A Linear Time Algorithm for Optimal Tree Sibling Partitioning and Approximation Algorithms in Natix

Carl-Christian Kanne Guido Moerkotte

Department of Mathematics and Computer Science
University of Mannheim
cc|moer@db.informatik.uni-mannheim.de

ABSTRACT

Document insertion into a native XML Data Store (XDS) requires to partition the document tree into a number of storage units with limited capacity, such as records on disk pages. As intra partition navigation is much faster than navigation between partitions, minimizing the number of partitions has a beneficial effect on query performance.

We present a linear time algorithm to optimally partition an ordered, labeled, weighted tree such that each partition does not exceed a fixed weight limit. Whereas traditionally tree partitioning algorithms only allow child nodes to share a partition with their parent node (i.e. a partition corresponds to a subtree), our algorithm also considers partitions containing several subtrees as long as their roots are adjacent siblings. We call this *sibling partitioning*.

Based on our study of the optimal algorithm, we further introduce two novel, near-optimal heuristics. They are easier to implement, do not need to hold the whole document instance in memory, and require much less runtime than the optimal algorithm.

Finally, we provide an experimental study comparing our novel and existing algorithms. One important finding is that compared to partitioning that exclusively considers parent-child partitions, including sibling partitioning as well can decrease the total number of partitions by more than 90%, and improve query performance by more than a factor of two.

1. INTRODUCTION

We consider the problem of tree partitioning from the perspective of native XML data stores (XDSs). In particular, we are concerned with the quality of the storage representation of XML documents in systems that natively store the ordered, labeled tree representation of the XML documents, and use navigational primitives to access this representation during query processing. Any storage engine designed to store trees that require more space than a single unit of secondary storage must have a tree partitioning algorithm. Tree partitioning decomposes the logical document tree into partitions smaller than a weight limit, which corresponds to the storage unit's capacity, e.g. the disk page size. The tree partitioning algorithms may be ad-hoc in some systems which arbitrarily place nodes wherever there is sufficient space. In general, however, it is a good idea to carefully design partitioning algorithms for XDSs because (1) the number and structure of partitions is an important determinant of query performance, since crossing storage units during query processing is expensive, and (2) performance of the partitioning algorithm itself affects overall system performance because document insertion is a frequent operation.

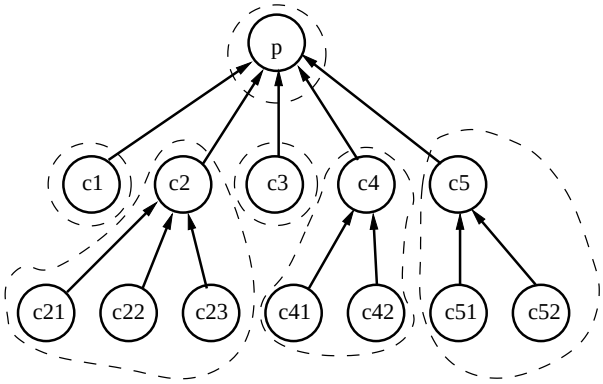

Figure 1: Partitioning with parent-child edges only

An important feature of the XML data model is order, and this must be taken into account when designing partitioning algorithms. The storage engine of an XDS not only has to store parent-child edges of a tree, but must also capture the sibling order. Storage engines for native XDSs such as IBM's System RX/DB2 Viper [2] and the Natix system [6] provide such ordered tree storage. They go even further and provide optimized storage for consecutive siblings that share a storage unit, even if their parent is located on a separate storage unit. Without such an optimization, access to nodes with a large number of children would suffer from bad performance. Consider the tree shown in Fig. 1. Assume that the root node p that does not fit on a storage unit together with any of its children's subtrees. If the storage format does not allow to put consecutive siblings into a storage unit that does not contain their parent, the resulting partitioning looks as indicated by the dashed lines in Fig. 1. In this case, each child is stored seperately, and every partition corresponds to a *single* subtree. Query evaluation with an XML query language such as XPath [1] and

VLDB '06, September 12-15, 2006, Seoul, Korea.

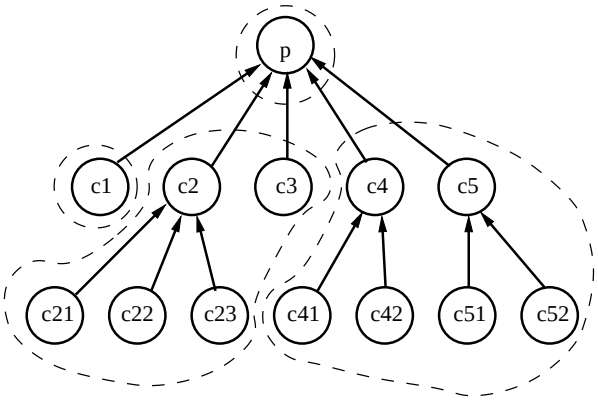

Figure 2: Partitioning with parent-child and sibling edges

XQuery [3] is expensive here. In case of an in-order traversal of all children or descendants of p, such as the evaluation of the `child` or `descendant` axis starting for context node p would access a different storage unit for every child of p, i.e. 5 storage units in total. If siblings can share a storage unit even if their parent is in a different storage unit, then we have a situation as shown in Fig. 2. Here, *several* subtrees may share a partition, as long as their roots are siblings. We call this partitioning style *sibling partitioning*. It results in fewer expensive crossings of storage unit borders (in our example, there are three), which in turn improves the query performance. To keep the number of such crossings as low as possible, a tree partitioning algorithm for XDS should create sibling partitionings and minimize the total number of partitions.

Our primary motivation for studying the tree sibling partitioning problem is our experience with the storage engine of our native XML data store Natix [6]. Natix uses a storage format where the storage units are physical records, each of which contains a fragment of the document tree whose nodes are connected by parent-child or sibling edges. Natix has two algorithms to determine which nodes share a physical record [9, 10]. The node-at-a-time algorithm [9] maintains the clustered XML storage format on incremental updates. Insertions of whole documents are handled by the bulkload component, whose design and implementation is described in [10]. Its standard partitioning algorithm for document import is a simple heuristics.

In practice, for several cases we observed peculiar partitioning decisions by this simple algorithm that lead to inacceptable query performance. Ad-hoc attempts to refine the heuristics were not very robust, i.e. always vulnerable to new pathological cases (some of them are presented throughout this paper). To be able to judge the quality of the various algorithms and to get an insight how to construct a more robust one, we wanted to know the theoretical optimum, i.e. a partitioning with a minimal number of partitions. However, determining the minimal number of partitions for a typical document is not an easy task: The number of potential sibling partitionings is exponential with respect to the number of nodes, so a brute force algorithm for determining the optimum is not feasible.

Over the last decades, a number of algorithms for tree partitioning has been developed, including [4, 5, 12, 13, 15, 17]. Several of them were specifically designed for the then-current storage engines. Tree partitioning algorithms have been studied in the context of hierarchical DBMS [13, 15], object-oriented DBMSs [17] and, recently, XDSs [4, 5]. Unfortunately, none of the algorithms considers sibling order or allows sibling subtrees to share a partition if their parent is in a different partition.

The three main contributions of this paper are:

1. We present a linear time algorithm for optimal tree sibling partitioning.

2. We present two novel, near-optimal heuristics that have much better runtime than the optimal algorithm.

3. We provide experimental results, comparing our algorithms and several existing heuristics with respect to the number of generated partitions and the query performance on the produced partitioning.

The paper is structured as follows. Sec. 2 formalizes the problem. Sec. 3 develops a sequence of algorithms for tree partitioning problems which culminate in a complete optimal algorithm for tree sibling partitioning. We discuss the problem substructure in detail, supported by formal proofs where necessary. The first of these algorithms is limited to flat trees and uses dynamic programming to partition the sequence of children. We proceed with an algorithm that applies the flat tree algorithm in a bottom-up manner to deep trees, using optimal solutions for subtrees to obtain a global solution. Unfortunately, this does not always yield an optimal solution. In some situations, a locally suboptimal tree partitioning is required for the global optimum. We present a method to generate the required local solutions. In a final step, we show how the proper local solutions can be chosen to achieve the global optimum. Although the algorithms get progressively more complex, all of them have a runtime proportional to the number of document nodes in the worst case. Sec. 4 explains why the optimal algorithm is not always a wise choice for document import into real XDSs, and presents a number of both existing and novel heuristics that are better suited for real systems. Sec. 5 assesses other existing algorithms for tree partitioning and XML document clustering. Sec. 6 evaluates our three novel and four existing sibling partitioning algorithms. Sec. 7 concludes the paper.

