6.854 Advanced Algorithms

Lecture 31: 11/29/2006 Lecturer: David Karger

Scribes: Dan Gulotta

External Memory Algorithms

31.1 Introduction

When working with large data sets, we may not be able to store all of the data are in memory. Since reading from and writing to the hard disk is much slower than reading from and writing to memory, we want to use algorithms that minimize the number of hard disk operations. Let B be the size of a hard disk block, and let M be the size of memory.

31.2 Linked Lists

If we keep $\Theta(B)$ items in a node of a linked list, then we can traverse k items in the list in $O(k/B)$ time. However, we need to make sure that we do this while keeping $O(1)$ time for insertions and deletions. Suppose that we keep at most B items in each node, and we keep at least $B/2$ items in each node except the last. If an insertion makes a node too large, we can split it in two. If a deletion makes a node too small, we can either take an element from the next node or merge with it. Then we have $O(1)$ insertions and deletions and $O(k/B)$ traversal time.

31.3 Trees

Since we get B items whenever we perform a read operation, it makes sense for trees to have a branching factor of $\Theta(B)$. Assuming the tree is balanced, searching will take $O(\log_B N)$ time.

The B-tree data structure is one implementation of a tree with a $\Theta(B)$ branching factor. It is a generalization of 2-3-4 trees. We require that each node in the tree have at most B keys (and therefore $B+1$ children if it is not a leaf) and that each node except the root have at least $B/2$ keys ($B/2+1$ children for non-leaves). Hence the depth of the tree is $O(\log_B N)$.

We use a strategy similar to that for linked lists to handle insert and delete. When an insert would result in a node having $B+1$ keys, we split it into two nodes of b keys and pass the middle key up to the parent. The parent may now have $B+1$ nodes, so we may need to recurse up the tree. In any case, the total number of disk operations required for insertion is $O(\log_B N)$.

Deletions are handled similarly. Suppose that a deletion would result in a node having $B/2-1$ keys. If an adjacent sibling has more than B keys, we can incorporate the key in the parent that separates

the two nodes into our node and use one of the sibling's keys as the new separator. If the sibling has exactly $B/2$ keys, then we can merge the two nodes. This may result in the parent having $B/2 - 1$ keys, in which case we do the same thing with it. Again, $O(\log_B N)$ disk operations are needed.

B-trees give optimal performance for searches in the comparison model. We need $\log N$ bits to locate an element, and we get $\log B$ information by comparing it to every element in a block. Therefore $O(\log_B N)$ reads are required for searching.

31.4 Sorting

31.4.1 Sorting with B-trees

It takes $O(N \log_B N)$ disk operations to insert all element into the B-tree. B-tree. Once the tree is constructed, it takes $O(N/B)$ operations to traverse it and list its elements. So the total time needed to sort using a B-tree is $O(N \log_B N)$.

31.4.2 Merge Sort

Consider merge sort. When we merge two arrays, we begin by reading the first block of each into memory. We begin computing the merged array. When we have a block of the merged larray, we can write it to disk, and when we have used a block of elements from one of the original arrays, we can read another block from the disk. Merge sort has a recursion depth of $O(\log N)$, and $O(N/B)$ blocks need to be read or written at each depth, so the running time is $O(N/B \log N)$.

A slight improvement can be made by sorting lists of size M or smaller entirely in memory. In this case, we only need to do disk reads down to depth $O(\log N/M)$, so the number of disk reads is $O(N/B) \log(N/M)$.

We would like to reduce the number of recursion levels even further. We can accomplish this by merging M/B lists at a time instead of two. The recursion depth is then $\log_{M/B} N/M$, and we still only need to perform $O(N/B)$ disk operations per recursion level. The total time is $O(N/B \log_{M/B} N/M)$.

31.4.3 Optimality

It turns out that M/B-way merge sort performs optimally in the comparison model. When we load a block into memory, we can compare its elements to all of the other elements in memory. The amount of information contained in these comparisons is $B \log M$ to leading order. However, if the contents of the block had previously been loaded into memory then we could already have determined the ordering of the elements in each block. Therefore to leading order $B \log B$ of this information is redundant. The amount of information from the initial reads can produce $N \log M$ information. The number of additinal reads needed is $O(N \log(N/M)/B \log(M/B)) = O(N \log_{M/B} N/M)$.

31.5 Buffer Trees

As we have seen, B-trees do not perform optimally for sorting. One might suspect that inserting elements into the tree one at a time is not a good way to make use of the block structure. We might try to remedy this by keeping a buffer for each node that lists items that we plan to insert into the subtree. When the buffer is full, we will determine to which child nodes we should send each item. The ideal size of these buffers is M. When we empty the buffer, we will need to perform at least M/B writes. In order to avoid performing asymptotically more, we use an M/B branching factor. We want to keep the buffer of the root node in memory in between inserts so that most inserts will require no disk operations.

Because there are only $O(M/B)$ disk operations when we empty the buffer, moving items down the tree costs just $O(1/B)$ per item. So we get a cost of $O(N/B \log_{M/B} N/B)$ for sorting. The B instead of the M in the denominator is asymptotically insignificant since $\log_{M/B} M/B = 1$.