Clustering Techniques for Minimizing External Path Length

A. A. Diwan **Sanjeeva Rane** **S. Seshadri** **S. Sudarshan**
Department of Computer Science and Engineering
Indian Institute of Technology
Bombay 400076, India
{aad,rane,seshadri,sudarsha}@cse.iitb.ernet.in

Abstract

There are a variety of main-memory access structures, such as segment trees, and quad trees, whose properties, such as good worst-case behaviour, make them attractive for database applications. Unfortunately, the structures are typically 'long and skinny', whereas disk data structures must be 'short-and-fat' (that is, have a high fanout and low height) in order to minimize I/O.

We consider how to cluster the nodes (that is, map the nodes to disk pages) of main-memory access structures such that although a path may traverse many nodes, it only traverses a few disk pages. The number of disk pages traversed in a path is called the external path length. We address several versions of the clustering problem. We present a clustering algorithm for tree structures that generates optimal worst-case external path length mappings; we also show how to make it dynamic, to support updates. We extend the algorithm to generate mappings that minimize the average weighted external path lengths. We also show that some other clustering problems, such as finding optimal external path lengths for DAG structures and minimizing weights for optimal height mappings, are NP-complete. We present heuristics for these problems.

We present a performance study (using quad-trees on actual image data as an example) which shows that our algorithms perform well. Our algorithms can also be applied for clustering complex objects in object-oriented databases.

Proceedings of the 22nd VLDB Conference Mumbai(Bombay), India, 1996

1 Introduction

In earlier generation databases, access structures were specially designed taking the page structuring of disks into account. For instance, B-trees have nodes whose size is a page. As database requirements grew to handle complex data types, the need was felt for specialized data structures, such as spatial index structures. Disk based spatial data structures such as R-trees [Gut84] and variants such as R* and R^+ trees [SRF87, BKSS90] were developed for this task.

However, there are a variety of specialized applications for which main-memory access structures exist but not good disk based access structures. Various forms of quad-trees, which were developed for in-memory use, have been found useful in spatial database systems/geographic information systems such as Quilt [SSN90]. Although disk versions of some kinds of quad trees have been developed, only in-memory versions have been developed for some variants of quad trees; see Section 3. Interval trees have been proposed for handling time intervals in temporal databases [KTF95]. Versioning of data structures is important in temporal databases, and versioning techniques developed for main-memory structures [DSST86] have been found to be useful in temporal databases [KTF95]. The two-dimensional index structure of [RS94] also demonstrates the importance

of using data structures developed for main-memory, in database systems. Furthermore, object-oriented database systems make it easy to reuse code developed for in-memory data structures in a persistent setting.

Unfortunately, while the main-memory structures have very attractive worst-case properties in memory, they cannot usually be used directly to store data on disk. Main-memory access structures are typically constructed as a set of dynamically allocated nodes linked by pointers. To perform a lookup on such a structure requires following one or more chains of pointers. Following fairly long chains of pointers is acceptable for data structures in memory, so "tall-and-skinny" structures such as binary trees are acceptable in memory. In the disk versions however, the corresponding page structure must however be "short-and-fat" in order to minimize I/O; for instance disk access structures such as B-trees have a high fanout, such as 100, and are short (perhaps 3 or 4 pages high).

If we want to use an index structure designed for main-memory in a persistent environment where data must be stored on disk, we have two options — either redesign the index structure to give it a very high fanout, or map the nodes carefully to disk pages such that although a path may traverse many nodes, it only traverses a few disk pages even in the worst case. The first option has a high cost in terms of algorithm design and implementation, but the cost is worthwhile for very commonly used data structures as has been proved by B-trees and R-trees.

The second option, namely careful mapping of nodes to disk pages, is attractive for several reasons such as to re-use existing code and avoid the cost of designing new structures. The *clustering* or *pagination* problem for a set of nodes is the question of how to allocate the nodes to disk pages to optimize some metric. An appropriate metric is the number of page I/O's required for performing a point lookup in an access structure assuming no pages are in-memory initially. We consider traversals on acyclic structures that start at a source (root) and go up to a sink (leaf). For a given mapping, we define "external path length" of a traversal to be the number of I/O's required for the traversal assuming no pages are in buffer initially, and the buffer holds only one page. For a given mapping, the "page-height" of an acyclic access structure is then defined as the maximum of the external path lengths over all possible traversals on the access structure. Our goal is to find a mapping that minimizes the "page-height" of access structures and this mapping is termed an "optimal" mapping.

In this paper we describe and address various versions of the pagination problem. The specific contributions of this paper are as follows:

1. In the static version of the pagination problem, an access structure which is a tree (or a rooted DAG) is given, and an optimal mapping has to be found.

   (a) We present a pagination algorithm for tree structures that generates optimal mappings.

   (b) We extend the above algorithm to generate mappings that minimize the average of the external path lengths with given access frequencies for each leaf node.

   (c) We show that finding optimal mappings for DAG structures is NP-complete. We present efficient heuristics for this problem.

2. In the dynamic version, we have an access structure whose nodes have already been allocated to pages in an optimal fashion, and the problem is to incrementally update the mapping in the face of an update to the access structure to maintain the optimality of the mapping.

   We present an algorithm for incrementally updating the optimal mapping on a tree structure. We show that the resultant mapping is equivalent to running the static mapping algorithm on the final tree. The algorithm works under a very general update model, and besides it only examines pages in the locality of affected pages, keeping the cost of reorganization small.

3. We show that the problem of finding a minimum weight (number of pages that have been mapped into) mapping amongst the optimal mappings is NP complete, but present heuristics which can guarantee at least a 50% node occupancy. Our performance study indicates that actual occupancies are around 75%.

4. We present a performance study (using quadtrees on actual image data) that shows that the the optimal mapping algorithm results in "page-heights" that are 33% to 60% of the worst case external path lengths generated by preorder clustering (linearly ordering nodes according to their preorder number and then sequentially assigning as many nodes to a page as is possible). Moreover, the average of external path lengths (with all leaf nodes equally likely to be accessed) using our basic pagination algorithm is better than the average of external path lengths of pre-order clustering by similar margins.