2. PROBLEM STATEMENT

2.1 Terms and Definitions

Let $T = (V, t, p, \lhd, w)$ be a rooted, ordered, and weighted tree with nodes V, a root t, a parent function p, a transitive sibling ordering $\lhd$, and a weight function w. p maps each nonroot node to its parent and the root to NIL, and w maps each node to a positive integer weight. In the following, the term tree always denotes a rooted, ordered and weighted tree.

Fig. 3 shows an example tree $T = (\{\mathsf{a},\mathsf{b},\mathsf{c},\mathsf{d},\mathsf{e},\mathsf{f},\mathsf{g},\mathsf{h}\},\mathsf{a}, p, \lhd, w)$, which we will use to illustrate our definitions below. In the figure, the nodes are represented as ovals with identifiers, the parent function p is represented using solid child-parent arrows, the sibling ordering is represented by the $\lhd$ symbols (with the transitive relationships such as $\mathsf{b} \lhd \mathsf{g}$ omitted), and the node weights w are the numbers in the ovals.

Given a tree $T = (V, t, p, \lhd, w)$, we denote the subtree induced by a node $v \in V$ with T_v. The *subtree weight* $W_T(v)$ is the sum of the weights of all nodes in T_v. In our example, the tree T_c consists of the nodes c, d, and e. c's subtree weight $W_T(\mathsf{c})$ is 5.

A *sibling interval* $(l, r)_T$ of T is a set of consecutive siblings determined by a first sibling l and a last sibling r with $l \lhd r$, such that $(l, r)_T := \{x | x = r \vee x = l \vee l \lhd x \lhd r\}$. A *tree sibling partitioning* P of T is a set of disjoint sibling intervals. The subtree weight of a sibling interval is $W_T(l, r) := \Sigma_{x \in (l,r)_T} W_T(x)$. The weight of a set S of sibling intervals $W_T(S)$ is the sum of the weights of the

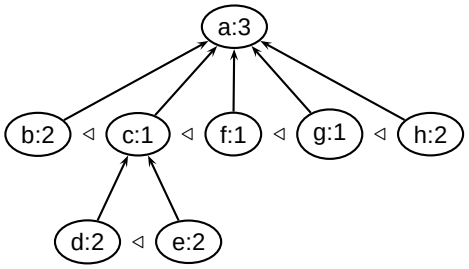

Figure 3: Example tree

contained intervals. In our example, the interval $(\mathsf{b},\mathsf{f})_T$ consists of the nodes b, c, and f, and has a subtree weight of 8.

Given a tree T and a tree sibling partitioning P as above, the *partition forest* F_T^P of T with respect to P is the set of trees that results from T when cutting the parent edges from those nodes that belong to a sibling interval in P. This is equivalent to having a parent function p^P such that for all $(l,r)_T \in P$, $\forall_{v \in (l,r)_T} p^P(v) := \mathsf{NIL}$. Hence, in F_T^P, each node that is contained in an interval in P becomes the root of a tree. The *partition* defined by an interval $(l,r)_T$ is the set of all trees from F_T^P whose root is in $(l,r)_T$. In our example, the partition defined by $(\mathsf{b},\mathsf{f})_T$ is $\{T_b, T_c, T_f\}$.

The *partition weight* $W_T^P(v)$ of a node v is its subtree weight in F_T^P. Analogously, the partition weight of a sibling interval $W_T^P(l,r)$ is the sum of all the partition weights of its nodes, and the partition weight of a set of sibling intervals is the sum of the partition weights of its intervals. The *root weight* of a partitioning is the partition weight of the root node, $W_T^P(t)$. In our example tree, consider the partitioning $P := \{(\mathsf{b},\mathsf{f})_T\}$. The root weight of P is 6, because only the nodes a, g, and h remain in the tree of the root a after the parent edges of b, c, and f have been removed.

Given T and a positive integer K, a tree sibling partitioning P of T is called *feasible* iff $(t,t)_T \in P$ and $\forall_{(l,r)_T \in P} W_T^P(l,r) \leq K$. A feasible partitioning of our example tree and $K = 5$ is $P := \{(\mathsf{a},\mathsf{a})_T, (\mathsf{b},\mathsf{b})_T, (\mathsf{c},\mathsf{c})_T, (\mathsf{f},\mathsf{g})_T\}$. Here, h is in the same partition as the root, and the root weight is 5.

A tree sibling partitioning is called *minimal* iff it is feasible and has the smallest possible cardinality of all feasible partitionings. A tree sibling partitioning P is called *lean* iff its root weight is minimal among all partitionings with the same cardinality. A tree sibling partitioning is called *optimal* iff it is both minimal and lean. In our example, $R := \{(\mathsf{a},\mathsf{a})_T, (\mathsf{c},\mathsf{c})_T, (\mathsf{f},\mathsf{h})_T\}$ is a minimal partitioning ($K = 5$) with cardinality of 3. b is in the same partition as the root, so R has a root weight of 5. However, R is not lean. There is a partitioning with the same cardinality and a smaller root weight: In $\mathcal{P} := \{(\mathsf{a},\mathsf{a})_T, (\mathsf{c},\mathsf{h})_T, (\mathsf{d},\mathsf{e})_T\}$, the root weight is 3. $\mathcal{P}$ is optimal. We will often denote optimal tree sibling partitionings with calligraphic letters such as $\mathcal{P}$ or $\mathcal{D}$.

2.2 The Tree Sibling Partitioning Problem

Given these terms, the problem we want to solve is formally stated as follows:

> **Tree Sibling Partitioning:** Given a tree T and a weight limit K, determine a minimal tree sibling partitioning.

To solve this problem, we develop algorithms that find partitionings with a stronger property, namely optimality. According to our definition, this means that the partitionings must have minimal root weight among all minimal partitionings. We will see below that the reason for this lies in our recursive, bottom-up approach: While minimality is all we need for the overall solution, the subproblems we solve must also be lean to guarantee minimality on higher levels.

3. OPTIMAL TREE SIBLING PARTITIONING

The number of feasible tree sibling partitionings for a given tree with n nodes is very large, even if a fixed weight limit K is provided. For every parent node, we have to decide which subset of children to place in the same partition as the parent. For the remaining children, we must decide how to combine the siblings into partitions. It is not at all obvious how to find a minimal partitioning in time proportional to n, given a fixed partition weight limit K. In fact, we shall see that even simplified versions of the problem are not obviously solvable in linear time.

We pursue an incremental strategy. We approach tree sibling partitioning formally, proving a sequence of properties that enable us to develop progressively more advanced algorithms.

We start out by showing that a bottom-up approach is viable because we can combine partitionings for subtrees to obtain a global solution. As a second step, we present a dynamic programming algorithm that can partition flat trees (i.e. trees where all nodes but the root node are leaves) in $O(nK^2)$ time.

Unfortunately, we will see that the bottom-up application of this algorithm to a deep tree does not necessarily yield an optimal solution: Sometimes we have to choose a suboptimal solution in the lower levels of the tree to avoid extra partitions on the next higher level. However, we can show that at each step, we only need to choose between an optimal and a nearly optimal solution, for a rather simple definition of "nearly optimal". We also show how to incorporate this choice into our dynamic programming algorithm, finally arriving at an $O(nK^3)$ algorithm for optimal tree sibling partitioning.

3.1 Bottom-Up Tree Partitioning

Our algorithms are based on the assumption that in order to determine a globally optimal partitioning, we can select a node v from the tree and determine a partitioning for the subtree induced by that node. Then we can recursively determine a global partitioning for the remainder of the tree and combine the two solutions to obtain the global solution. We will now formalize this basic assumption.

Recall that we consider a solution optimal if it is not only minimal, but also lean. The reason for this is explained below.

In the following lemma, we do *not* assume that the local subtree partitioning for a subtree T_v, called S, is locally optimal. We just assume that we know for some reason that S is part of some global solution, and show how to get a global solution based on S. We do this by collapsing T_v from the original tree into a single node v with a weight that represents the whole collapsed subtree. We recursively determe an optimal solution $\widetilde{\mathcal{P}}$ for this new tree $\widetilde{T}$, and merge this result with S to obtain an optimal solution $\mathcal{P}'$.

LEMMA 1. *Let $T = (V, t, p, \lhd, w)$ be a tree. Let $v \in V$ be a node from T. Let V_v be the nodes of T_v. Let S be a feasible tree sibling partitioning of T_v such that there exists some optimal tree sibling partitioning $\mathcal{P}$ of T that contains S and has no other intervals among the descendants of v, i.e. with $S - \{(v,v)_T\} = \{(l,r)_T \in \mathcal{P} | (l,r)_T \subseteq T_v\}$.*

Further, let $\widetilde{T} = (V - V_v \cup \{v\}, t, \widetilde{p}, \widetilde{\lhd}, \widetilde{w})$ be the tree T with the descendants of v removed, such that $\widetilde{p}$, $\widetilde{\lhd}$ and $\widetilde{w}$ are the functions from T restricted to $\widetilde{T}$, with the exception of a new weight for v: $\widetilde{w}(v) := W_T^S(v)$. Let $\widetilde{\mathcal{P}}$ be an optimal tree sibling partitioning of $\widetilde{T}$.

