Algorithmica (2003) 37: 1–24
DOI: 10.1007/s00453-003-1021-x

Algorithmica
© 2003 Springer-Verlag New York Inc.

The Buffer Tree: A Technique for Designing Batched External Data Structures[1]

Lars Arge[2]

Abstract. We present a technique for designing external memory data structures that support batched operations I/O efficiently. We show how the technique can be used to develop external versions of a search tree, a priority queue, and a segment tree, and give examples of how these structures can be used to develop I/O-efficient algorithms. The developed algorithms are either extremely simple or straightforward generalizations of known internal memory algorithms—given the developed external data structures.

Key Words. I/O efficiency, Internal memory algorithms, Batched external data structures, Buffer tree.

1. Introduction. In recent years, increasing attention has been given to Input/Output-efficient (or I/O-efficient) algorithms. This is due to the fact that communication between fast internal memory and slower external memory such as disks is the bottleneck in many computations involving massive datasets. The significance of this bottleneck is increasing as internal computation gets faster and parallel computing gains popularity.

Much work has been done on designing external versions of data structures designed for internal memory. Most of these structures are designed to be used in on-line settings, where queries should be answered immediately and within a good worst case number of I/Os. As a consequence they often do not take advantage of the available main memory, leading to suboptimal performance when they are used in solutions for batched (or off-line) problems. Therefore several techniques have been developed for solving massive batched problems without using external data structures.

In this paper we present a technique for designing external data structures that take advantage of the large main memory. We do so by only requiring good amortized performance and by allowing query operations to be batched. We also show how the developed data structures can be used in simple and I/O-efficient algorithms for a number of fundamental computational geometry and graph problems. Our technique has subsequently been used in the development of a large number of I/O-efficient algorithms in these and several other problem areas.

[1] An extended abstract version of this paper was presented at the Fourth Workshop on Algorithms and Data Structures (WADS '95) [3]. The author was supported in part by the National Science Foundation through ESS Grant EIA-9870734 and CAREER Grant CCR-9984099. Part of this work was performed while he was at BRICS, University of Aarhus, Denmark, and supported by the ESPRIT II Basic Research Actions Program of the EC under Contract No. 7141 (project ALCOM II) and by Aarhus University Research Foundation.

[2] Department of Computer Science, Duke University, Durham, NC 27708, USA. large@cs.duke.edu.

Received April 8, 2002; revised October 1, 2002, November 24, 2002, and December 8, 2002. Communicated by S. Albers. Online publication April 25, 2003.

1.1. *I/O Model and Previous Results.* We will be working in the standard I/O model introduced by Aggarwal and Vitter [2]. The model has the following parameters:

$$\begin{aligned} N &= \text{number of elements in the problem instance,} \\ M &= \text{number of elements that can fit into main memory,} \\ B &= \text{number of elements per block,} \end{aligned}$$

where $M < N$ and $1 \leq B \leq M/2$. An I/O operation (or I/O) is a swap of B elements from internal memory with B consecutive elements from external memory (disk). The measure of performance we consider is the number of such I/Os needed to solve a problem. As we shall see shortly, N/B (the number of blocks in the problem) and M/B (the number of blocks that fit into internal memory) play an important role in the study of I/O complexity. Therefore, we use n as shorthand for N/B and m for M/B. We say that an algorithm uses a linear number of I/Os if it uses $O(n)$ I/Os. In [45] the I/O model is extended with a parameter D. Here the external memory is partitioned into D distinct disk drives, and if no two blocks come from the same disk, D blocks can be transferred per I/O.

Early work on I/O algorithms concentrated on algorithms for sorting and permutation-related problems in the single disk model [2], as well as in the extended version of the I/O model [37], [36], [45]. In the single disk model, external sorting requires $\Theta(n \log_m n)$ I/Os,[3] which is the external equivalent of the well-known $\Theta(N \log N)$ internal sorting bound. Searching for an element among N elements requires $\Theta(\log_B N)$ I/Os. This bound is obtained by the ubiquitous B-tree [20], [26], [33] (or more generally an (a, b)-tree [31]), which also supports updates in $O(\log_B N)$ I/Os. More recently external algorithms and data structures have been developed for a number of problems in different areas. See recent surveys for further references [6], [5], [43], [42].