   We show that our merging heuristics result in occupancies of about 75%. Even on pre-order traversal (for which pre-order clustering is optimal) our algorithms result in a performance loss

of only around 30%, assuming all nodes in a path from the root to a leaf can fit into the buffer.

Another important application of the pagination problem is in object-oriented databases, where programmers tend to program much like they do with in-memory data structures, but the actual nodes are allocated on disk pages. OODBs allow special purpose access structures to be made persistent very easily. However, an intelligent mapping of nodes to disk pages is important for reducing access costs. For OODB data-structures similar to indices that require traversals from a root (or set of roots) to a leaf, minimizing the external path length in the paginated version of the structure helps in reducing access time. There has been work on clustering of data in OODBs based on access frequencies (see Section 3). However, to our knowledge our work is the first to address the issue of minimizing external path lengths, which is important for access structures even in an OODB.

2 Problem Definition

Suppose we are given an access structure consisting of nodes and directed links between nodes. We assume that the links are acyclic, and the access structure is rooted. (Non-rooted DAGs can be easily made rooted by adding a pseudo-root.) We define the problem for the general case where nodes have varying sizes, but the special case where all nodes have the same size is also of considerable interest.

Let N be the set of nodes, and B a set of pages, where $\mid B \mid \leq \mid N \mid$. Let $S(n)$ denote the size of node n, and let k denote the size of a page; we assume that that all nodes are smaller than a page. Let M be a mapping $N \rightarrow B$. A *legal mapping* M is one such that for every page $b \in B, (\sum_{n \in M^{-1}(b)} S(n)) < k$ That is, the sum of the sizes of the nodes mapped to a page does not exceed the size of the page.

Given a legal mapping M, and a sequence of nodes $\mathcal{P}$ corresponding to an external path (path from a source to a sink) in the access structure, let n be the number of nodes in $\mathcal{P}$ and let us denote the j^{th} node in the sequence by $\mathcal{P}[j]$, $1 \leq j \leq n$. Then define the *external path length* of $\mathcal{P}$ under M, $length_M(\mathcal{P})$, as

$$1 + \mid \{j : M(\mathcal{P}[j]) \neq M(\mathcal{P}[j+1]), 1 \leq j < n\} \mid$$

In other words, $length_M(\mathcal{P})$ is the number of pages I/O's that have to be incurred to follow path $\mathcal{P}$ under mapping M assuming a buffer size of 1.

Given a set of nodes N and a legal mapping M, the *weight* of the mapping is the number of pages that have at least one node from N mapped to them (formally, the cardinality of the image of N under M).

Given a tree or a DAG structure A and a mapping M, the maximum of $length_M(P)$ over all external paths P in A is called the *page height* of A under M. We similarly define the page height of a node in a rooted tree to be the the number of pages I/O's that have to be incurred to follow the path from the root to the node assuming a buffer size of 1. It will be clear from the context which page height we are referring to. Given an access structure A, the *height minimization* problem is to find a legal mapping M that minimizes the page height. Such a mapping is called a *height minimal* mapping. The *height-weight minimization* problem is to find the mapping of minimum weight among the height minimal mappings. The *weight minimization* problem is to find a legal mapping with the minimum weight. The *weight-height minimization* problem is to find a legal mapping that minimizes the page height amongst those mappings that are of minimum weight.

We say that two mappings are *equivalent* to each other if one can be obtained from the other by a renumbering of the pages. Formally, two mappings f and g from a set of nodes N to a set of pages $B = \{1 \ldots \mid B \mid\}$ are equivalent if there is a permutation π on $1 \ldots \mid B \mid$ such that $\forall n \in N, f(n) = \pi(g(n))$.

3 Related Work

Bannerjee et al. [BKKG88] describe a technique for clustering DAGs for CAD databases, which extends earlier work by Schkolnick [Sch77]. Their approach is to perform a traversal of the DAG structure, such as a DFS, a BFS or a modified DFS called child-DFS, in order to get a linear ordering on the structure. The linear ordering is then mapped to a B-tree. They describe how to maintain the mapping in the face of updates to the DAG structure. While their DFS based clustering is good for pre-order traversals of the DAG structure, it is not good for index traversals that follow a path from a root to a leaf. Right-most paths are particularly hard hit, and each node along such a path could be mapped to a different page, giving the tree or DAG structure a bad page height. Thus, worst case external path lengths could be high with their clustering technique.

Numerous tree structures have been proposed to handle different kinds of spatial data, such as points, lines and regions, and to efficiently handle specific types of queries, such as point lookups or region lookups (see, e.g., [Sam95]). Most of these are designed primarily as in-memory structures. An example is the quad-tree, which partitions space in the same way for all data sets[1] making it easier to perform spatial joins

[1] The depth to which the partitioning goes depends on the data.

with a quad-tree than using an R-tree. Quad trees of various flavors have been proposed to handle different kinds of data, especially in geographic information systems (see, e.g., [Sam95]).

Work more closely related to ours deal with mapping specific index structures to disk. Shaffer and Brown [SB93] describe a quad-tree structure that they call a "leaf-less quad-tree", and a way to map its nodes to disk pages in a pre-order fashion. They show that their structure occupies less space and is faster than the more commonly used linear quad-tree structure, which represents a quad-tree as a list of leaf nodes, stored in a B-tree. Their mapping is very similar to that of Banerjee et al. [BKKG88], and has the same drawbacks for simple lookups, although the pre-order traversal mapping is good if the entire tree is to be scanned. Other approaches to mapping quad-trees to disk have been based on linearizing space using space filling curves such as z-orders and Morton orders (see, e.g. [Jag90]). Linearizing orderings preserve spatial locality fairly well, but like pre-order traversals, while the mappings work well for some kinds of traversals and queries [Ore89, Ore90], they do not have good worst case properties for path lookups.

A more closely connected paper is that of Ramaswamy and Subramanian [RS94], which presents results on optimal pagination of certain extensions of binary trees such as segment trees and priority search trees. They present some interesting results on replication of data associated with internal nodes (which they call path-caching), in order to reduce page traversals. However, their technique for mapping nodes to pages simply maps sub-trees of height $log_2(B)$ to a page, where B is the number of nodes that can fit in a page. While this technique works reasonably well with *full* binary trees, clearly it can result in many partially filled nodes if the tree is not full, and the resultant trees will not necessarily be of minimal height.