Then $\mathcal{P}' := \widetilde{\mathcal{P}} \cup (S - \{(v,v)_T\})$ is an optimal tree sibling partitioning of T.

We omit the proofs of our lemmas due to space constraints. They are contained in the extended version of this paper [11].

Lemma 1 suggests an algorithm that traverses the tree in a bottom-up manner. For each non-leaf node v, determine a partitioning S for T_v that is part of a global solution (we will explain how to do this in the remaining section). We then remove the sibling intervals in S (except $(v,v)_T$), and replace the whole subtree T_v by a single node whose weight is equal to the total weight of all the nodes in T_v that are not part of an interval in S. Then we proceed with the next node.

This approach reduces our original problem of finding partitionings for arbitrary trees to the simpler problem of finding partitionings only for flat trees. Flat trees are trees in which all nodes but the root node are leaves. Our bottom-up approach guarantees that, once we reach an inner node, all deeply nested subtrees below that node have been pruned and only a flat tree remains.

The bottom-up traversal is also the reason why we require the local solutions to be lean in addition to be minimal: By cutting away as much of the tree weight as possible, we generate a simpler subproblem in the next higher level of the tree. Of course, we only do so if it does not introduce additional sibling intervals, because the ultimate goal is to find a minimal partitioning.

However, we have not yet specified the subroutine to compute a suitable partitioning for flat trees. We turn to this problem in the next section.

3.2 Flat Tree Partitioning

Before looking at an optimized way to determine an optimal tree sibling partitioning for flat trees, we want to verify that a simple brute-force algorithm is not a viable solution. Let us look at the number of feasible tree sibling partitionings in a flat tree. Assume a flat tree with n leaf nodes in which all the nodes have weight 1. In this case, we can put up to $K-1$ leaves of the n leaves in the same partition as the root. There are $\binom{n}{K-1}$ possible ways to do this. Hence, a lower bound for the number of feasible root partitions is $\Omega(n^{K-1})$. This estimate does not yet include the different possibilities to group the remaining sibling nodes into intervals. Hence, it is reasonable to conclude that a brute force algorithm is impractical for typical values[1] of K.

We approach the problem by showing that an optimal solution for a tree with n leaf nodes contains an optimal solution for a tree with less than n leaves. Then, we use this knowledge to develop a dynamic programming algorithm that finds the optimal solution in $O(nK^2)$ time. Finally, we discuss the optimization potential of the algorithm.

3.2.1 Optimal Substructure for Flat Trees

Consider the options we have for a leaf in the solution: either the child is put into the same partition with the root, or it belongs to an interval of the result partitioning. Together with the fact that there is only a limited number of feasible intervals to which the child can belong, this forms a useful problem substructure for dynamic programming.

The following lemma states that we can find an optimal tree sibling partitioning for a flat tree with n leaf nodes by choosing the best candidate solution from one of the following two subproblems: Either (1) the solution is the same as an optimal tree sibling partitioning for a similar tree with $n-1$ nodes, which represents the original tree with the last child put into the root partition, or (2) the solution can be constructed by adding a single interval to an optimal partitioning for a smaller tree.

LEMMA 2. *Let $T = (\{t, c_1, \ldots, c_n\}, t, p, \triangleleft, w)$ be a tree in which all nodes but the root node are leaves, i.e. $p(c_i) = t$. $\triangleleft$ orders the c_i according to their index value i. The tree $T_j^s = (\{t, c_1, \ldots, c_j\}, t, p, \triangleleft, w_j^s)$ is the tree T with all children $\{c_i | i > j\}$ removed and a different weight s for the root node t, with p and $\triangleleft$ regarded as restricted to $\{c_1, \ldots, c_j\}$, and $w_j^s(t) := s$, and for $v \in \{c_1, \ldots, c_j\}$, $w_j^s(c_i) := w(c_i)$. Let D_j^s be the set of optimal tree sibling partitionings for T_j^s.*

Then, for $j = 0$ and any $s \leq K$ holds $D_0^s = \{\{(t,t)_T\}\}$.

For each j with $1 \leq j \leq n$, at least one of the following statements holds:

1. *$D_{j-1}^{s'} \subseteq D_j^s$ with $s' := s + w(c_j)$.*
2. *For some m with $0 \leq m < j \wedge m < K$: $\forall_{M \in D_{j-m-1}^s} (M \cup \{(c_{j-m}, c_j)_T\}) \in D_j^s$.*

Given the notation used above, the problem that has to be solved by our algorithm can be stated like this: *Find an arbitrary element of $D_n^{w(t)}$.*

Since such an arbitrary element of $D_n^{w(t)}$ can be computed from optimal sibling partitionings D_j^s for smaller trees ($j < n$), the problem is susceptible to a dynamic programming approach, as we show below.

3.2.2 Algorithm FDW for Flat Trees

Our algorithm FDW (*F*lat trees, *D*ynamic programming for tree *W*idth) employs dynamic programming by starting out with a tree that only contains the root node. We then successively add all leaf nodes in left-to-right order and iterate over all potential weights of the root node. For each such intermediate tree, we compute an optimal partitioning. For each node, we have to decide whether it is going to be part of a sibling interval in the solution, or whether it will be part of the root partition. This decision is based on optimal solutions for already processed intermediate trees.

More formally, for each $j < n$ and each $s \in \{w(t), \ldots, K\}$, we determine a single element $\mathcal{P} \in D_j^s$, and store it in a table indexed by j and s.

For $j = 0$, we have $D_0^s = \{\{(t,t)_T\}\}$ for all s, hence $\mathcal{P} = \{(t,t)_T\}$. For $j > 0$, Lemma 2 states that we only have to consider a limited number of candidates: Either our desired partitioning $\mathcal{P}$ is an arbitrary element of $D_{j-1}^{s'}$, or, for some m with $0 \leq m < K$, an arbitrary element of D_{j-m-1}^s together with $(c_{j-m}, c_j)_T$.

Hence, we have at most $K+1$ candidates in each step. We process the steps in increasing order of j. This makes sure we already know a partitioning from every D_i^s with $i < j$. To determine $\mathcal{P}$ in each step, we just check all candidates for feasibility and store a candidate with minimum cardinality that also has minimum root weight.

The algorithm in Fig. 4 implements this strategy. It uses a table $D(s,j)$ to store a partitioning from D_j^s. It is assumed that out-of-bounds accesses to D (i.e. where $s > K$) always return a dummy interval with card $= \infty$. This makes exposition of our algorithm simpler (we do not need to check for out-of-bounds conditions in the pseudocode).

Lemma 2 tells us that each partitioning $D(s,j)$ is either the same as an existing partitioning, or it extends an existing partitioning by at most one interval. Hence, it is sufficient to store as entries in D either a copy of another partitioning, or only the added interval and a pointer to the remaining chain of intervals. This `next` pointer is implemented as a pair of indices of another partitioning in the

[1] Keep in mind that K is the size limit for a storage unit, and typical disk page sizes are thousands of bytes.

input $T = (\{t, c_1, \ldots, c_n\}, t, p, \lhd, w)$ flat tree
output D dynamic programming table with final result in $D(w(t), n)$

for $s := w(t)$ **to** k
  $D(s,0)$.begin:=t
  $D(s,0)$.end:=t
  $D(s,0)$.card:=1
  $D(s,0)$.rootweight:=$w(t)$
  $D(s,0)$.next:=(0,0)
for $j := 1$ **to** n
  for $s := w(t)$ **to** K
    $s' := s + w(c_j)$
    $P := D(s', j-1)$
    $w := 0$
    $m := 0$
    while $m < j \wedge m < K \wedge w < K$
      $w := w + w(c_{j-m})$
      if $w \leq K$
        crd:= $D(s, j-m-1)$.card+1
        rw:= $D(s, j-m-1)$.rootweight
        if crd< P.card$\vee$(crd= P.card$\wedge$rw< P.rootweight)
          P.begin:=c_{j-m}
          P.end:=c_j
          P.card:=crd
          P.rootweight:=rw
          P.next:=$(s, j-m-1)$
      $m := m + 1$
    $D(s,j) := P$

Entries in the dynamic programming table $D(i,j)$

begin	first node of interval
end	last node of interval
card	cardinality of best parititioning so far (length of next chain)
rootweight	rootweight of best parititioning so far
next	index of next interval

Figure 4: Algorithm FDW for flat tree partitioning

table. The new or copied interval is represented by the two bounding nodes (`begin` and `end`). In addition, each entry in D (Fig. 4) has two fields that contain the cardinality and the weight of the root partition to avoid recomputing them during comparisons.

