6.854 Advanced Algorithms	
Lecture 21: 11/15/2004	Lecturer: Erik Demaine
Scribes: Dan Ports	

External-Memory Algorithms

21.1 The External-Memory Model

Previously we have considered the costs of algorithms in the context of the *random-access machine* (RAM) model, where each elementary operation and each memory access has a constant cost. While sufficient for many purposes, this model does not always predict how algorithms will behave on real memory. For example, if the entire data set does not fit into main memory, segments will periodically have to be paged in from disk; disk access times are so much larger that they dominate the runtime. CPUs typically have one or more layers of cache that are even faster than main memory. In general, modern systems have a multi-layered memory structure, where the memory closest to the CPU is fastest but smallest, and the memory farthest from the CPU is largest but slowest.

The *external-memory model*, introduced by Aggarwal and Vitter in 1988, is an alternative to the RAM model that takes into account disk access time. It uses a two-layered memory structure, as shown in Figure 21.1. The CPU has a fast connection to the *cache*, which has size M. The *disk* is much larger and much slower to access[1]. Both the cache and disk are divided into *blocks* of size B; the cache thus holds $\frac{M}{B}$ blocks, while the disk can hold many more. A *memory transfer* operation reads a block from disk to cache, or writes a block from cache to disk. These are necessary since the CPU can only operate directly on the contents of the cache; blocks on disk must first be transferred to the cache before they can be processed.

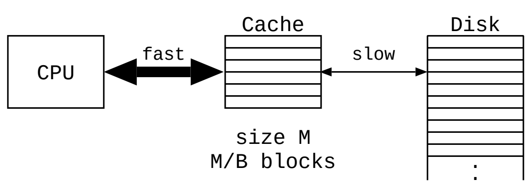

Figure 21.1: The External-Memory Structure

We assume there is no cost to access the cache and perform operations on it. Instead, the cost of an algorithm in the external-memory model is the number of memory transfers required. Because disk seek times are so much higher than memory access times, this cost captures the runtime more realistically than the runtime given by the RAM model.

[1]For clarity, we exclusively use "cache" and "disk" to refer to the two types of memory. Some papers instead use "cache" and "memory" and others use "memory" and "disk"; we use the term "memory" only to refer to memory in general, without regard for its location in the hierarchy.

[2]Images taken from 2003 scribe notes, courtesy of David R. Cheng

Note that all transfers take place in blocks of size B. Thus, accessing one element in memory requires fetching the entire block that contains it. The goal is to take advantage of locality: if nearby memory locations are accessed together, then the block containing them will only need to be transferred once. Any algorithm that has runtime $T(N)$ in the RAM model clearly makes no more than $T(N)$ memory transfers in the external-memory model. Our goal is to reduce this amount. Ideally, we will be able to reduce it to $\frac{T(N)}{B}$.

21.2 Scanning Algorithms

Scanning is a simple standard technique that takes advantage of locality. A scanning algorithm processes the input by visiting each element in order. For example, to find the smallest element in an array of integers, we can scan over the array, keeping track of the smallest element seen so far. Since memory locations are accessed sequentially, adjacent memory locations are accessed together, and each block needs to be transferred from disk to cache only once. Only $\lceil \frac{N}{B} \rceil + 1$ memory transfers are required to visit N elements. This is optimal.

A slightly more complicated example is the problem of reversing the order of an array. Observe that this can be done by swapping pairs of elements: the ith element and the $N - i$th element are interchanged. We can implement this using blocks: we first read the first and last blocks, reverse their elements and swap their positions, then proceed to the second and penultimate blocks, etc. This can be viewed as performing two scanning sequences in parallel, one from the beginning of the array and one from the end. The number of required memory transfers is $O\left(\frac{N}{B}\right)$.

Note that the array reversal algorithm requires the cache to be large enough to hold two blocks, i.e. $M \geq 2B$. This is not a problem. In general, we can assume that the cache size is $\Omega\left(N^{1+\epsilon}\right)$. This is known as the *tall-cache assumption* and is standard.