Although all the above mentioned work is related to spatial databases, we would like to stress that our techniques are general purpose techniques, and not targeted specifically at spatial databases. Therefore we do not attempt a direct comparison with any mapping technique specially designed for a particular application.

Another related work is that of Kumar et al. [KTF95], which describes the design of access structures for bitemporal databases. Instead of using segment trees, they paginate interval trees, which were also developed for in-memory use. This work as well as [RS94] clearly demonstrate the importance of extending in-memory structures to work well on disk. However, just like [RS94], their approach to mapping nodes to pages simply consists of mapping subtrees to a page. Each page is assigned a subtree of equal size, starting from the root of the tree. However, these mappings do not provide good worst-case guarantees; using trees of equal height is too pessimistic for normal tree structures since it assumes that the trees are balanced, while using trees of equal size does not provide good guarantees about the page height of the resultant structure.

Nodine et. al. [NGV93] consider blocking (clustering) and paging strategies for undirected graphs, including blocking strategies that replicate nodes. They present lower and upper bounds on the speedup (that is, reduction in I/O) that can be achieved, for traversals of (possibly cyclic) paths in the graph. Unlike the approach in this paper, their notion of optimality are only asymptotic. Further, for the case of trees, their results are useful (even asymptotically) only with replication of nodes.

There has been a lot of work on clustering in the area of object-oriented databases, but none of these addresses the issue of worst-case path lengths for access structures. Tsangaris and Naughton [TN91, TN92] have a stochastic approach to the problem of clustering. They use an object graph with edges between objects indicating the probabilities of accessing one object after accessing the other. Their approach to clustering is to use a minimum weight partitioning of the graph such that each partition fits in a page. While their approach has advantages on arbitrary graph structures, it does not make use of more specific traversal information associated with access structures. Their clustering algorithm can result in bad page heights for such structures. For example, on a full binary tree, with all leaves having equal probability of access, every edge will have the same weight in their stochastic model. The clustering technique of Cheng and Hurson [CH91] also uses weighted graphs, and suffers from the same problem as above when applied to access structures. Other work on clustering in object-oriented databases includes [CK89, LS92]. None of these addresses the issue of worst-case path lengths for access structures.

4 Tree Structures

In this section, we assume the access structure is a tree. We consider the case when the access structure is a DAG later. Section 4.1 addresses the static case, where the tree is given and its nodes have to be mapped to pages, and Section 4.2 addresses the dynamic case where a tree that has already been mapped is changed. Section 4.3 addresses minimization of average number of page traversals over all paths from the root to leaves. Finally, Section 4.4 addresses the weight minimization problem.

4.1 Static Height Minimization

The algorithm for finding the height minimal mapping is given below:

```
Algorithm Bottom-Up-Tree-Clust(T)
  Input: Tree T whose nodes are to be clustered
  Output: Mapping of nodes to pages
  /* Process tree T bottom-up as follows */
  While there are nodes in T not yet processed
    Choose a node P that is either a leaf or all of
      whose children have already been processed
    process_node(P)

Procedure process_node(P)
  If P is a leaf node create a new page C containing
    node P.
  Else { /* P is an internal node */
    Let P_1 ... P_n be the children of P.
    Let C_1 ... C_n be the pages containing P_1 ... P_n
      respectively.
    Let P_i1 ... P_im be the children among the above
      whose page height is the greatest.
    If node P and the contents of the pages C_i1 ... C_im
      can be merged together in a single page
      /* there is enough space */
    Then merge the contents of C_i1 ... C_im into a
      new page C and delete C_i1 ... C_im.
    Else create a new page C containing only P
    /* Child pages not merged with their parents above
      are completed */
  }
```

Theorem 4.1 *The mapping of nodes generated by Algorithm Bottom-Up-Tree-Clust is a height minimal mapping.* □

Proof: In the description of Algorithm Bottom-Up-Tree-Clust, let P be a node in tree T visited during the bottom-up traversal, and let C be the page it is mapped to. The proof follows once we prove the following claim.

Claim: The page mapping for the nodes in the subtree rooted at P (a) has the minimum page height over all mappings of the nodes in the subtree, and (b) among all mappings with minimum page height, page C has the lowest utilization.

We prove the claim by induction on the height of node P in T. The base case is for height $= 1$, in which case P is a leaf node and is mapped to a page containing only itself thus proving the claim. For the induction step, suppose the claim is true for all the children $P_1 \ldots P_n$ of node P.

Case 1: P is merged with $C_1 \ldots C_m$ (as described in the algorithm).

Case 1.1: The page height of P is not minimum.

Let the page height of P in the bottom-up solution be h_1 and let the minimum page height of P be $h < h_1$. Therefore the page height of the children $P_1 \ldots P_m$ in the bottom-up solution is also h_1 which, by the induction hypothesis is the minimum possible. Considering the mapping M in which P has page height h and restricting it to the nodes of the subtree rooted at P_1, we get a mapping in which P_1 has page height $\leq h < h_1$, contradicting the induction hypothesis.

Case 1.2: The tree is of minimum height $= h_1$, but the utilization of C is not the least.

Let $S_1 \ldots S_m$ be the sum of the sizes of the nodes in the pages $C_1 \ldots C_m$ in the bottom-up solution. The total utilization of the page C is the bottom-up solution is thus $S_1 + S_2 + \ldots + S_m + \text{size}(P)$. Suppose there is a mapping M in which P has page height h_1 but the page C containing P has smaller utilization.

Obviously P needs to be included in C. All the children $P_1 \ldots P_m$ of P must be mapped to C — otherwise for some i, $1 \leq i \leq m$, the page height of P_i is $< h_1$ - a contradiction with the induction hypothesis.

If we restrict M to the subtree rooted at P_i, we get a mapping of the nodes in this subtree such that P_i has page height h_1 and P_i is contained in page C. By the induction hypothesis, the sum of sizes of nodes in the subtree rooted at P_i which are mapped to C must be at least S_i. Thus the sum of sizes of nodes mapped to C by M must be at least $S_1 + S_2 + \ldots + S_m + \text{size}(P)$, contradicting the fact that the utilization of C is less than that in the bottom-up solution.