Having run the algorithm, an optimal tree sibling partitioning can be obtained by starting at the interval in $D(w(t), n)$ and traversing the list of next pointers until we reach a next pointer with value $(0, 0)$.

It is easy to specify the runtime of this algorithm: There are three nested loops, the outermost loop is processed at most n times, and the two inner loops are processed at most K times. Hence, the algorithm has a worst-case runtime of $O(nK^2)$.

3.2.3 Optimizations

We do not need to determine $D(s, j)$ for every value of s. For each j, we only need to consider those values of s that are needed by higher values. These are always the sum of node weights. Hence, we only need s values that are sums of the weight of the root node and the weights of nodes to the right of c_j.

A simple way to achieve this is to use the memoization technique: We do not precompute all entries in the D table as shown in our pseudocode. Instead, we only compute the entries on demand and remember them in D. Subsequent requests for the same entry can then be satisfied in $O(1)$ time. This does not affect the asymptotical complexity, but significantly reduces real-world runtime: Only a fraction of the whole D table is really needed for real trees (for an example, see Sec. 3.3.6).

input T tree
output D dynamic programming table

for all nodes v of T in postorder
  for $s := w_T(v)$ **to** K
    $D(v,s,0)$.begin:=NIL
    $D(v,s,0)$.end:=NIL
    $D(v,s,0)$.card:=0
    $D(v,s,0)$.rootweight:=s
    $D(v,s,0)$.next:=(0,0)
  for $j := 1$ **to** $childcount(v)$
    for s := $w_T(v)$ **to** K
      $s' := s + D(v).rootweight$
      $P := D(v, s', j-1)$
      $w := 0$
      $m := 0$
      while $m < j \wedge m < K \wedge w < K$
        $w := w + D(c_{j-m}(v)).rootweight$
        if $w \leq K$
          crd:= $D(v, s, j-m-1)$.card+1
          rw:= $D(v, s, j-m-1)$.rootweight
          if crd< P.card
              $\vee$(crd= P.card$\wedge$rw< P.rootweight)
            P.begin:=$c_{j-m}(v)$
            P.end:=$c_j(v)$
            P.card:=crd
            P.rootweight:=rw
            P.next:=$(s, j-m-1)$
        $m := m + 1$
      $D(v,s,j) := P$

Figure 5: Algorithm GHDW for deep tree partitioning

3.3 Optimal Deep Tree Partitioning

In this section, we extend the FDW algorithm for optimal flat tree partitioning to deep trees. We first propose a straightforward extension of FDW to deep trees, called GHDW (for Greedy–Height/Dynamic–Width). GHDW uses locally optimal partitionings to construct a global partitioning. However, there are cases in which GHDW yields suboptimal results, and we show that a globally optimal solution for our partitioning problem sometimes requires locally suboptimal solutions. We then characterize the kind of locally suboptimal partitionings that are needed and show how to generate them. We incorporate this into our dynamic programming algorithm. Finally, we arrive at a linear time algorithm for optimal tree sibling partitioning.

3.3.1 The GHDW Algorithm

It is tempting to use the result of the FDW algorithm as S in Lemma 1. In a bottom-up manner, we can collapse flat subtrees into single nodes, producing a flat tree at the next higher level. When reaching the root, we have a feasible tree sibling partitioning that is constructed from locally optimal partitionings. This approach uses our dynamic programming algorithm for each inner node, but with respect to the "height" of the tree, it procedes in a greedy manner, always choosing an optimal partitioning for each subtree. We call this approach GHDW (employing a *G*reedy strategy for the *H*eight of the tree, and *D*ynamic programming for the *W*idth).

The pseudo-code is shown in Fig. 5. The code looks like FDW with an additional outer loop over all nodes. However, we now deal with deep trees and use appropriate primitives to access the tree structure: We use $c_j(v)$ to specify the jth child of v, and $childcount(v)$ to denote the number of children of v.

For each node, flat tree partitioning is performed once and the results are stored in the dynamic programming table D. In GHDW, D has an extra dimension compared to FDW because we have one

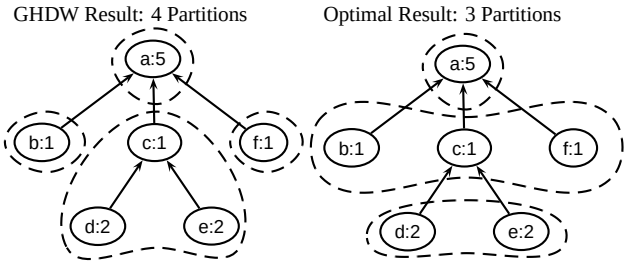

Figure 6: Failure of the greedy strategy ($K = 5$)

"flat" result for each node v. We abbreviate with $D(v)$ the result of $D(v, w(c_j(v)), childcount(v))$, which is the best partitioning for T_v that GHDW can find. Note that the `card` field in the D entries counts only the extra partitions that result from partitioning the children of v, not the partitions on the lower levels. We do not need the actual partition count, but only need to choose the locally minimal one here. Collapsing the optimally partitioned subtree into a single node at the next higher level is realized in a simple way: in those places where FDW uses the weight of a node, GHDW uses the rootweight field of the optimal subtree partitioning for the nodes on the lower level.

Note that the initial s loop for $j = 0$ is different from FDW; instead of an interval $(v, v)_T$ for the subtree's root v, we store an empty "dummy" interval, because the actual interval to which v belongs is determined in a higher level. The global result is extracted from the D table by returning for each node the next-chain of partitions (without the empty interval), and finally adding an extra *root interval* $(t, t)_T$.

The complexity of GHDW is the same as FDW: $O(nK^2)$. At first glance, there is an extra outer loop with n steps. However, the inner loop on j does not range over all nodes any more, but only over the children of the current outer node v. Hence, the total number of iterations of the j loop over all v is n, and the asymptotical complexity remains the same as for FDW.

We will see in the evaluation (Sec. 6) that the GHDW algorithm yields good results. However, the results obtained by this algorithm are not always optimal.

As an example for a suboptimal result of GHDW, consider Fig. 6. Here, the weight limit is $K = 5$. The numbers represent the node weights.

The left part of the figure shows the result of the GHDW algorithm, which consists of the four intervals {(a,a), (b,b), (c,c), (f,f)}. When processing bottom-up, GHDW decides that the optimal solution for the flat subtree induced by node c is to make d and e share a partition with c. This optimal subtree partitioning yields a tree T_c with a total weight of 5. Hence, on the next level c is treated as a single node of weight 5. But this means that both b and f get a partition of their own: They can neither be put into the same partition as the root a nor with the c subtree, because both alternatives would exceed the weight limit.

The right part of the figure shows a feasible partitioning {(a,a), (b,f), (d,e)} with just three partitions. By introducing an extra partition at the lowest level of the tree (d and e), it becomes possible to merge all of the siblings on the middle level into a single partition. This is an optimal sibling partitioning for this tree. Here, the subtree of c is distributed over two partitions instead of the locally optimal solution, which requires only one partition for that subtree.

Before we investigate what kind of locally suboptimal partitionings we have to take into account, we first provide some additional definitions needed in the remaining sections.

3.3.2 Definitions

Let $T = (V, t, p, \triangleleft, w)$ be a tree and Q be a tree sibling partitioning for T. Q is called *nearly minimal* iff it contains exactly one more interval than a minimal partitioning. Q is called *nearly optimal* iff it is both nearly minimal and lean.

$\Delta W(v)$ describes for any node $v \in V$ the difference in root weight of the optimal and nearly optimal partitioning of the subtree T_v induced by v. Let $\mathcal{P}$ be an optimal and Q a nearly optimal tree sibling partitioning for T_v, respectively. Then we define

$$\Delta W(v) := \begin{cases} 0 & \text{if } Q \text{ does not exist} \\ W^{\mathcal{P}}_{T_v}(v) - W^{Q}_{T_v}(v) & \text{otherwise} \end{cases}$$

3.3.3 Optimal Substructure for Deep Trees

We now show which partitionings of subtrees are suitable candidates to construct an optimal tree sibling partitioning at the next higher level. It turns out that there are only two candidates. In fact, the following lemma shows that if an optimal subtree partitioning for a subtree is not part of a globally optimal solution, then a nearly optimal one must be.