21.3 B-Trees

We now turn to the problem of building a comparison-based search tree. B-trees are a data structure well-suited for use in the external-memory model. Recall that a B-tree is a balanced search tree where each node contains b values and has a branching factor of $b + 1$. Thus, the tree has height $\Theta\left(\log_{b+1} N\right)$, and performs operations in $\Theta\left(\log_{b+1} N\right)$ time. Why is this well-suited for the external-memory model? We can choose b to be a constant factor less than B such that the representation of each node fits in a block. Then any operation requires $\Theta\left(\log_{B+1} N\right)$ memory transfers.

Is this a desirable runtime? The following theorem from information theory shows that in fact it is optimal:

Theorem 1 *Searching for an element x in a sorted list of N elements requires $\Omega\left(\log_{B+1} N\right)$ memory transfers.*

Proof: Suppose that we have an sorted array of numbers $1 \ldots N$ and we wish to discover where x is located in this array. The most succinct encoding of the answer an index into the array, which has $\lg N + 1$ bits. Initially the cache is empty and we have no information. Each time a transfer from disk to cache is performed, we read at most B elements. We therefore determine where x fits into these x elements; since they are sorted, the smallest encoding of the information determined is the $\lg B + 1$ bit index into the array. So we learn no more than $\lg B + 1$ new bits of information per memory transfer. Since we need to determine $\lg N + 1$ bits of information, at least $\frac{\lg N+1}{\lg B+1} = \lg_{B+1} N$ memory transfers are required. ■

Corollary 1 *Finding an element in a search tree requires* $\Omega\left(\log_{B+1} N\right)$ *memory transfers.*

Proof: The problem of searching for an element in a sorted list is reducible to the problem of finding an element in a search tree, so the same lower bound applies. ■

21.4 Sorting

Consider now the problem of comparison-based sorting. In the RAM model, we can sort n elements by inserting each of them into a B-tree, then repeatedly extracting the minimum element until the tree is empty. This requires $\Theta(n \log n)$ time, which we know is optimal.

We can use the same approach in the external-memory model. Again there are n insert and extract-min operations, but now they each require $\Theta\left(\log_{B+1} N\right)$ memory transfers, so the total cost of the algorithm is $\Theta\left(N \log_{B+1} N\right)$. This is *not* optimal in the external-memory model.

21.4.1 Mergesort

The standard mergesort algorithm has desirable runtime characteristics in the external-memory model because merging can be performed via a scanning algorithm. We simply scan through each of the two input arrays in parallel, adding the smallest element from either of the two arrays to the output array and fetching new blocks as needed. This requires $O\left(\frac{N}{B}\right)$ time. Recall that mergesort divides a problem of size N into two sorting subproblems of size $\frac{N}{2}$ and a merge of size N. This gives the following recurrence:

$$T(N) = 2T\left(\frac{N}{2}\right) + O\left(\frac{N}{B}\right) \tag{21.1}$$

The recursion tree for this recurrence is shown in Figure 21.2. Note also that once the problem has been reduced to sorting an array of size M, we can solve it in constant time, because all the work is done in cache. Thus, there are $\frac{N}{M}$ elements on the last level of the recursion tree, and so the tree has height $\lg \frac{N}{M}$. At each level, the total work done in the merging steps sums to $\frac{N}{B}$ memory transfers. So the solution to the recurrence and the total cost of the algorithm is

$$T(N) = O\left(\frac{N}{B} \log \frac{N}{M}\right) \tag{21.2}$$

This is an improvement over B-tree sorting, but it is still not optimal. We can make a small modification to the mergesort procedure to improve it:

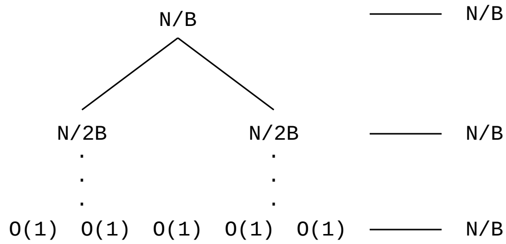

Figure 21.2: Recursion tree for mergesort

21.4.2 $\frac{M}{B}$-way Mergesort

Previously we considered a *binary* mergesort algorithm: it divides each problem of size N into two subproblems of size $\frac{N}{2}$. But we are not limited to a two-way division. We can divide the problem into any number of subproblems. A cache of size M holds $\frac{M}{B}$ blocks, so $\frac{M}{B}$-way division is ideal. This is the largest choice that makes the scanning merge algorithm possible, since one block from each sorted subproblem array can fit into cache. (To be precise, we actually need a cache of size $M + O(1)$ since we must store the output block in cache too, but this detail can be safely ignored.) The recursion is now:

$$T(N) = \frac{M}{B} T\left(\frac{N}{\frac{M}{B}}\right) + O\left(\frac{N}{B}\right) \tag{21.3}$$

which leads to the recursion tree in Figure 21.3. As before, once the problem has been reduced to