An important consequence of the fundamental external memory bounds for sorting and searching is that, unlike in internal memory, use of on-line efficient data structures in algorithms for batched problems often leads to inefficient algorithms. Consider for example sorting N elements by performing N insert followed by N delete operations on a B-tree. This algorithm performs $O(N \log_B N)$ I/Os, which is a factor $B \log_B m$ from optimal. Thus while for on-line problems $O(\log_B N)$ is the I/O bound corresponding to the $O(\log_2 N)$ bound on many internal memory data structure operations, for batched problems the corresponding bound is $O((\log_m n)/B)$. The problem with the B-tree in a batched context is that it does not take advantage of the large main memory—effectively it works in a model where the size of the main memory is equal to the block size.

1.2. *Our Results.* In this paper we present a technique for designing efficient batched external data structures. We use our technique to develop a number of external data structures, which in turn can be used to develop I/O-optimal algorithms for several computational geometry and graph problems. All these algorithms are either extremely simple or straightforward generalizations of known internal algorithms—given the developed external data structures. This is in contrast to most other known I/O algorithms,

[3] For convenience we define $\log_m n = \max\{1, (\log n)/(\log m)\}$.

which are often very I/O specific. Using our technique we manage to isolate all the I/O-specific parts of the algorithms in the data structures, something that is nice from a software engineering point of view. More specifically, the results in this paper are the following:

Sorting. We develop a simple linear space dynamic tree structure (the *buffer tree*) with operations *insert*, *delete*, and *write*. We prove amortized I/O bounds of $O((\log_m n)/B)$ on the first two operations and $O(n)$ on the last. Note that $(\log_m n)/B < 1$ for all practical values of M, B, and N, and insertions and deletions take less than one I/O amortized. Using the structure we can sort N elements with the standard tree-sort algorithm in the optimal number of I/Os. One novel feature of the algorithm is that it does not require all N elements to be given by the start of the sorting algorithm.

Graph problems. We extend the buffer tree with a *deletemin* operation in order to obtain an external *priority queue*. We prove an $O((\log_m n)/B)$ amortized bound on the number of I/Os used by this operation. Using the structure we provide an alternative implementation of the so-called time-forward processing technique [25]. Previous implementations only work for large values of m, while our algorithm works for all m. In [25] the time-forward processing technique is used to develop an I/O-efficient list-ranking algorithm, which in turn is used to develop efficient algorithms for a large number of graph problems. All these algorithms thus inherit the constraint on m and our new algorithm removes it from all of them.

Computational geometry problems. We extend the buffer tree with a *batched rangesearch* operation (obtaining a one-dimensional *buffered range tree*). We prove an $O((\log_m n)/B + r)$ amortized bound on the number of I/Os used by the operation, where r is the number of *blocks* reported. Furthermore, we use our technique to develop an external version of the *segment tree* supporting *insert*, *delete*, and *batched search* operations with the same I/O bounds as the corresponding operations on the buffered range tree structure. The two structures enable us to solve the orthogonal line segment intersection, the batched range searching, and the pairwise rectangle intersection problems in the optimal number of I/Os. We solve these problems with exactly the same plane-sweep algorithms as are used in internal memory [38]. Optimal algorithms for the three problems are also developed in [30] and [13], but as discussed earlier these algorithms are very I/O specific. It should be noted that the query operations on the buffered range tree and buffered segment tree are *batched* in the sense that the result of a query is not reported immediately. Instead parts of the result are reported at different times when other operations are performed. This suffices in the plane-sweep algorithms we are considering, as the sequence of operations performed on the data structure in these algorithms does not depend on the results of the queries in the sequence. In general, problems where the sequence of operations on a data structure is known in advance are known as *batched dynamic problems* [28].