LEMMA 3. *Let $T = (V, t, p, \triangleleft, w)$ be a tree, and let $v \in V$ be a node in T. Then, there exists an optimal tree sibling partitioning $\mathcal{P}$ of T, and a tree sibling partitioning S of T_v, such that S is either optimal or nearly optimal, and $S - \{(v, v)_T\} = \{(l, r)_T \in \mathcal{P} | (l, r)_T \subseteq T_v\} - \{(v, v)_T\}$, i.e. the set of intervals in $\mathcal{P}$ that lie in the subtree below v is exactly S.*

Further, if S is nearly optimal, then $\Delta W(v) > 0$.

So, the good news is that we only need to consider at most two candidate subtree partitionings for each node. The bad news is that we do not know which of the two is part of a global solution until we are processing the next higher level.

Before we can tackle this problem, we need an algorithm that computes a nearly optimal tree partitioning. Fortunately, we can use any algorithm for optimal tree sibling partitioning to also obtain nearly optimal partitionings, as we show in the next subsection.

3.3.4 Nearly Optimal Partitioning

Once we have an optimal tree sibling partitioning for a tree T, obtaining a nearly optimal partitioning is just a matter of rerunning the algorithm with a slightly modified tree, as the following lemma shows.

LEMMA 4. *Let $T = (V, t, p, \triangleleft, w)$ be a tree and $\mathcal{P}$ an optimal tree sibling partitioning for T. Let $T' = (V, t, p, \triangleleft, w')$ be the tree T with a modified weight for the root node $w'(t) := w(t) + K - W^{\mathcal{P}}_T(t) + 1$, and for $v \in V - \{t\}$, $w'(v) := w(v)$.*

Then, any optimal tree sibling partitioning for T' is a nearly optimal tree sibling partitioning for T.

Further, if there is no optimal tree sibling partitioning for T', then $\Delta W(t) = 0$.

Lemma 4 states that we can find the nearly optimal partitions in the D table of the GHDW algorithm by looking at entries with modified s values. The nearly optimal partition $Q(v)$ for the subtree T_v can be determined using the D table as follows (remember that $D(v)$ stands for the optimal partitioning of the subtree T_v):

$$Q(v) := D(v, w(v)+K-D(v).rootweight+1, childcount(v))$$

.

While Lemma 4 provides us with a simple way to obtain a nearly optimal partitioning, we still have to address the issue that we do not know whether to choose an optimal or nearly optimal partitioning until we have reached the next higher level in the tree. How to efficiently solve this problem is explained in the next subsection.

3.3.5 Algorithm DHW for Deep Trees

A brute-force approach for deep trees is easy to specify: Traversing the tree bottom-up, we determine both an optimal and a nearly optimal sibling partitioning for each non-leaf node. This leads to two potential weights for this node in the next higher level. When determining the optimal sibling partitioning in the next higher level, we run the flat-tree algorithm once for each of the two potential weights of each node, trying to determine which one yields a lower total interval count.

However, with two choices for the weight of each node, in the worst case the brute-force algorithm has to compute an optimal tree sibling partitioning for 2^n weight combinations, each requiring $O(nK^2)$ time to check. This exponential time usage prohibits its usage on real-world document sizes.

Instead of such a brute-force approach, we again use dynamic programming for the decision whether to use an optimal or nearly optimal tree sibling partitioning for each child node's subtree, and incorporate this choice into the GHDW algorithm from Sec. 3.3.1. We call the resulting algorithm DHW (*D*ynamic programming for *H*eight and *W*idth).

The following lemma shows that the use of nearly optimal partitionings is an exception rather than the rule. In a lot of cases, only an optimal tree sibling partitioning for a node v needs to be considered as part of an optimal solution. In particular, according to the statements of the lemma below, (1) an optimal partitioning is sufficient if v shares a partition with its parent, and (2) we only need to consider a nearly optimal partitioning for the subtree of a node v if all the other nodes u in the same interval with a greater ΔW already use a nearly optimal partitioning for their subtrees.

Given a tree $T = (V, t, p, \triangleleft, w)$, for any node $x \in V$, let $\mathcal{P}_x$ be an optimal tree sibling partitioning for T_x, and Q_x be a nearly optimal tree sibling partitioning of T_x such that $\Delta W(x) > 0$. Note that Q_x may not exist.

LEMMA 5. *For each tree $T = (V, t, p, \triangleleft, w)$ there exists an optimal tree sibling partitioning $\mathcal{P}$ of T such that for each $v \in V$, $\mathcal{P}_v - \{(v, v)_T\} \subseteq \mathcal{P}$ if any of the following statements are true:*

1. *There is no interval $(l, r)_T \in \mathcal{P}$ such that $v \in (l, r)_T$.*
2. *There is an interval $(l, r)_T \in \mathcal{P}$ such that $v \in (l, r)_T$, and there is a $u \in (l, r)_T$ for which Q_u exists and $Q_u - \{(u, u)_T\} \not\subseteq \mathcal{P} \wedge \Delta W(u) < \Delta W(v)$.*

This result helps us to extend the GHDW algorithm (see Fig. 5). GHDW decides for each node, based on partitionings of smaller trees, whether to put the new node into a partition with the parent, or whether to create a new sibling interval containing the new node. From statement 1 in Lemma 5 it follows that the only point in the algorithm where we need to consider nearly optimal tree partitionings is when adding new intervals, i.e. in the loop on m.

Further, statement 2 tells us that we only need to consider nearly optimal partitionings for the subtree of a node if a nearly optimal partitioning has already been used for all nodes v of the same interval which have a greater $\Delta W(v)$. Hence, given an interval with a weight larger than K, we can order the contained nodes by descending ΔW value and store them in a list. Then, we select a node from the beginning of the list and use a nearly optimal partitioning for that node. This will decrease the weight of the interval by the node's ΔW. We continue to do so while the weight of the interval is larger than K and there are still nodes in the list. We remember each node for which we use a nearly optimal partitioning. When considering the resulting interval as part of a global solution and comparing the overall cardinalities, we must take into account that

input T tree
output D dynamic programming table

for all nodes v of T in postorder
  for $s := w_T(v)$ **to** K
    $D(v, s, 0)$.begin:=t
    $D(v, s, 0)$.end:=t
    $D(v, s, 0)$.nearlyopt:=$\emptyset$
    $D(v, s, 0)$.card:=1
    $D(v, s, 0)$.rootweight:=s
    $D(v, s, 0)$.next:=(0,0)
  for $j := 1$ **to** $childcount(v)$
    for $s := w_T(v)$ **to** K
      $s' := s + D(c_j(v), w_T(c_j(v)), childcount(c_j(v))).rootweight$
      $P := D(v, s', j-1)$
      $w := 0$
      $dw := 0$
      $m := 0$
      while $m < j \wedge m < K \wedge w - dw < K$
        $w := w + D(c_{j-m}).rootweight$
        $dw = dw + \Delta W(c_{j-m}(v))$
        if $w - dw \leq K$
          crd:= $D(v, s, j-m-1)$.card+1
          rw:= $D(v, s, j-m-1)$.rootweight
          C :=nodes $(c_{j-m}(v), c_j(v))_T$
              ordered by descending $\Delta W(c_i(v))$
          $w' := w$
          $N := \emptyset$
          while $w' > K$
            $u := head(C)$
            $w' := w' - \Delta W(u)$
            $N := N \cup \{u\}$
            $C := C - \{u\}$
            crd:=crd+1
          if crd< P.card$\vee$(crd= P.card$\wedge$rw< P.rootweight)
            P.begin:=$c_{j-m}(v)$
            P.end:=$c_j(v)$
            P.nearlyopt:=N
            P.card:=crd
            P.rootweight:=rw
            P.next:=$(s, j-m-1)$
        $m := m + 1$
      $D(v, s, j) := P$

Figure 7: Algorithm DHW for deep tree partitioning

we have not only added the interval itself, but also use one extra interval for each node that uses a nearly optimal partitioning.

The resulting algorithm DHW is shown in Fig. 7. As explained, it is an extension of the GHDW algorithm. The dynamic programming table D has a new field `nearlyopt` which contains the subset of nodes in the interval that use a nearly optimal partitioning.

The algorithm uses the function $\Delta W(v)$, which is computed as $\Delta W(v) := D(v).rootweight - Q(v).rootweight$. For the nodes required in each step, these results are already available in the table. As explained in Sec. 3.3.4, the entries for Q can be found in the table for D using modified s values.

New intervals are added by ranging over all possible m values as before. However, to determine the maximal size of the interval, we do not use the sum of root weights of optimally partitioned subtrees w, but the sum of root weights of nearly optimal partitionings $w - dw$. The loop terminates if the interval becomes heavier than K even if only nearly optimal partitionings are used.