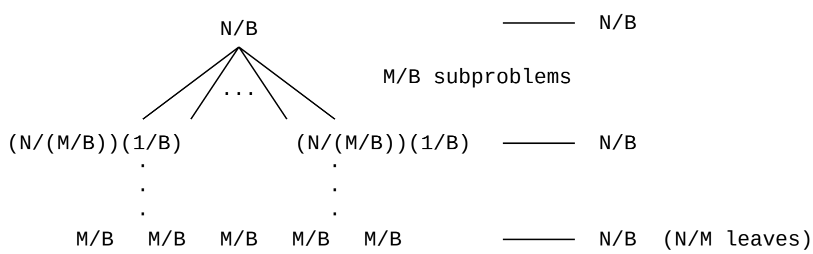

Figure 21.3: $\frac{M}{B}$-way mergesort recursion tree

sorting an array of size M, we can solve it in constant time. Each level of the recursion tree requires $O\left(\frac{N}{B}\right)$ work, and the height of the tree is $\log_{\frac{M}{B}} \frac{N}{M}$. So the solution to the recurrence is

$$T(N) = \frac{N}{B}\left(1 + \log_{\frac{M}{B}} \frac{N}{M}\right) = \frac{N}{B}\left(2 + \log_{\frac{M}{B}} \frac{N}{B}\right) \tag{21.4}$$

$$= O\left(\frac{N}{B} \log_{\frac{M}{B}} \frac{N}{B}\right) \tag{21.5}$$

This is the number of memory transfers required by the algorithm.

21.4.3 Optimality

We have seen that $\frac{M}{B}$-way mergesort sorts N elements using $\frac{N}{B}\log_{\frac{M}{B}}\frac{N}{B}$ memory transfers. Is this optimal? As with B-trees, information theory shows that it is.

Theorem 2 *To sort N elements, $\Omega\left(\frac{N}{B}\log_{\frac{M}{B}}\frac{N}{B}\right)$ memory transfers are required.*

Proof: We need to identify which of the permutations of the N input elements is the sorted order. This is $\lg N!$ bits of information. But we actually need only learn $\lg\frac{N!}{B!}$ bits of information, since once we read a block into cache, we can sort it instantly and its sorted ordering therefore comes for free.

We can always keep the cache sorted, since operations on the cache incur no cost. Every time we read a block of B elements, we determine where it fits in among the elements currently in cache. That is, we learn which of the interleavings of B elements into M elements ($M+B$ total) is the sorted order. There are $\binom{M+B}{B}$ such interleavings, so at most $\lg\binom{M+B}{B}$ bits of information are obtained from each memory transfer. Therefore, the number of memory transfers we need to determine our requisite $\lg\frac{N!}{B!}$ bits of information is

$$\frac{\lg\frac{N!}{B!}}{\lg\binom{M+B}{B}} = \Omega\left(\frac{\lg\frac{N!}{B!}}{\lg\frac{(M+B)!}{M!B!}}\right) \tag{21.6}$$

$$= \Omega\left(\frac{N\lg\frac{N}{B}}{(M+B)\lg(M+B)-M\lg M-B\lg B}\right) \tag{21.7}$$

$$= \Omega\left(\frac{N\lg\frac{N}{B}}{M\lg M+B\lg M-M\lg M-B\lg B}\right) = \Omega\left(\frac{N\lg\frac{N}{B}}{B\lg\frac{M}{B}}\right) \tag{21.8}$$