The main organization of the rest of this paper is the following. In the next section we describe the main idea in the buffer technique, and in Section 3 we then develop the basic buffer tree structure, which can be used to sort optimally. In Sections 4 and 5 we extend this structure with deletemin and batched rangesearch operations, respectively. The batched segment tree is described in Section 6. Using techniques from [36], all the developed structures can be modified to work in the D-disk model—that is, the I/O bounds can be divided by D. We discuss the necessary modifications in Section 7. Finally conclusions are given in Section 8.

2. The Buffer Technique. In this section we present our technique for designing external data structures that support batched operations I/O efficiently. The main idea in the technique is to perform operations on an external (high fan-out) tree data structure in a "lazy" manner using main-memory-sized "buffers" associated with internal nodes of the tree.

As an example, imagine we are working on a height $O(\log_m n)$ search tree structure with elements stored in the leaves, that is, a structure with fan-out $\Theta(m)$ internal nodes and N elements stored in sorted order in n leaves with $\Theta(B)$ elements each. (Intuitively, one can think of obtaining such a high-degree structure from a binary internal structure by grouping the binary internal nodes in the structure into super-nodes of size $\Theta(m)$, and grouping the leaves together into blocks). We assign buffers of size $\Theta(m)$ blocks to each of the $O(n/m)$ internal nodes of the structure. When we then want to insert a new element, we do not search down the tree for the relevant leaf right away. Instead, we wait until we have collected a block of insertions (or other operations), and then we insert this block into the buffer of the root. When a buffer "runs full" the elements in the buffer are "pushed" one level down to buffers on the next level. We call this a *buffer-emptying process*. Deletions or other and perhaps more complicated updates, as well as queries, are basically performed in the same way. Note that as a result of the laziness, we can have several insertions and deletions of the same element in the tree at the same time, and we therefore "time stamp" elements when they are inserted in the root buffer. As discussed below, the laziness also means that queries are batched since a query result may be generated (and reported) lazily by several buffer-emptying processes.

To show the desired I/O bounds we need to be able to empty a buffer in $O(m + r')$ I/Os, where r' is the number of blocks reported by queries while emptying the buffer. In this case we can assign each element $O((\log_m n)/B)$ credits when it is inserted in the root buffer, and maintain the invariant that each block in the buffer of a node v that is the root of a subtree of height h has $\Theta(h)$ credits. As we only perform a buffer-emptying process when a buffer runs full (contains $\Theta(m)$ blocks), and as we can charge the r'-term to the queries that cause the reports, the blocks in the buffer can pay for the emptying process as they all get pushed one level down the tree. Of course, we also need to consider, e.g., rebalancing of the transformed structure. We return to this, as well as the details in other operations, in later sections. Another way of looking at the above amortization argument is that we touch each *block* of elements a constant number of times on each level of the structure. Note that the argument works as long as we can empty a buffer in a linear number of I/Os in the number of elements in the buffer. In later sections we use this property several times when we show how to empty a buffer containing $x > m$ blocks in $O(m + x) = O(x)$ I/Os. The argument also works if the fan-out of the tree is $\Theta(m^c)$ for $0 < c \leq 1$, as the tree height remains $O(\log_m n)$ even with this smaller fan-out.

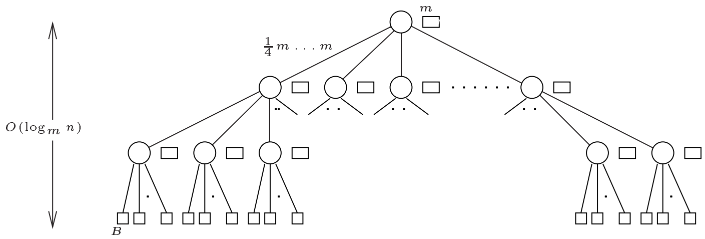

Fig. 1. A buffer tree.