Case 2: P is not merged with $C_1 \ldots C_m$ by the algorithm.

Case 2.1: The page height of P is not minimum.

Arguing exactly as in case (1.2), we can show that if there is a mapping M in which P has the same page height (h_1) as $P_1 \ldots P_m$, then the sum of sizes of nodes in the page C containing P must exceed k, a contradiction. Thus the minimum page height for P is $h_1 + 1$, as obtained by the bottom-up algorithm.

Case 2.2: The tree is of minimum height, but the utilization of page C is not the least.

This case is trivially false, since P needs to be in C and and no node other than P is mapped to C.

Hence the mapping obtained by the bottom-up algorithm has minimum page height. □

The Bottom-Up-Tree-Clust algorithm generates height minimal mappings, but may generate pages with low occupancy. It is easy to improve space occupancy to at least 50% by arbitrarily merging nodes that are less than half-full. Merging nodes in such a manner does not increase the external path length and is therefore safe.

One point to note with arbitrary merging of nodes is that it can create cycles of the following form: a node in one page points to a node in a second page and so on, till a node in the nth page points to some other node in the first page. Although the nodes form an acyclic structure, there is a cycle at the level of pages which may be undesirable under certain circumstances. A simple way to avoid cycles is to only merge pages whose root nodes are at the same page height. No page can then be merged with an ancestor, and all pointers from a page will only be to pages whose nodes are at a lower page height. Hence there will be no cycles. The heights of the root nodes of pages are already available since they are used to compute the node mappings.

4.2 The Dynamic Case

Updates to tree structure must start at the root and follow one or more paths. The tree after the update is related to the tree before the update as follows: some of the nodes in the paths followed by the update, are deleted, inserted, or updated. The relevant updates here are changes to the pointers in the nodes. Subtrees of the original tree that are not in the path of the update remain unchanged.

In our model of updates, we require two functions, *new_node(node)* that takes a node in the new sub-tree and returns true if it is a newly inserted node and false otherwise, and a function *old_subtree(node)* that takes a node in the old tree and returns true if the node and its sub-tree were not modified by the update, and false otherwise. We call a node an *affected* node if either it is a *new_node* or it is an old node for which *old_subtree* is false. We call a node a *fringe node* if it is not *affected* but its parent is *affected*. We assume a function called *fringe(node)* exists that returns true if the node is a fringe node, false otherwise. We also assume that each node deleted is maintained in a list called the delete-list. More specific details of the update are not required. Thus our model of updates is very general, and not tied down to any specific access structure.

The clustering algorithm described in the last section lends itself to dynamization since it is bottom-up. The result of clustering will not change for any subtree that is not affected by the update. However, the root of a particular subtree which has not been affected may have been clustered with its parent and siblings. A change to its parent or sibling could cause this clustering to be invalid. Therefore all that is required is to undo the clustering on any node for which either *old_subtree(node)* is false or the node is a *fringe* node in the same page as its parent, and restart the bottom-up algorithm on the new nodes, and nodes whose clustering has been undone.

```
Algorithm Recluster-BottomUp(T)
  Input: Tree T and functions new_node and old_subtree.
    List of deleted nodes, delete_list.
  Output: Update of node mapping to pages
  S = { };
  For each node n in delete-list
    Remove n from its current page
    If n is the last node in the page
      Then delete the page.
  decluster(root);
  /* S is now the set of nodes that are affected */
  While there are nodes in S that are not yet processed {
    Choose an affected node P that is either a leaf or
      all of whose children are either not in S,*
      or have been processed
    process_node(P);
  }
procedure decluster(node n)
  add n to S
  if (!new_node(n)) {
    remove n from its current page.
    if n is the last node in the page
      Then delete the page.
  }
  for each child n1 of n {
    if (new_node(n1) or ! old_subtree(n1) )
      Then decluster(n1)
    else /* n1 is a fringe node */
      if (n1 is in the same page as n)
        move n1 and all its descendants in the same
          page as n to a new page
  }
```

Theorem 4.2 *The mapping of nodes to pages generated by executing Algorithm Recluster-BottomUp after an update to a tree is equivalent to the mapping generated by Algorithm Bottom-Up-Tree-Clust when applied to the tree after the update.* □

An important property of the above algorithm is that it only reads and writes pages in the access path of the updates, plus some of the pages containing fringe-nodes. Thus reorganization is localized.

Each page can be merged with an adjacent page (in terms of the order in which pages are completed) if their combined occupancy is less than 100%, as mentioned before. This technique works even in the dynamic case, and still guarantees 50% occupancy.

4.3 Average Path Length Minimization

In this section, we consider the problem of finding a mapping from nodes to pages which minimizes the av-

erage external path length for tree structures. In other words, we want to find a mapping M from nodes to pages which minimizes

$$\sum_{l \in leaf\ nodes} f(l) * length(P(l))$$

where $P(l)$ is the path from the root to the leaf l and $f(l)$ is the given access frequency of leaf node l. We call this quantity the total external path length for the mapping M.

We assume that all nodes have the same size, and therefore without loss of generality assume the size is 1. In the general case with nodes of different sizes, the problem can be easily shown to be NP-Hard by reducing the knapsack problem to it.

Our algorithm is essentially an application of dynamic programming. We assume the subtrees of a node P in the tree T are ordered in some fashion as we would like to consider them in some order (Note the actual order itself is not important but we only need to order them for convenience). For every node P in the tree T, we compute the quantities $S[P, i, j]$ in a bottom-up manner, where $S[P, i, j]$ denotes the minimum total external path length for the tree rooted at P and containing only the first i subtrees of P, given that the page containing P has exactly j nodes mapped to it. Here, $0 \leq i \leq n$ and $1 \leq j \leq k$ where n is the number of children of P and k is the size of a page. We also denote by $S_1[P, j]$ the minimum total external path length in the subtree rooted at P, given that the page containing node P has exactly j nodes mapped to it. Thus $S_1[P, j] = S[P, n, j]$. Let $S_2[P]$ be the minimum total external path length in the subtree rooted at P, $S_2[P] = min_{1 \leq j \leq k} S_1[P, j]$. Let $F[P]$ be the sum of the access frequencies of the leaves in the subtree rooted at P.