For each m value, a list C of the interval's nodes is built as explained above. The algorithm consecutively removes the node from the beginning of this list and uses a nearly optimal partitioning for the corresponding subtree. This list determines for which subtrees a nearly optimal partitioning has to be used. Those nodes (N) whose

subtrees use a nearly optimal partitioning are stored in the dynamic programming table as field nearlyopt.

We now turn to the runtime complexity of the algorithm. The computation of the nearly optimal partitionings does not change the asymptotical complexity because the required entries are present in the D table anyway (refer to Sec. 3.3.4 for an explanation). In addition to the loops present in the GHDW algorithm, DHW has an additional loop with $O(K)$ steps to determine the subset Q for each interval candidate. DHW also needs to create the ordered C list, which requires $O(K \log K)$ time in the worst case. Hence, the added complexity for each inner step is $O(K \log K)$. In GHDW, there were $O(nK^2)$ steps, so the overall time complexity of DHW is $O(nK^3 \log K)$.

This means DHW is a linear time algorithm for optimal tree sibling partitioning.

3.3.6 Optimizations

The DHW algorithm provides several opportunities for optimization. The memoization approach already discussed in Sec. 3.2.3 is still applicable, of course. For example, measurements for a 20 MB sample document and $K = 256$ show that on average, less than 4 of the potential 256 values for s actually occur for inner nodes.

Further, it can be shown that for the subtree induced by the first and last nodes of an interval, an optimal partitioning is always sufficient to generate a globally optimal solution. Hence, we need to add to C all nodes but the first and last node of an interval. We omit the proof here for brevity reasons, as it does not impact asymptotical complexity.

It is also possible to avoid the construction of C from scratch during each iteration of the loop on m. Instead, C could be stored as a priority queue and incrementally maintained by adding new nodes in every step. This lowers the overall complexity to $O(nK^3)$ (the inner loop with K steps remains).

4. IMPLEMENTATION ASPECTS AND APPROXIMATION ALGORITHMS

Our optimal algorithm DHW runs in linear time with respect to the number of nodes, and enables us to determine optimal sibling partitionings for real documents in a reasonable amount of time. Unfortunately, the DHW algorithm has a number of disadvantages that make its application as an algorithm for document insertion a suboptimal choice, as we will see below.

In this section, we discuss practical issues that influence the choice of a partitioning algorithm for native XML Data Stores. We further present several tree partitioning algorithms that do not achieve optimal results, but are better suited for inclusion in real XDSs. Some of these algorithms are existing algorithms in their original form, some of them are minor modifications of existing algorithms, and one is a novel variation of an existing algorithm that we propose based on our study of the optimal algorithm.

4.1 Implementation Aspects

A partitioning algorithm for document import in a native XDS must meet a number of requirements in addition to a small number of generated partitions. While implementation details are out of the scope of this paper (details about Natix' document importer can be found in [10]), there are some properties that a good candidate algorithm for a real XDS must have: (1) It must be as fast as possible, (2) it must scale to very large documents, and (3) it must be robust, i.e. the quality of its result should not vary greatly for typical documents.

Unfortunately, our DHW algorithm has resource requirements that can be a problem in practice:

1. The dynamic programming table requires a large amount of memory.
2. While the algorithm is linear with respect to the number of nodes, the factor of K^3 is significant, as we will see in the evaluation section.
3. The final partitioning is only determined after the whole tree has been processed, because the decision on whether to use an optimal or nearly optimal partitioning for each subtree depends on the decision at the next higher level of the tree. Ultimately, these decisions depend on the partitioning of the children of the root node. This means that, when used as an algorithm for document insertion, the whole document to be inserted must be kept in main memory until the partitioning is completed. This does not scale to very large documents.

These issues prompt us to investigate approximation algorithms that are better suited for implementation and still deliver good (small) partitionings.

To avoid issue (3) above, algorithms must be able to decide about the definitive assignment of a node to a partition before it has seen all the document nodes. If this is the case, a node may be stored on secondary storage as soon as it is assigned to a partition. Further, if the associated information for such a node is not required for processing of the remaining nodes, the whole node can be removed from main memory. We call algorithm that exhibit this property *main-memory friendly*. We have already seen that DHW is not main-memory friendly.

In the following, we will present several approximation algorithms with respect to their applicability as XDS partitioning algorithms.

4.2 Top-Down Approximation Algorithms

Tsangaris et al. [17] study several partitioning algorithms in the context of object-oriented DBMS. We consider two of their top-down algorithms. For both algorithms, the resulting clustering is not optimal and does not produce storage units that only contain connected nodes. However, we can easily adapt two of their algorithms to produce approximate tree sibling partitionings, as explained below.

4.2.1 DFS

The DFS algorithm processes a graph using depth-first search, assigning nodes greedily to the current cluster. New clusters are created whenever the current cluster cannot hold the current node.

The original algorithm does not care for the connectedness of the partitions, as required by tree sibling partitioning. However, we can modify the algorithm such that it starts a new partition not only when the current one is full, but also when a node to be processed is not connected with any of the nodes in the current partition by sibling or parent-child edges.

This variant of DFS is main-memory friendly because we decide immediately for each node to which partition it belongs. The algorithm is particularly suited to XML processing because typical XML parsers deliver the input documents as a stream of parsing events in depth-first preorder of the document tree.

4.2.2 BFS

The same strategy explained above can also be applied to breadth-first search, where for each visited node, we try to add it to the partition of its parent, and if that is full, to the partition of its previous sibling. BFS is not main-memory friendly, as we need to see all of the nodes in the document to perform proper breadth-first search. We include it for reasons of completeness.

4.3 Bottom-Up Approximation Algorithms

We now discuss algorithms that operate in depth-first postorder, i.e. bottom-up. While the typical result delivery of XML parsers is depth-first preorder, all of the algorithms below can be implemented in a main-memory friendly way because they can start assigning nodes to partitions before the whole document has been parsed: Whenever the algorithms leave a subtree that is larger than K, they process its nodes, creating partitions until the subtree falls below K.

In the worst case, i.e. when the document consists of a root node with a very large fan-out, these algorithms still need to keep the whole document tree in memory. However, it is possible to mitigate this problem, as proposed in [10]. Instead of waiting until all of the children of the currently processed nodes have been delivered by the parser, we can already run the algorithm if the main memory consumption for the representation of the current node's subtree exceeds a certain threshold. We partition the subtree seen so far, moving some partitions from main memory to secondary memory, and then continue accepting further nodes from the parser. While this technique deteriorates the quality of the result, it achieves an upper bound for the memory usage that is proportional to the document height, and not to the number of document nodes.

4.3.1 GHDW

We have already presented a bottom-up approximation algorithm in detail: The GHDW algorithm that we developed as a precursor to the optimal algorithm in Sec. 3.3.1. It achieves very good results in practice (see Sec. 6), and is also memory-friendly, because it determines a definitive subtree partitioning for every encountered subtree that is heavier than K.

4.3.2 Rightmost Siblings (RS)

The existing Natix document insertion algorithm [10] applies a very simple heuristic. When processing a node whose subtree is larger than K, it iterates over the node's children from right to left and adds siblings to a new partition until the partition weight reaches K. The algorithm continues to create partitions until the current node's subtree weight falls below K.

This approach is main-memory friendly and very simple to implement.

4.3.3 Kundu and Misra (KM)

The algorithm of Kundu and Misra [12] minimizes the total number of partitions for a given tree, and enforces connectedness of partitions, albeit only based on parent-child edges.

The algorithm KM operates by processing the nodes in bottom-up fashion. Whenever the subtree induced by the current node p is heavier than K, the algorithm selects the heaviest child v of p and creates a partition for its subtree. This is repeated until the subtree weight of p falls below K. The result is a tree partitioning with minimal cardinality.

In our terminology, the produced partitionings only contain intervals with a single node, i.e. of the form $(v, v)_T$. Neighboring intervals are not merged even if their respective subtrees have a combined weight equal to or smaller thank K. This introduces costly and unnecessary extra storage units. However, the algorithm is very fast. It has linear runtime and processes each node exactly once. Its complexity does not depend on the weight limit K, and it is memory-friendly.

4.3.4 Enhanced Kundu and Misra (EKM)

One way of extending the Kundu and Misra algorithm to sibling partitioning can be derived from a problematic case of the GHDW algorithm. Recall Fig. 6, where the optimal result was obtained by a "layered" partitioning that created one partition for every level of the tree. Further, recall that the reason for the failure of GHDW's greedy strategy is that an optimal partitioning sometimes requires additional partitions at one level below the currently processed level.