3. The Buffer Tree. In this section we describe the details of the basic buffer tree algorithms, considering the operations needed in order to use the structure in a simple sorting algorithm. In Section 4 we extend the structure in order to obtain an external priority queue, and in Section 5 to obtain a (one-dimensional) buffered range tree.

The buffer tree is an (a, b)-tree [31] with $a = m/4$ and $b = m$ over $O(n)$ leaves containing $\Theta(B)$ elements each, extended with a size m buffer attached to each node. In such a tree, all leaves are on the same level and all nodes (except the root) have fan-out between $m/4$ and m. Thus the height of the tree is $O(\log_m n)$ and the structure uses $O(n)$ space. Refer to Figure 1. As discussed in Section 2, we perform an insertion or deletion as follows: We construct an element consisting of the element in question, a time stamp, and an indication of whether the element corresponds to an insertion or a deletion. When we have collected B such elements in internal memory we insert them in the buffer of the root. If this buffer now contains more than m blocks we perform a buffer-emptying process. We define *internal nodes* to be nodes that do not have leaves as children. Nodes that are not internal are called *leaf nodes*. The buffer-emptying process used on internal nodes (corresponding to the discussion in the last section) is given in Figure 2; we maintain the invariant that at most one block in a buffer is non-full and distribute elements in sorted order. The buffer-emptying process is only performed recursively on internal nodes—only after finishing *all* buffer-emptying processes on internal nodes do we empty the buffers of the relevant leaf nodes (Figure 3). Note also that during recursive buffer-emptying processes, the buffer of a node can grow larger than M

- Load the first M elements in the buffer into internal memory, sort them, and merge them with the rest of the (already sorted) elements in the buffer. If corresponding insert and delete elements with time stamps in the correct order are encountered the two elements are deleted.
- Scan through the sorted list of elements from the buffer while distributing elements to the appropriate buffers one level down (maintaining the invariant that at most one block in a buffer is non-full).
- If the buffer of any of the children now contains more than m blocks, and if the children are internal nodes, then recursively apply the emptying-process to these nodes.

Fig. 2. The buffer-emptying process on an internal node v.

- As long as there is a leaf node v with a full buffer (size greater than m blocks) do the following (k is the number of leaves of v):
  1. Sort the elements in the buffer as for internal nodes and remove "matching" insert/delete elements.
  2. Merge the sorted list with the sorted list of elements in the leaves of v while removing "matching" insert/delete elements.
  3. If the number of blocks of elements in the resulting list is **smaller** than k do the following:
     (a) Place the elements in sorted order in the leaves of v.
     (b) Add "dummy-blocks" until v has k leaves and update the partition elements of v.
  4. If the number of blocks of elements in the resulting list is **bigger** than k do the following:
     (a) Place the k smallest blocks in the leaves of v and update the partition elements of v.
     (b) Repeatedly insert one block of the rest of the elements and rebalance.
- Repeatedly delete one dummy block and rebalance; before a rebalance operation (fuse/share) on a node v is performed a buffer-emptying process is performed on the two sibling nodes (possibly) involved in a rebalance operation (v' and v'' in Figure 5). If the delete (the buffer-emptying processes) results in any leaf buffer becoming full, these buffers are emptied (and insert rebalancing performed) before the next dummy block is deleted.

Fig. 3. Emptying the buffers of leaf nodes.

elements (if most of the elements from a buffer are distributed to one child). To handle this we utilize that elements are distributed in sorted order, which means that when a buffer grows larger than M elements it consists of at most M unsorted elements followed by a list of sorted elements—it can thus be sorted in a single scan. The buffer-emptying process on an internal node buffer containing $x > m$ blocks can be performed in $O(x)$ I/Os as required; the elements are first sorted in $O(x)$ I/Os using a single scan. Then they are scanned again using $O(x)$ I/Os and distributed to the appropriate buffers one level down. The distribution uses at least one I/O per child and thus it is performed in $O(x + m) = O(x)$ I/Os. (Note that the worst case cost of emptying a buffer containing $o(m)$ blocks is $\Theta(m)$.)