The main idea behind the algorithm is to find the minimum total external path length in the subtree rooted at each node assuming that the page containing the root node has exactly j nodes mapped to it. Unlike the bottom-up algorithm for the minimum height mapping, it is not sufficient to just compute the minimum total external path length mapping for each subtree, since it is possible that a suboptimal mapping for a subtree leads to an optimum mapping for the tree rooted at the parent. This happens as the page utilization in the page containing the root may be smaller in a suboptimal mapping thus allowing more mergings and greater reductions in total external path length for the tree rooted at the parent.

Also the j nodes contained in the top page may be distributed arbitrarily amongst the subtrees of the root. If we do this naively, it would lead to an exponential time algorithm. However, we can avoid this by finding the best way of distributing the j nodes in the first i subtrees of the root. This is done for all values of j, $1 \leq j \leq k$, and for all values of i, $1 \leq i \leq n$ where n is the number of children of the node. Thus, when considering the $i + 1^{th}$ child, we assume that some m of the j nodes in the page containing the root belong to this subtree and the remaining $j - m$ nodes are distributed amongst the first i subtrees. Since we have already computed the best solution with these distributions, we can find the value of m which minimizes the total external path length subject to these restrictions. These computations can be represented succinctly by recurrence relations as shown below.

We compute the quantities $S[P, i, j]$, $S_1[P, j]$, $S_2[P]$ and $F[P]$ for all nodes in the tree by traversing the tree in a bottom-up fashion. For a given node P, we compute the values of $S[P, i, j]$ in increasing order of i for all values of j in increasing order.

For a leaf node l, we have $S[l, 0, 1] = S_1[l, 1] = S_2[l] = f(l)$ and $S[l, 0, j] = \infty$ for $2 \leq j \leq k$. Also $F[l] = f(l)$. Let P be any internal node in the tree and let $P_1, \ldots P_n$ be the children of P. Now, we define that $S[P, 0, 1] = 0$ and $S[P, 0, j] = \infty$ for $2 \leq j \leq k$. We compute $S[P, i, j]$ for $i > 0$ using the following recurrence:

$$S[P, i, j] = min(A, B)$$

where

$$A = S_2[P_i] + F[P_i] + S[P, i - 1, j]$$

and

$$B = min_{1 \leq m < j}(S_1[P_i, m] + S[P, i - 1, j - m])$$

Once the $S[P, i, j]$ have been computed for all $0 \leq i \leq n$ and $1 \leq j \leq k$, we can easily find $S_1[P, j]$ and $S_2[P]$. Also $F[P]$ can be computed easily as $F[r] = \sum_{1 \leq i \leq n} F[P_i]$. The value of $S_2[R]$ for the root node R of the tree T gives the minimum total external path length for the original tree T.

To find the actual clustering, we keep track of which quantity in the recurrence gives the minimum and cluster accordingly after knowing the value of the optimal solution. Thus if for the root node R, $S_2[R] = S_1[R, j]$ then the page containing the root node contains j nodes. To find how many of these belong to the n^{th} subtree, we look up the value of m which gives the minimum value for $S[R, n, j]$ in the recurrence, then put m nodes from this subtree in the top page. These can be identified by a similar procedure applied recursively to the n^{th} child. The remaining $j - m$ nodes in the top page can be identified by applying a similar procedure to the first $n - 1$ subtrees of the root.

The time complexity of this algorithm can be easily seen to be $O(k^2 N)$ while the space complexity is

$O(kN)$ where N is the number of nodes in the original tree. The algorithm can be easily extended to the general case. Assuming, the input sizes are encoded in unary, the algorithm would be polynomial.

Theorem 4.3 *The clustering generated by the above recurrence equations has minimal average external path length, with the given access frequencies $f(l)$ for each leaf node l.* □

Although the above algorithm is polynomial in k and N, k can be quite large (say, 1000), leading to a fairly high cost. Therefore, the following heuristic may be of use: instead of trying each value of occupancy from 1 to k in the dynamic programming algorithm, try only a smaller number of occupancies, going up in steps.

The dynamic programming in the above algorithm can be extended to give the mapping with minimal average external path length *amongst* mappings that are height-minimal.

To the matrix $S[P, i, j]$ we add another dimension, h. Now, $S[P, i, j, h]$ denotes the minimum total external path length for the tree rooted at P and containing only the first i subtrees of P, given that the page containing P has exactly j nodes mapped to it, under the condition that the page height is less than or equal to h. The matrix S can be computed bottom-up, as before; we omit details.

4.4 Weight Minimization

In this section, we consider the issue of *minimizing* the number of pages in the mapping of a tree, which we call *weight minimization*. If we ignore the page height, finding a mapping that minimizes weight, is trivial when all nodes are of the same size. However, the problem is not very interesting, since we are interested in reducing the height.

The height-weight minimization problem (namely, finding legal mappings of minimum weight among those of minimum height) is less straightforward. Interestingly, this problem is NP-complete even in the special case where all nodes have the same weight, as shown in the following theorem.

Theorem 4.4 *The height-weight minimization problem is NP-complete even if all nodes are of the same size.*

Proof: We prove the NP-Completeness by reducing bin-packing to this problem. In the bin-packing problem, we are given a set of n objects $x_1, \ldots, x_n$ with object x_i having a size S_i and bins of size k and have to find the minimum number of bins necessary to pack all the n objects such that the sum of sizes of objects in a bin does not exceed k. We note that the decision version of the bin-packing problem (deciding whether a given number of bins suffice), is strongly NP-complete and the input numbers can be assumed to be encoded in a unary representation. Moreover we can also assume that $n \leq k$ since we can always multiply all the object and bin sizes by a suitable constant without changing the solution.