$$= \Omega\left(\frac{N}{B}\log_{\frac{M}{B}}\frac{N}{B}\right) \tag{21.9}$$

So $\Theta\left(\frac{N}{B}\log_{\frac{M}{B}}\frac{N}{B}\right)$ is optimal. ■

21.5 Buffer Trees

The buffer tree is a data structure similar to a search tree that provides better amortized memory transfer bounds. It can perform the Insert, Delete, Delete-Min, Batched-Search, and Batched-Range-Search operations using $O\left(\frac{1}{B}\log_{\frac{M}{B}}\frac{N}{B}\right)$ amortized memory transfers per operation. This is no ordinary bound — it is $o(1)$! (Fortunately, this is an amortized bound, so the universe will not implode.) The Insert, Delete, and Delete-Min operations are standard; Batched-Search, and Batched-Range-Search require a bit more explanation. Batched-Search(x) finds the element with value closest to x, and Batched-Range-Search(x, y) finds all elements with values between x and y. However, they do not return their answer immediately. Instead, they return their answer — which is still based on the state of the tree at the time the query was performed — at some later time. There is also a FLUSH operation which causes the results for all pending batched operations to be returned immediately.

Note that this data structure provides a superset of the priority queue operations. Hence, we can use it for sorting. Inserting n items and performing DELETE-MIN operations repeatedly gives us the sorted output with $O\left(\frac{1}{B}\log_{\frac{M}{B}}\frac{N}{B}\right)$ memory transfers, which we have seen is optimal.

The design of this data structure is similar to that of a B-tree. Now, however, each internal node has size $\Theta\left(\frac{M}{B}\right)$ (hence a branching factor of $\Theta\left(\frac{M}{B}\right)$ and each leaf has size $\Theta(B)$. Each internal node also has a buffer of size $\Theta(M)$. The root node has a buffer of size $\Theta(M)$ which is always kept in cache. This is shown in Figure 21.4. We use the standard data structure of laziness: requests are stored in the buffers at each level before being lazily processed and pushed down into the children when the buffers become full.

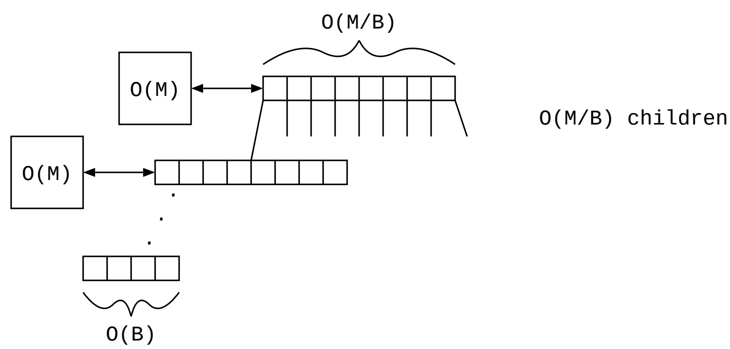

Figure 21.4: Buffer Tree Structure

We now describe the procedures for performing each of the buffer tree's operations:

21.5.1 INSERT

To insert an element, we simply add it to the root buffer. We tag it with a "insert" request classification, and a timestamp. The timestamp is necessary because, for example, we need to distinguish between elements that were inserted before a batched search was performed and those that were inserted after the search was initiated but before it was completed (and hence should not be output). The root buffer is always in cache, so this requires no memory transfers, provided that the root buffer does not overflow.

If the root buffer overflows, we sort the buffer. It is in cache, so sorting incurs no cost. We then distribute the elements in the buffer to the appropriate children's buffers. Each of the children's buffers must only be touched once, since the root buffer's elements are in sorted order, and there are $\Theta\left(\frac{M}{B}\right)$ children, so $O\left(\frac{M}{B}\right)$ memory transfers are required. But the root buffer only overflows when it contains $\Theta(M)$ elements, so we can charge the cost of the overflow to each of them, and each one incurs an amortized cost of $O\left(\frac{1}{B}\right)$ due to this step. We then check each of the children's buffers for overflows.

21.5.2 Overflows

Interior nodes If an interior node's buffer overflows, note that it contains $\Theta(M)$ old elements. We can sort these using $O\left(\frac{M}{B}\log_{\frac{M}{B}}\frac{M}{B}\right) = O\left(\frac{M}{B}\right)$ memory transfers. The newly added elements just inserted by the parent are in sorted order, so we can merge them in $O\left(\frac{M}{B}\right)$ memory transfers. We then distribute the elements to the children (in $O\left(\frac{M}{B}\right)$ memory transfers) and check the children for overflows.