A buffer-emptying process on a leaf node (Figure 3) may result in the need for rebalancing the structure. Leaf node buffers are only emptied after all internal node buffer-emptying processes have been performed in order to prevent rebalancing and buffer-emptying processes from interfering with each other. Rebalancing is basically performed as on normal (a, b)-trees [31]. After inserting a new leaf into an (a, b)-tree, rebalancing is performed using a series of node *splits* (Figure 4). Similarly, after deleting a leaf, rebalancing is performed using a series of node *fusions* possibly ending with a node *sharing* (Figure 5). In the buffer tree case we modify the delete rebalancing algorithm slightly. The modification consists of performing one or two buffer-emptying processes before every fuse or share operation. Specifically, we perform a buffer-emptying process on nodes v' and v'' in Figure 5 (possibly) involved in a fuse or share rebalancing operation. This way we can perform the actual rebalancing operation as normal, without having to worry about elements in the buffers: The buffer-emptying process maintains the invariant that buffers of nodes on the path from the root to a leaf node with full buffer are all empty. This means that when removing "dummy-nodes" after emptying the buffers of all leaf

Rebalancing after inserting an element below v:

```
DO $v$ has $b+1$ children ->
  IF $v$ is the root ->
    let $x$ be a new node and make $v$ its
    only child
  ELSE
    let $x$ be the parent of $v$
  FI
  Let $v'$ be a new node
  Let $v'$ be a new child of $x$ immediately
  after $v$
  Split $v$:
    Take the rightmost $\lceil(b+1)/2\rceil$ children
     away from $v$ and make them children
     of $v'$.
  Let v=x
OD
```

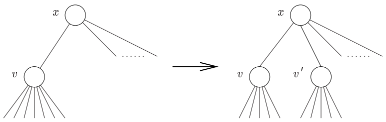

Fig. 4. Insert in an (a, b)-tree.

Rebalancing after deleting an element below v:

```
DO $v$ has $a-1$ children AND
    $v'$ has less than $a+t+1$ children ->
  Fuse $v$ and $v'$:
     Make all children of $v'$ children of
     $v$
  Let $v$=$x$
  Let $v'$ be a sibling of $x$
  IF $x$ does not have a sibling ($x$ is
  the root)
      AND $x$ only has one child ->
     Delete $x$
     STOP
  FI
  Let $x$ be the parent of $v$.
OD
(* either $v$ has more than $a$ children
  and we are finished, or we can
  finish by sharing *)
IF $v$ has $a-1$ children ->
  Share:
     Take $s$ children away from $v'$ and
     make them children of $v$.
FI
```

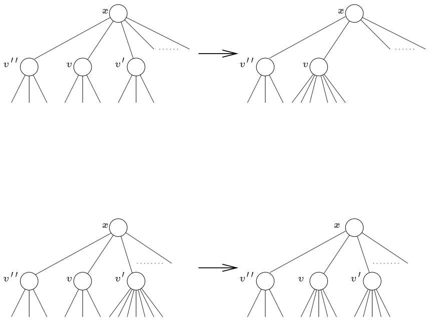

Fig. 5. Delete from an (a, b)-tree ($s = \lceil((b/2 - a) + 1)/2\rceil$ and $t = (b/2 - a) + s - 1$).

nodes (Figure 3), nodes v in fuse or share rebalance operations performed as a result of the deletions will already have empty buffers. If the buffer-emptying process on v' and v'' results in a leaf buffer running full, the invariant will be fulfilled since all nodes on the path from x to the root have empty buffers. No buffer-emptying is needed when splitting nodes during insertions since splits are only performed on nodes with empty buffers. This explains the special way of handling deletes with dummy blocks—insertion of blocks cannot result in buffer-emptying processes, while a deletion of a block may result in many such processes. The use of dummy blocks also ensures that no buffers are full during delete rebalancing. Note that the reason we empty the buffers of both v' and v'' in a delete rebalancing operation is that after the possibly resulting (insert) rebalancing, one (but not both) of these nodes may not have the same parent as v (if v splits), preventing it from fusing with v.