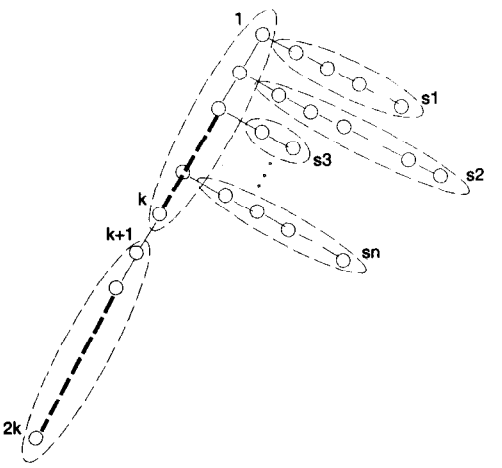

Figure 1: Mapping of Bin Packing

Now, given an instance of bin-packing satisfying these conditions, we construct a binary tree as shown in Figure 1. The tree has a left branch of height $2k$, and the first n nodes on the left branch have right subtrees that are right-most branches; the right branch of the ith node from the root has length S_i. Let the size of each node be 1 and the size of a page be k, and let $n \leq k$. The minimum height mapping for this tree clearly has to be of height at least 2 since there is a path of length $2k$. In fact it is easy to construct a mapping of height 2 as shown by the clustering in the figure. It is also easy to see that the nodes along the left-most path must be clustered together in two groups of k in any minimum height mapping.

From this it follows that the nodes along each right branch of the tree must be clustered together in any minimum height mapping. If they are not, the height of the tree would be at least 3.

Now it is easy to see that the bin-packing problem has a solution with m bins iff there is a minimum height mapping of the nodes in the tree with weight $m + 2$.

Showing that the decision version of the problem is in NP is straightforward. □

If node sizes differ, there is an even simpler reduction of the bin packing problem which shows that finding even the minimum weight mapping is NP-complete.

Since the height-weight minimization problem is

hard, we are forced to use heuristics for the problem. The basic problem is to merge pages of the height-optimal clustering to minimize weight which is again equivalent to bin packing. So we look at heuristics for this merging. The motivation for the heuristics that we study is that the merging should reduce the number of I/O's required for performing pre-order traversal (this could be the equivalent of a scan for some main memory index structures) and ease of implementation. The heuristic *pre-order merging* algorithm performs a pre-order traversal of the tree and merges pages in the order in which they are first encountered. At each stage there is a current page. If the contents of the next page and the current page will fit in a page, the next page is merged into the current page. Otherwise the next page becomes the current page, and the traversal continues. We consider another heuristic called the *previous merging* which is to merge pages in the order in which they are completed (see Procedure process_node). This can easily be incorporated into the clustering algorithm without requiring an extra pass over the tree. We study the performance of these heuristics in Section 6.

5 Minimum Height Mappings for DAG Structures

We presented a linear time algorithm for finding the minimum height mapping for tree structures. The algorithm cannot be used on DAG structures, since as we proceed upwards from a node we may have to choose which of two or more parents to merge into the same page as the node, and there is no obvious way to choose among the parents. In fact, we show that the problem is NP-complete for a DAG.

Theorem 5.1 *Finding the minimum height mapping of a DAG structure is NP-Hard.* □

Since the page-height minimization problem is NP complete for DAG structures, we are forced to look at heuristics in order to get an efficient technique for mapping nodes to pages. We consider a heuristic that is a simple modification of the bottom-up optimization algorithm for tree structures. The difference from the tree case is that a node may have multiple parents and it is not clear which one to cluster the node with. The *deepest-node-first* heuristic chooses the parent which has the longest path from a root of the DAG. The algorithm implementing the above heuristic is as follows:

Algorithm Heuristic-DAG-Clust(D)
Input: DAG D whose nodes are to be clustered
Output: Mapping of nodes to pages

1. Order the nodes in the DAG such that if a node a precedes node b, the longest path from a root to a is no smaller than the longest path from a root to b.
2. For each node n in the DAG in the above order
   process_node(n) ;
   /* see Algorithm Bottom-Up-Tree-Clust */

The ordering of the DAG nodes in the first step of the algorithm can be generated easily by a small modification of the standard topological search algorithm. Due to the order in which we consider nodes, each node is considered after all its children have been considered. If a node has multiple parents, one of the parents with maximum depth from a root is considered first, and the page containing the node is either merged with the pages containing its siblings from that parent or is left unchanged. In either case, the page may later be merged with pages containing siblings from another parent.

6 Preliminary Performance Evaluation

We present results of a preliminary performance study of our algorithms. We implemented the tree clustering algorithm which provides the height minimal mapping, and the page merging heuristics that we introduced in Section 4.4, namely, the pre-order merging and the previous merging. The merge techniques increase occupancy and reduce the cost of pre-order traversal without affecting the worst-case page height. We would like to emphasize that the study limits itself to studying the feasibility of the techniques we have proposed.

The experiments were performed using a pointer-based quad tree implementation, and tests were run on two actual image data sets, shown in Figure 2. Both images are pixel maps where a pixel occupies 1 byte of storage. The first, which we call *landuse* represents land usage data, and consists of a $400x285$ pixel map. The quadtree representation of the data requires about 212 KB of storage, and had 9 levels of nodes. The other, which we call *ban*, is an aerial photograph of a region, and consists of a $256x256$ pixel map; the quadtree representation of this dataset occupies 1.2 MB of space (the larger size is because of a lot of variation within the image), and has 8 levels of nodes. In all cases, pointers were assumed to occupy eight bytes of data.

We would like to repeat here that our techniques are meant to deal with arbitrary access structures, not just with quad trees or spatial data structures. Therefore we do not attempt a comparison with spatial access techniques, such as R-trees or linear quad-trees, which