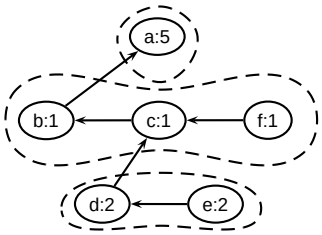

Figure 8: Binary tree for Enhanced KM ($K = 5$)

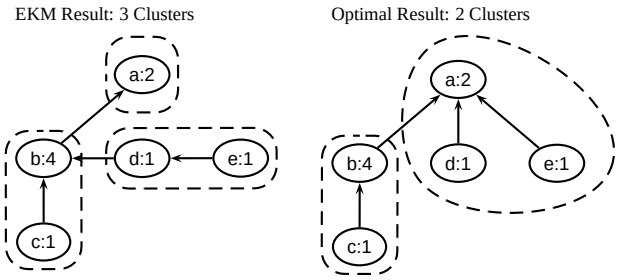

Figure 9: Failure of Enhanced KM ($K = 5$)

This knowledge about tree sibling partitioning can be leveraged to create a variant of the Kundu and Misra algorithm: Instead of running the algorithm on the n-ary tree representation as in the original, we convert the tree into a binary tree representation, where every node has only two children: the left child is its first child in n-ary representation (if there is one), and the right child is the right sibling in n-ary representation (if there is one). The binary representation of the tree from Fig. 6 is shown in Fig. 8.

We propose the "enhanced Kundu and Misra algorithm" EKM, which is simply KM applied to the binary representation. For the tree from Fig. 8, the algorithm proceeds as follows: When processing node c, the algorithm finds that the subtree of c (comprising nodes c,d,e,f) is heavier than $K = 5$. It decides to create a partition for the heaviest child of c, which is d, because its subtree has a weight of 4, while the subtree of f only has a weight of 1. The final result is the optimal partitioning as produced by DHW on the same tree.

This is plausible: The algorithm has, for each node, the choice to create a partition either for the right siblings of the node, or for its children one level below the current node. Hence, it sometimes can make exactly those choices that make the DHW algorithm superior to GHDW. The EKM algorithm is also memory-friendly, and very easy to implement, even easier than the original KM algorithm. The KM algorithm must consider a large number of children for each node, and sort them according to weight, whereas EKM only needs to select the heavier out of at most two children.

However, the algorithm is still a heuristic, and has its own problematic cases. A simple example can be seen in Fig. 9. For the tree shown on the right in n-ary representation, the optimal partitioning has two partitions and d,e are in the same partition as the root. On the left side we see the same tree in binary representation, and the partitioning created by EKM. EKM decides to create a partition for the subtree of the right child d of b, because the subtree with d and e is heavier than c. The result, with three partitions, is suboptimal.

5. OTHER ALGORITHMS AND RELATED WORK

Looking at an extreme end of the spectrum of available algorithms, BIN PACKING can be used to minimize the number of partitionings while completely ignoring the tree structure. However, BIN PACKING is NP-hard. Further, disregarding the tree structure may have severe negative consequences on performance, as closely related nodes be placed in different storage units.

Schkolnick [15] 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 clusters objects into base collections, which can be joined to efficiently answer queries. While this may be applied to join-based XML query processing, it does not solve our problem of finding weight-limited partitions.

Lukes [13] presents a linear time algorithm for tree partitioning that incorporates edge weights. It finds a partitioning of optimal value, e.g. one where the total weight of all edges that do not go across partitions are maximized. For unit edge weights, the algorithm solves the same problem as the Kundu and Misra algorithm: It finds a partition with minimal cardinality. The algorithm does not consider sibling partitioning.

Bordawekar and Shmueli [4] investigate Lukes' algorithm in the context of XML, and report runtimes of several hours on modern PCs for very small documents ($\sim$100K). They extend Lukes' algorithm by introducing several techniques to limit memory usage and improve runtime. This breaks the optimality, but achieves approximate partitionings whose value is quite close to the optimum for single-node intervals. They also do not consider sibling partitioning.

The strength of Lukes' algorithm and its extensions lies in their ability to optimize the partitioning for anticipated query workloads. However, in many environments it is unclear how to anticipate query workloads. In such cases, the default is to assume unit edge weights, in which case the algorithm will produce a partitioning of minimum cardinality, albeit without allowing siblings to share a partition.

6. EVALUATION

We have presented seven algorithms for tree sibling partitioning. We now evaluate them as partitioning algorithms for document insertion into an XDS. Only DHW generates an optimal result in the sense of our problem statement.

Our initial experiments are based on a simple main-memory implementation of the algorithms, to compare their "pure" performance without the overhead of integration into a real XDS. We compare the runtime of the algorithms, and the cardinalities of the produced partitionings. As it turns out, the EKM heuristic is surprisingly good, both in terms of result quality and runtime.

In a final experiment, we show the impact of sibling partitioning on query performance. To this end, we integrate the EKM and KM algorithms into Natix and evaluate the runtime of various queries.

6.1 Environment

The documents to partition were chosen from the University of Washington's XML data repository [14] and the XMark benchmark [16], the latter with a scaling factor of 0.1. Some of the documents, such as the XML representation of the `partsupp` and `orders` relations, have a very simple structure, while others, such as the Mondial data, represent nested structures with larger subtrees.

The documents are mapped into instances of our problem as follows. An ordered tree is constructed by parsing the documents and converting the parser's DOM [8] representation into a simple main-memory representation. This simple representation does not retain tag names of elements or node contents for text nodes and attribute nodes, but has a weight value for each node instead. Real-world storage engines typically align objects on secondary storage to some "slot" size for efficiency reasons. This is reflected in our weight value, which does not represent the node's size in number of bytes, but in number of "slots" it requires on disk. Each node is considered to use one slot for its metadata, such as tag name and node type. In addition, text and attributes nodes take up a number of slots proportional to the length of the node's content string. We use a slot size of 8 bytes.

We have implemented the algorithms in C++. The executables were compiled using g++ 3.3.5 with O3 optimization. Experimental results were obtained on a machine with 1 GB memory and an Intel Pentium IV CPU at 2.4GHz.

6.2 Number of Partitions

In our first experiment, we compare the number of partitions generated by the algorithms for the various documents, using a weight limit of $K = 256$ slots of 8 bytes, corresponing to a storage unit capacity of 2KB. The results are presented in Tab. 1. We also list, for each document, its size in the file system, number of nodes, and the total weight of the document tree (including alignment effects as explained in Sec. 6.1) divided by $K = 256$. The latter is a lower bound for the number of partitions if connectedness of the partitions is not enforced and every partition reaches maximum size.

It becomes obvious what can be gained by a tree sibling partitioning: For all documents, the sibling partitioning algorithms that were specifically designed for XML storage (DHW, GHDW, EKM, RS) produce a much smaller number of partitions than the KM algorithm, which only considers single node intervals. For the XML documents that represent relational data (`partsupp`, `orders`), the number of partitions is less than 10% of the KM result.

The DFS and BFS algorithms, although allowing siblings to share a partition, perform sometimes even worse than KM. Their top-down approach results in premature decisions about which nodes to put into a partition. Overall, these two algorithms are not very robust, as the quality of their result relative to the other algorithms varies greatly for the various documents.

Another observation is that the GHDW algorithm comes very close to the optimal result. For the data with relational structure (`partsupp.xml`, `orders.xml`), optimality is achieved. The difference between GHDW and the optimal result for the other documents is always below 4%.

The biggest surprise is the EKM algorithm, whose quality is also very close to the optimal result, although the algorithm is quite simple to implement. It even beats GHDW for the `uwm.xml` document by a margin of a few partitions. For the others, EKM is always the third-best algorithm.

6.3 Partitioning Time

The higher result quality of GHDW and DHW comes at a price, as shown in Tab. 2, which lists the algorithm runtime. In all cases, the time needed to partition a tree is much larger for DHW and GHDW than for the other algorithms. The document structure has a significant impact on the runtime of GHDW and DHW. For `uwm.xml` they both are much faster than for the equally large `partsupp.xml` document.

The simple structure of the EKM algorithm is reflected in the results: For all documents but the XMark document, EKMs runtime is below the precision of measurement.