THEOREM 1. *The total cost of an arbitrary sequence of N intermixed insert and delete operation on an initially empty buffer tree is $O(n \log_m n)$ I/Os, that is, the amortized cost of an operation is $O((\log_m n)/B)$ I/Os. The tree uses $O(n)$ space.*

PROOF. As a buffer-emptying process on a node containing $x > m$ blocks uses $O(x)$ I/O, not counting I/Os used on rebalancing, the total cost of all buffer-emptying processes on nodes with *full* buffers is bounded by $O(n \log_m n)$ I/Os. In [31] it is shown that if $b > 2a$, the number of rebalancing operations in a sequence of K updates on an initially empty (a, b)-tree is bounded by $O(K/(b/2 - a))$. As we are inserting blocks in the $(m/4, m)$-tree underlying the buffer tree, the total number of rebalance operations in a sequence of N updates on the buffer tree is bounded by $O(n/m)$. Each rebalance operation takes $O(m)$ I/Os ($O(m)$ I/Os may be used by each delete rebalance operation to empty a *non-full* buffer), and thus the total cost of the rebalancing is $O(n)$. This proves the first part of the theorem. The structure uses linear space since the tree itself uses $O(n)$ blocks, the $O(n/m)$ non-full buffers use $O(m)$ blocks each, and since each element is stored in $O(1)$ places in the structure. The theorem follows. □

In order to use the buffer tree in a simple sorting algorithm, we need an empty/write operation that empties all buffers and then reports the elements in the leaves in sorted order. All buffers can be emptied simply by performing a buffer-emptying process on all nodes in the tree in BFS order. As emptying one buffer costs $O(m)$ I/Os amortized (not counting recursive processes), and as the total number of buffers in the tree is $O(n/m)$, we obtain the following:

THEOREM 2. *The amortized cost of emptying all buffers of a buffer tree after performing N updates on it is $O(n)$ I/Os.*

COROLLARY 1. *A set of N elements can be sorted in $O(n \log_m n)$ I/Os using the buffer tree.*

As mentioned, the buffer tree sorting algorithm is I/O optimal and the first algorithm that does not require all elements to be given by the start of the algorithm. The ideas in the basic buffer tree have been used in several other algorithms, e.g., in algorithms for performing a bulk of operations on an R-tree [41], [8] and in I/O-efficient string sorting algorithms [7].

REMARKS. (i) By rebalancing the buffer tree top-down instead of bottom-up (as, e.g., discussed in [35]), we can obtain a simpler rebalancing algorithm than the one described above. Such a strategy requires "tuning" the buffer tree constants (fan-out and buffer size) such that the maximal number of elements in the buffers of a subtree is less than half the number of elements in the leaves of the subtree. In this case we can perform a split, a fuse, or a share as part of the buffer-emptying process on a node v if needed in order to guarantee that there is room in v to allow all its children to fuse or split. In this way we can make sure that rebalancing will never propagate up the tree. Unfortunately, we have not been able to make this simpler strategy work when rangesearch operations (as discussed in Section 5) are allowed.

(ii) In practice, good space utilization is often one of the most important considerations when working with massive datasets. Thus it is important to consider the constant hidden in the $O(n)$ space bound, or (almost) equivalently, the average fill ratio of the disk blocks occupied by a buffer tree. It is well known that in practice the average block fill ratio for a B-tree (that is, an (a, b)-tree with $a = B/2$ and $b = B$) is around 70%, and that the fill ratio is dominated by the ratio for the leaves. In the buffer tree the underlying structure is also an (a, b)-tree (but with $a = m/4$ and $b = m$), but unlike the B-tree a very underutilized block may be required in each internal node v to store the elements in the buffer of v. However, since the number of internal nodes is only a fraction of the number of leaves, we expect that as in the B-tree case the leaf fill ratio will determine the overall fill ration. Since updates are performed on the leaves in a batched way (when the buffer of a leaf node is emptied) at most every $m/4$ leaf (one for each leaf node) is non-full. Thus we expect the space utilization of a buffer tree to be at least as good as that of a B-tree. This has been empirically validated, e.g., when using the buffer technique to bulk load (construct) the R-tree [8].