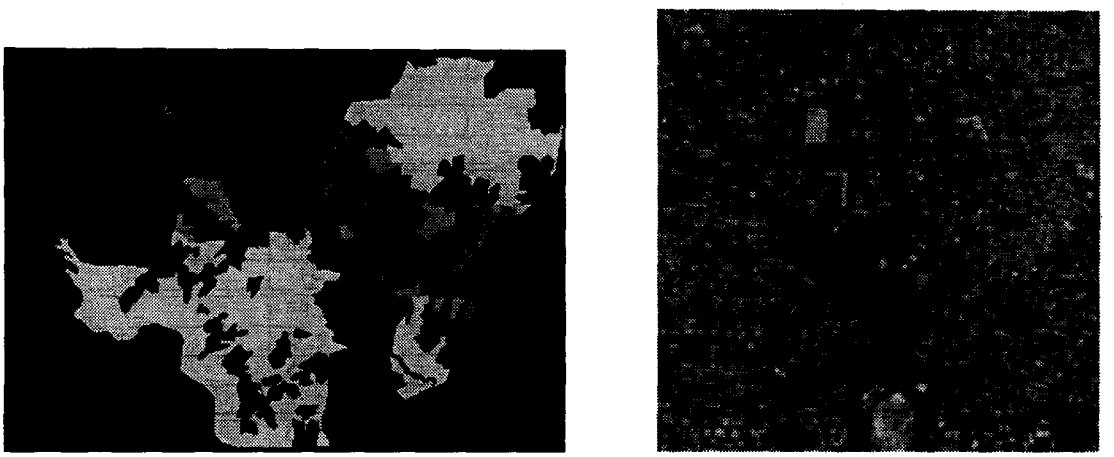

Figure 2: Images: (a) Land use (b) Ban

are optimized for disk access structures. We use a quad-tree merely as a convenient example of a large access structure. Further, as mentioned in Section 3, none of the OODB clustering techniques attempts to optimize access path lengths, concentrating mainly on full traversals or on stochastic traversals. Thus their goals are very different from ours. We compare our techniques with only one of them, namely pre-order clustering, since we are interested in pre-order traversals in addition to access path traversals.

The following numbers were measured: worst case page height, average case page height with each leaf node equally likely to be accessed, and the number of page traversals for performing a pre-order traversal of the entire tree. For the last measure, we assume there are enough buffers to hold a full path in the tree and all parent pages of a page being accessed are pinned in memory. Further, we measured the occupancy of the pages.

We compare mappings using our height minimal clustering (incorporating the pre-order merge heuristic and the merge heuristic) with a pre-order clustering, and with a heuristic we call Smart-BFS. The pre-order clustering basically linearly orders the tree nodes according to their pre-order numbering and then maps as many nodes as it can into a page by considering the nodes in the above order.

Our Smart-BFS clustering heuristic performs BFS locally, starting from the root. Once enough nodes have been visited to fill a node, or there are no more nodes to be searched, the nodes searched so far are assigned to a node. Each of the remaining nodes in the BFS queue then becomes the root of a separate Smart-BFS search. The recursion terminates when all nodes have been searched. Finally, there may be multiple pages with low occupancies, corresponding to Smart-BFS searches started at nodes near the leaf of the tree. These are merged together using a heuristic such as the pre-order merging used for merging pages generated by optimal clustering.

The results of the performance study are summarized in Table 1. The performance numbers show that the optimal height clustering is significantly better than pre-order clustering for the worst case as well as the average case page height. Pre-order clustering performs about 1.6 to 3 times worse on these measures. Smart-BFS performs about the same as optimal height clustering in most cases, with one exception (the landuse data set with page size at 4096) where it results in a greater worst-case height.

Pre-order clustering is clearly the optimal way to cluster data for pre-order traversals. However, it is interesting to note that even on pre-order traversals optimal height clustering with the pre-order merge heuristic is never more than about 30% worse than pre-order clustering. Optimal clustering and Smart-BFS were each better than the other on this metric on different data sets.

The occupancy of pages is about 75% to 80% with optimal clustering along with either of the merging heuristics. This percentage is about the same as in B-trees, and therefore quite acceptable. The page size does not have a significant effect on the occupancy, although it appears that occupancy increases with the complexity of the tree, perhaps because more small clusters are formed. The occupancy of Smart-BFS shows more variation, ranging from 63% to 84%.

The merge heuristics, of course, have no effect on the worst case page height. Although not illustrated by the datasets we used, the merge heuristics can have an effect on average case page heights. The main effect of the merge heuristics is on the cost of pre-order traversal, where pre-order merging is never much worse, and can be much better than previous merging. The number of pages generated with the two merging techniques are always very similar, and it can be

Data Set	Pagesize	Clustering	Merging	Max. Ht.	Avg. Path Len.	Preord. Trav.	Occup.
Ban	1024	Optimal	Preorder	3	3.00	1473	80.7
		Optimal	Previous	3	3.00	1441	82.5
		Preorder	None	8	5.37	1194	99.5
		Smart-BFS	Preorder	3	2.999	1420	84.4
Ban	4096	Optimal	Preorder	3	2.75	348	84.5
		Optimal	Previous	3	2.75	346	85.0
		Preorder	None	7	4.58	295	99.7
		Smart-BFS	Preorder	3	2.347	497	63.5
Landuse	1024	Optimal	Preorder	3	2.97	274	76.3
		Optimal	Previous	3	2.98	329	76.0
		Preorder	None	8	4.78	211	99.1
		Smart-BFS	Preorder	3	2.92	257	84.0
Landuse	4096	Optimal	Preorder	2	2.00	69	75.1
		Optimal	Previous	2	2.00	81	74.0
		Preorder	None	6	3.81	52	99.6
		Smart-BFS	Preorder	3	2.06	62	82.2

Table 1: Costs With Different Clustering Techniques

shown that with pre-order merging, the cost of pre-order traversal is exactly equal to the number of pages.

To summarize, the optimal clustering with either of the merge techniques, and Smart-BFS both perform well across all the metrics we considered. Both can be computed by a simple linear time algorithm. The optimal clustering has the important benefit that it will never generate a clustering with a worst case height greater than Smart-BFS. It is not hard to generate data sets where Smart-BFS generates a bad clustering. One such data set can be constructed to having a collection of N balanced binary trees each of which has as many nodes as will fit in a page, and to link them up with the root of each tree but the first being the rightmost descendant of the previous tree. If the roots of all trees fit in a page, optimal clustering gives a height of 2, while Smart-BFS gives a height of N.

7 Discussion

An alternative to the definition of external path length of a path $\mathcal{P}$ under a mapping M in Section 2 would be to use the number of distinct pages in the mapping of $\mathcal{P}$. This alternative is reasonable if we assume that no page in a path is removed from the buffer pool while traversing the path. Under this definition, the problem of minimizing external path length (with varying node sizes) is NP-complete. A simple reduction from bin-packing establishes this result.