				Algorithm						
Document	Size	Nodes	Weight/K	DHW (Optimal)	GHDW	EKM	RS	DFS	KM	BFS
SigmodRecord.xml	477KB	42054	352	382	384	402	405	1153	1294	2987
mondial-3.0.xml	1785KB	152218	1236	1358	1376	1407	1433	3268	11625	17312
partsupp.xml	2242KB	96005	1026	1083	1083	1091	1091	2282	15876	8192
uwm.xml	2338KB	189542	1446	1727	1790	1746	1817	4345	5449	11039
orders.xml	5379KB	300005	2247	2476	2476	2482	2482	5832	29876	15474
xmark0p1.xml	11670KB	549213	7532	8603	8838	8975	9631	25046	20519	42155

Table 1: Number of generated partitions

	Algorithm						
Document	DHW	GHDW	EKM	RS	DFS	KM	BFS
SigmodRecord.xml	24.83	0.28	<0.01	<0.01	<0.01	0.05	<0.01
mondial-3.0.xml	184.17	6.02	<0.01	<0.01	<0.01	0.11	0.02
partsupp.xml	474.13	5.55	<0.01	<0.01	<0.01	0.16	0.02
uwm.xml	401.38	1.18	<0.01	<0.01	<0.01	0.21	0.04
orders.xml	565.01	9.73	<0.01	<0.01	<0.01	0.35	0.07
xmark0p1.xml	2041.18	6.24	0.02	0.03	<0.01	0.63	0.11

Table 2: CPU time (in seconds)

DHWs performance makes it an unlikely candidate for use as an "online" document insertion algorithm. However, there might be applications where the optimal partitioning is important and partitioning can be done "offline" as part of physical reorganization. GHDW is faster (around 500KB/sec), and may be useful in an XDS if document insertion is rare, and a large number of queries can benefit from the smaller number of partitions, compensating for the partitioning effort.

In terms of quality and speed, EKM is the much better choice. It is faster by five orders of magnitude than DHW, with a very good quality of the result.

6.4 Query Performance

Based on the results for the pure main-memory implementation above, we have chosen to incorporate implementations of both EKM and KM into Natix [6].

Our claim is that on average, fewer storage units result in more navigation between nodes of the same storage unit, leading to a better query performance. The goal of this experiment is to verify this claim by comparing query evaluation times on storage layouts produced by the different algorithms. We want to focus on the performance of pure navigation operations when accessing the data. For this reason we have chosen a set of simple queries from the XPathMark benchmark (Q1–Q7 from [7]), which are evaluated against an XMark document of scaling factor 0.1.

We loaded the document into the XDS with both EKM and KM, using a size limit $K = 256$ that corresponds to storage units of 2KB. We executed the queries several times to increase precision, but do not include the time for the very first execution of each query to preload the buffer. We also use a buffer pool that is larger than the document, so that there is no page fault during query evaluation. We also measured the total amount of disk space used in Natix for the document for the two algorithms. This deviates somewhat from the theoretical results, as additional metadata is needed to maintain the on-disk structures, and fragmentation effects occur.

The results are shown in Tab. 3.

Remarkably, KM uses slightly less total space, because the smaller partitions (records in the Natix storage format) can be better placed by the record manager, which stores several records on a single disk page. For the larger partitions of the EKM algorithms, the record manager chooses to allocate a new page slightly more often, due to page fragmentation.

Query evaluation on the sibling partitioning produced by EKM outperforms evaluation on the KM partitioning for all queries. In some cases, the performance is improved by more than a factor of 2. This confirms our claim that sibling partitioning signifiantly improves the query performance of an XDS.

7. CONCLUSION

We investigated algorithms to efficiently solve tree partitioning problems for document insertion in XML Data Stores, where variable-size document nodes have to be assigned to storage units of limited capacity, such as records on disk pages. Motivated by current tree storage engine designs [2, 6], and previous work [10], we have based our work on the assumption that an important aspect of such tree partitioning is whether it allows siblings to be placed in the same partition even if their parent node belongs to a different partition. Further, navigation between nodes of the same partition is much cheaper than navigation between nodes of different partitions. Hence, our goal was to minimize the total number of partitions to optimize query performance.

We formalized this *tree sibling partitioning* problem, and studied its structure. We presented a dynamic programming algorithm called DHW that is capable of finding a minimal tree sibling partitioning in $O(nK^3)$ time in the worst case, where n is the number of nodes.

While this is an interesting result in its own right, and may be of use in other areas than XDSs, we also discussed concrete implementation issues for XDSs. The large resource requirements of DHW prompted us to investigate approximation algorithms that are better suited for integration in an XDS. In addition to adapting several existing algorithms to tree sibling partitioning, we introduced two novel heuristics, GHDW and EKM, based on our study of the optimal algorithm.

We implemented the algorithms and performed some experiments.

Query	KM	EKM
Total Occupied Disk Space	ca. 8192KB	ca. 8232KB
`/site/regions/*/item`	0.065	0.036
`/site/closed_auctions/closed_auction/annotation/ description/parlist/listitem/text/keyword`	0.033	0.023
`//keyword`	0.770	0.595
`/descendant-or-self::listitem/descendant-or-self::keyword`	0.344	0.262
`/site/regions/*/item[parent::namerica or parent::samerica]`	0.150	0.074
`//keyword/ancestor::listitem`	0.870	0.650
`//keyword/ancestor-or-self::mail`	0.854	0.607

Table 3: Query processing time (in seconds)

We demonstrated that sibling partitioning can reduce the total number of partitions by more than 90% compared to the optimal solution for partitions that are connected by parent-child edges only. Further, simple heuristics such as DFS or BFS provide inacceptable result quality. However, the algorithm for the optimal solution, DHW, is quite slow, and may only be applicable if document insertion is rare or the optimal partitioning is of critical importance. To a lesser extent, this is also true of the faster GHDW approximation algorithm. The EKM algorithm, a novel approximation algorithm based on the Kundu and Misra algorithm for tree partitioning, provides by far the best trade-off between result quality and runtime. EKM is now the default partitioning algorithm for the Natix system.

To validate our claim of better query performance of sibling partitioning, we ran some experiments using the Natix query processor. In some cases, queries can be executed twice as fast when EKM is used for partitioning compared to simple partitioning that does not consider sibling edges.

Acknowledgements We thank the anonymous referees for their suggestions, Simone Seeger for her help with preparing the manuscript, and Alexander Böhm for implementing some of the approximation algorithms.

8. REFERENCES

[1] A. Berglund, S. Boag, D. Chamberlin, M.F. Fernandez, M. Kay, J. Robie, and J. Siméon. XML path language (XPath) version 2.0. Technical report, World Wide Web Consortium (W3C) Working Draft, 2002.

[2] Kevin Beyer, Roberta J. Cochrane, Vanja Josifovski, Jim Kleewein, George Lapis, Guy Lohman, Bob Lyle, Fatma Özcan, Hamid Pirahesh, Normen Seemann, Tuong Truong, Bert Van der Linden, Brian Vickery, and Chun Zhang. System RX: One Part Relational, One Part XML. In *SIGMOD Conference*, pages 347–358, 2005.

[3] S. Boag, D. Chamberlin, M.F. Fernandez, D. Florescu, J. Robie, and J. Siméon. XQuery 1.0: An XML query language. Technical report, World Wide Web Consortium, November 2005. W3C Candidate Recommendation.

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

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

[6] Thorsten Fiebig, Sven Helmer, Carl-Christian Kanne, Guido Moerkotte, Julia Neumann, Robert Schiele, and Till Westmann. Anatomy of a Native XML base management system. *VLDB Journal*, 11(4):292–314, 2003.

[7] Massimo Franceschet. XPathMark: An XPath benchmark for the XMark generated data. In *XSYM'05 Workshop*, volume 3671 of *LNCS*, pages 129–143, Berlin, August 2005.

[8] Arnaud Le Hors, Philippe Le Hégaret, Lauren Wood, Gavin Nicol, Jonathan Robie, Mike Champion, and Steve Byrne. Document object model (DOM) level 2 core specification. Technical report, WWW Consortium (W3C), 2000.

[9] Carl-Christian Kanne and Guido Moerkotte. Efficient storage of XML data. In *ICDE Conf.*, page 198, 2000.

[10] Carl-Christian Kanne and Guido Moerkotte. The importance of sibling clustering for efficient bulkload of XML document trees. *IBM Systems Journal*, 45(2), 2006.

[11] Carl-Christian Kanne and Guido Moerkotte. A linear time algorithm for optimal tree sibling partitioning and its application to XML data stores. Technical Report TR-2006-006, University of Mannheim, Department for Mathematics and Computer Science, 2006.

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

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

[14] University of Washington. XMLdata repository. http://www.cs.washington.edu/research/xmldatasets/, 2004.

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

[16] Albrecht Schmidt, Florian Waas, Martin Kersten, Michael J. Carey, Ioana Manolescu, and Ralph Busse. XMark: A benchmark for XML data management. In *VLDB Conf.*, pages 974–985, 2002.

[17] Manolis M. Tsangaris and Jeffrey F. Naughton. On the performance of object clustering techniques. In *Proc. SIGMOD Conference*, pages 144–153, 1992.