Leaf nodes If a leaf node overflows, we split the leaf just as we would a B-tree, leaving all associated buffers empty. This happens rarely.

Analysis Observe that an overflow involving $\Theta(M)$ elements requires $O\left(\frac{M}{B}\right)$ memory transfers. So we can charge $O\left(\frac{1}{B}\right)$ to each of the elements involved. Note that elements only move down in the tree. The tree is balanced with branching factor $\frac{M}{B}$, so the height is $O\left(\log_{\frac{M}{B}}\frac{N}{B}\right)$. Thus, each element is only involved in this many overflows. So the amortized cost per element is $O\left(\frac{1}{B}\log_{\frac{M}{B}}\frac{N}{B}\right)$.

21.5.3 Flush

The Flush operation empties all buffers in the tree. Our amortized analysis does not directly apply in this case, because we will be performing the overflow element-distribution procedure even on buffers that are not full. However, the total cost is still less than it would be in the case in which all buffers are full. From above, this is $O\left(\frac{M}{B}\log_{\frac{M}{B}}\frac{N}{B}\right)$.

21.5.4 Delete

To perform a Delete operation, we insert it into the root buffer as in an Insert operation, except tagged as a "delete" element. We propagate it downward through the tree in the same way. When a "delete" element is inserted into a buffer that contains an "insert" element for the same value (with an earlier timestamp), the two cancel each other out and can be removed. If they meet in a leaf instead of an interior node, we perform a B-tree delete instead of a B-tree insert.

21.5.5 Batched-Search

A Batched-Search finds the element in the tree with the value closest to the query value. This is accomplished by inserting the value into the root buffer, tagged with the request type and timestamp, and propagating it downward as before. If the "batched-search" element is ever in the same buffer as an "insert" element for the same value with an appropriate timestamp, that "insert" element is an exact match and we can return it. If this never occurs by the time the "batched-search" element reaches a leaf, we perform a B-tree search for its value and return the closest match.

21.5.6 Batched-Range-Search

A Batched-Range-Search on x and y finds all elements in the tree with values between x and y. We insert the search into the buffer as with a Batched-Search. As the search proceeds, we accumulate the answers "off to the side" in a separate memory location so we can return them all together. Each time the "batched-range-search" element is inserted into a buffer, we add all matching elements in that buffer to the output buffer. When the buffer overflows, we distribute the "batched-range-search" element to *all* children that contain some part of the range between x and y. When the query element reaches a leaf, we perform a B-tree range search. When all query elements have reached the leaves, we output the contents of the output buffer.

This operation requires amortized $O\left(\frac{1}{B}\log_{\frac{M}{B}}\frac{N}{B}+\frac{K}{B}\right)$ memory transfers, where K is the number of matching elements.

21.5.7 Delete-Min

To support Delete-Min queries, we maintain an extra buffer of size $\Theta(M)$ that contains the minimal elements, which we keep in cache at all times. As each new element is inserted, we add it into the buffer if there are empty slots in the buffer or if it is smaller than one of the elements in the buffer (this incurs no cost since the buffer is kept in cache). As elements are deleted, we remove them from the buffer.

Then, to perform a Delete-Min procedure, we simply take the minimal element from the buffer and perform a Delete operation on it. The only complication is that the buffer may be empty. In this case, we need to refill it by acquiring the next $\Theta(B)$ minimal elements. Note that in a B-tree these are in the leftmost leaf, but in a buffer tree they may be further up the tree because they have not yet propagated down to the leaf. However, because of the search tree ordering, they will always lie along the left spine of the buffer tree. Thus, we simply need to traverse the left spine of the tree, flushing all the buffers. When this is completed, we refill the minimal-element buffer from the leftmost leaf.

This requires flushing $O\left(\log_{\frac{M}{B}}\frac{N}{B}\right)$ buffers, since this is the height of the tree. Each contains $O\left(\frac{M}{B}\right)$ elements, so $O\left(\frac{M}{B}\log_{\frac{M}{B}}\frac{N}{B}\right)$ memory transfers are required. But we only need to perform this refill step after $\Theta(M)$ operations, so the amortized cost is $O\left(\frac{1}{B}\log_{\frac{M}{B}}\frac{N}{B}\right)$.