There are several avenues for future work. The first direction would be to expand the performance study. We would like to compare our algorithm with linear quad trees and evaluate our update algorithm for trees. Another direction is to try to derive approximation algorithms for the DAG case and the height-weight minimization problem that can guarantee their results are within a constant factor of the optimal. The performance of heuristics for DAGs also need to be studied empirically. Yet another direction is to create dynamic versions of the DAG mapping heuristics/approximation algorithms. An updater can access an edge through a short path and delete it or insert a new one, causing changes in the mapping on long paths. Updating the mapping along a long path would result in a cost not proportional to the cost of the change in the DAG itself. Dynamic mapping techniques with limited propagation of updates along long paths would be of interest.

Finally, it would be interesting to go beyond DAGS to arbitrary graph structures, but where certain kinds of traversals are defined and there are no cyclic traversals. A binary tree whose leaves are doubly linked would be an example of such a structure. One way to handle such structures is to ignore some links, such as back links in a doubly linked list. Alternative schemes would be of interest. Another direction is to consider more complex traversals, and to introduce limited replication of data in the fashion of [RS94] to improve the cost of traversals.

8 Conclusion

We have presented several algorithms for mapping nodes in access structures to disk pages. We have shown how to generate a page height optimal mapping, and presented heuristics for merging unrelated pages in the mapping to reduce storage costs. We have extended the algorithm to work with average path lengths with given access frequencies for leaf nodes, and to handle the dynamic case. We have shown that the mapping problem is hard if we must get minimum weight mappings among those of minimum page

height, as also if the structure to be mapped is a DAG. We presented heuristics to handle the above cases. Finally we presented a performance study that bears out our analytical results, and shows that the optimal clustering and Smart-BFS techniques perform much better than pre-order clustering on path length measures, and only a little worse on pre-order traversal.

Our techniques are likely to be of importance for creating special purpose access structures on disk, and can reduce effort significantly as compared to designing new disk access structures. Our technique are also likely to be of importance in object-oriented databases, which make creation of linked structures on disk very easy; in particular, our techniques will make it easy to get good performance from the access structures. The programmer will not have to worry either about where to map nodes, or how to convert long skinny structures to short fat structures.

Acknowledgements

We would like to thank P. Venkatachalam of CSRE, IIT Bombay, for providing the data sets on which we ran our tests, and P.P.S. Narayan for implementing the in-memory quad-tree code.

References

[BKKG88] J. Banerjee, Won Kim, S-J. Kim, and J. F. Garza. Clustering a DAG for CAD databases. *IEEE Transactions on Software Engg.*, SE-14(11):1684–1699, November 1988.

[BKSS90] N. Beckmann, H.-P. Kriegel, R. Schneider, and B. Seeger. The R*-tree: An efficient and robust access method for points and rectangles. In *Procs. of the ACM SIGMOD Conf. on Management of Data*, May 1990.

[CH91] J. Cheng and A. Hurson. Effective clustering of complex objects in object-oriented database s. In *Procs. of the ACM SIGMOD Conf. on Management of Data*, May 1991.

[CK89] Ellis E. Chang and Randy H. Katz. Exploiting inheritance and structure semantics for effective clustering and buffering in an object oriented dbms. In *Procs. of the ACM SIGMOD Conf. on Management of Data*, pages 348–357, May 1989.

[DSST86] James R. Driscoll, Neil Sarnak, Daniel Sleator, and Robert E. Tarjan. Making data structures persistent. In *Eighteenth Annual ACM Symp. on Theory of Computing*, 1986.

[Gut84] Antonin Guttman. R-trees: A dynamic index structure for spatial searching. In *Procs. of the ACM SIGMOD Conf. on Management of Data*, pages 47–57, 1984.

[Jag90] H.V. Jagadish. Linear clustering of objects with multiple attributes. In *Procs. of the ACM SIGMOD Conf. on Management of Data*, May 1990.

[KTF95] Anil Kumar, Vassilis J. Tsotras, and Christos Faloutsos. Designing access methods for bitemporal databases. In *Pre-VLDB'95 Workshop on Temporal Databases*, 1995.

[LS92] Q. Li and J.L. Smith. A conceptual model for dynamic clustering in object databases. In *Procs. of the International Conf. on Very Large Databases*, 1992.

[NGV93] Mark Nodine, Michael Goodrich, and Jeffrey S. Vitter. Blocking for external graph searching. In *Procs. of the ACM Symp. on Principles of Database Systems*, 1993.

[Ore89] J. Orenstein. Redundancy in spatial databases. In *Procs. of the ACM SIGMOD Conf. on Management of Data*, 1989.

[Ore90] J. Orenstein. A comparison of spatial query processing techniques for native and parameter spaces. In *Procs. of the ACM SIGMOD Conf. on Management of Data*, May 1990.

[RS94] Sridhar Ramaswamy and Sairam Subramanian. Path caching: A technique for optimal external searching. In *Procs. of the ACM Symp. on Principles of Database Systems*, 1994.

[Sam95] Hanan Samet. Spatial data structures. In Won Kim, editor, *Modern Database Systems*, pages 361–385. ACM Press/Addison Wesley, 1995.

[SB93] Clifford A. Shaffer and Patrick R. Brown. A paging scheme for pointer-based Quadtrees. In *The 3rd International Symposium on Large Spatial Databases*, pages 89–104, 1993.

[Sch77] Mario Schkolnick. A clustering algorithm for hierarchical structures. *ACM Transactions on Data Base Systems*, 2(1):27–44, March 1977.

[SRF87] Timos Sellis, Nick Roussopoulos, and Christos Faloutsos. The R+-tree: A dynamic index for multi-dimensional objects. In *Procs. of the International Conf. on Very Large Databases*, pages 507–518, 1987.

[SSN90] C. A. Shaffer, H. Samet, and R. C. Nelson. QUILT: A geographic information system based on quadtrees. *International Journal of Geographical Information Systems*, 4:103–131, 1990.

[TN91] Manolis M. Tsangaris and Jeffrey F. Naughton. A stochastic approach for clustering in object stores. In *Procs. of the ACM SIGMOD Conf. on Management of Data*, pages 12–21, Denver, Colorado, May 1991.

[TN92] Manolis Tsangaris and Jeffrey Naughton. On the performance of object clustering techniques. In *Procs. of the ACM SIGMOD Conf. on Management of Data*, 1992.