partitions, and a New Dynamic Programming Technique for Trees. *Mathematics of Operations Research*, 8(1), 1983.

[8] C. Kanne and G. Moerkotte. Efficient Storage of XML Data. In *Proceedings of the 16th International Conference on Data Engineering*, San Diego, U.S.A., March 2000. IEEE Computer Society.

[9] S. Kundu and J. Misra. A linear tree partitioning algorithm. *SIAM Journal of Computing*, 6(1):151–154, 1977.

[10] J. A. Lukes. Efficient Algorithm for the Partitioning of Trees. *IBM Journal of Research and Development*, 13(2):163–178, 1974.

[11] M. Schkolnick. A Clustering Algorithm for Hierarchical Structures. *TODS*, 2(1):27–44, 1977.

[12] A. R. Schmidt, F. Waas, M. L. Kersten, D. Florescu, M. J. Carey, I. Manolescu, and R. Busse. Why and How to Benchmark XML Databases. *ACM SIGMOD Record*, 3(30), September 2001.

[13] H. Schoning and J. Wasch. Tamino - An Internet Database System. In *Advances in Database Technology – EDBT'00*, pages 383–387, 2000.

[14] 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*, pages 144–153. ACM Press, 1992.

A An Example of Tree Partitioning using LUKES

To illustrate LUKES, we consider the XML tree (T) displayed in Figure 7(a), the bold numbers denote nodes. Figure 7(b) presents the corresponding clustering tree (T') in which node weights (inside the circles) are the text sizes of the corresponding XML nodes, and the edge weights (next to the edges) model navigational behavior. Consider the case presented in Figure 7(c) with the weight bound $W = 20$. LUKES performs the following operations:

1. First, optimal partitions are generated for the leaf nodes (2 and 4) and initial partitions are generated for internal nodes (0, 1, and 3).

2. The inner loop (Step 2) is invoked on the internal nodes, e.g., node 1. Node 1 iterates over its only child, node 2. Node 2 has only one associated partition, $p^2_{10} = p^2_{opt} = \{(2)\}$. Node 1 has the initial partition, $p^1_{11} = \{(1)\}$. In the inner loop, two intermediate partitions are created: p^1_{21}, containing only the (merged) cluster $(1, 2)$, and p^1_{11} containing the (concatenated) clusters $(1), (2)$. (The subscripts are the pivot cluster size.) Since $W = 20$, the partition with weight 21 and value 25, $p^1_{21} = \{(1, 2)\}$, is immediately discarded and the partition with weight 11 and value 0, $p^1_{11} = \{(1), (2)\}$, is retained. This partition is marked optimal for node 1. Similarly, for node 3, two partitions are created, one with weight 8 and value 30, $p^3_8 = \{(3, 4)\}$, and the other with weight 5 and value 0, $p^3_5 = \{(3), (4)\}$. The partition $p^3_8 = \{(3, 4)\}$ is marked as optimal for node 3.

3. Finally, the inner loop is invoked for node 0. Node 0 has only one associated partition: $p^0_6 = \{(6)\}$. The inner loop then iterates over 0's two children, namely, nodes 1 and 3, and tries to create new partitions using the three children partitions: p^1_{11}, p^3_8, and p^3_5 (Step 2b). From among the newly formed partitions, a partition that has the maximum value will be chosen to be the final result partition.

B The XC Clustering Algorithm

XC is invoked via a procedure call, on an XML node, during parsing of an XML document within a single pass. Below we provide a high level description of the procedure, XC. Constant *hwm* (resp., *lwm*) is the high (resp., low) water mark, they satisfy $lwm < hwm < memory\_use\_limit$.

XC(XML Node x, chunk_size C)

1. While $current\_memory\_use > hwm$ do

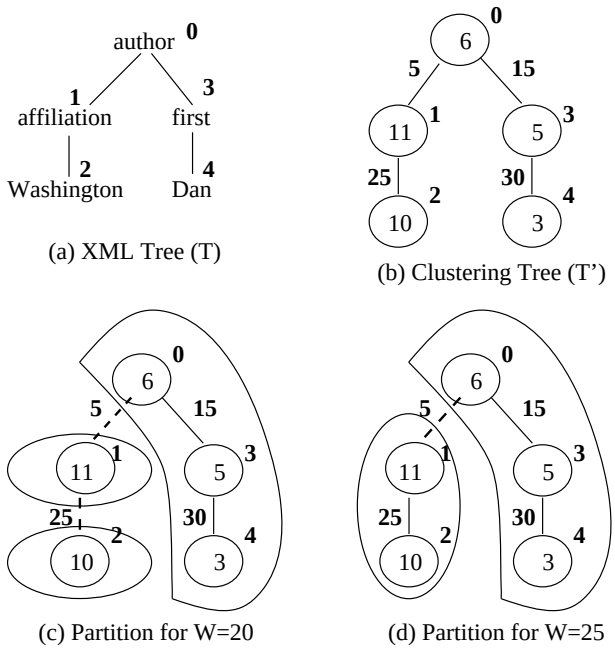


Figure 7: An XML Tree partitioned using LUKES

- From HL, the list of highest valued partitions (of processed subtrees), obtain the highest value partition P, say associated with node y. Delete P from HL.
- Forward P's clusters to Page Assignment.
- Delete y from the clustering tree.
- Let $C = W$ (C is the effective chunk_size.)

2. Initialize x:
   - Create a clustering node, x, with weight w_x which equals to x's text size. Connect x to its parent[4].
   - Form a partition P_w with value 0 and weight w, consisting of a single cluster containing node x ($P_w = \{\{x\}\}$).
   - If v is a text node (leaf), mark P_w as the optimal partition for node x. For all other nodes, mark this partition as an initial partition.
3. For the subtree rooted at node x, compute for $w' = w_x/C, \cdots, W/C$, the best intermediate partitions, with pivot cluster weight w', invoking steps specified in Section 3, LUKES Steps 2(a), (b), and (c).
4. Identify the (locally) optimal partition P^x_{opt}, for the subtree with root x, as the intermediate partition with a maximum value among the intermediate partitions associated with node x. Delete the children of node x from the clustering tree. Now node x is a leaf of the clustering tree.
5. Identify ready clusters (a ready cluster does not contain x and appears in all partitions associated with x). Forward such clusters to disk page assignment and reclaim their associated in-memory data structures.

[4]Unless x is the root node which does not have a parent.

6. If $current\_memory\_use < lwm$ then $C = chunk\_size$ (resuming normal operation).

Note: Unlike LUKES, each partitioned subtree can have at most $W_c = \frac{W}{chunk\_size}$ partitions, leading to an $O(nW_c{}^2)$ overall running time.