(iii) Experimental results on buffer-tree-based sorting have been reported in [32], and other experimental results on the buffer tree technique has been reported in, e.g., [22], [8], [27], [41], and [40]. These experiments show that if the (a, b)-tree fan-out and buffer size is chosen carefully, the buffer tree technique is relatively efficient in practice.

4. External Priority Queue. A search tree structure can normally be used to implement a priority queue because the smallest element in a search tree is in the leftmost leaf. The same strategy cannot immediately be used on the buffer tree since the smallest element is not necessarily stored in the leftmost leaf—smaller elements could reside in the buffers of the nodes on the leftmost root-leaf path. However, there is a simple strategy for performing a deletemin operation in the desired amortized I/O bound. To perform a deletemin operation we simply perform a buffer-emptying process on all nodes on the path from the root to the leftmost leaf using $O(m \cdot \log_m n)$ I/Os amortized. Then we delete the $\frac{1}{4}m \cdot B$ smallest elements in the tree, which are now stored in the children (leaves) of the leftmost leaf node, and keep them in internal memory. This way we can answer the next $\frac{1}{4}m \cdot B$ deletemin operations without performing any I/Os. Of course we then also have to update the minimal elements as insertions and deletions are performed, but we can do so in a straightforward way in internal memory without performing extra I/Os. A simple amortization argument then shows the following:

THEOREM 3. *Using $m/4$ blocks of internal memory, an arbitrary sequence of N insert, delete, and deletemin operations on an initially empty buffer tree can be performed in $O(n \log_m n)$ I/Os, that is, in $O((\log_m n)/B)$ I/Os amortized per operation. The tree uses $O(n)$ space.*

Recently a number of other I/O-efficient priority queue structures have been developed using the buffer technique on a heap [29], [34] or tournament tree structure [34]. The latter structure allows updates to be performed in $O((\log_2 N)/B)$ I/Os amortized. Note that if the key of an element to be updated is known (and assuming without loss of generality that all keys are distinct), the update can be performed in $O((\log_m n)/B)$ I/Os using a deletion and an insertion. Other external priority queues are designed in [22] and [23].

REMARK. In some applications (see, e.g., [16]) it is preferable to use less than $m/4$ blocks of internal memory for the external priority queue structure. We can make our priority queue work with $\frac{1}{4}m^{1/c}$ $(0 < c \leq 1)$ blocks of internal memory simply by decreasing the fan-out and the size of the buffers to $\Theta(m^{1/c})$ as discussed in Section 2.

Application: *time-forward processing*. As mentioned in the Introduction, an approach called *time-forward processing* for evaluating circuits (or "circuit-like" computations) in external memory is described in [25]; given a connected directed acyclic graph G with N edges, where each vertex v has indegree $v_d \leq M/2$ and an associated function f_v: $\mathbb{R}^{v_d} \rightarrow \mathbb{R}$, the problem is to compute the value of f_v for each sink v (out-degree 0 node). It is assumed that G is topologically sorted, that is, that the labels of the vertices come from a total order $<$ and that $v < w$ for every edge (v, w). The main issue with evaluating the function computed by G is to ensure that when evaluating the function f_v at a particular vertex v the values of v's inputs are in the internal memory. Thinking of vertex v as being evaluated at "time" v then motivates the term *time-forward processing* for the approach of processing the vertices in topological order and sending results along edges "forward in time."

In [25] an $O(n \log_m n)$ algorithm for time-forward processing is presented. The algorithm only works for large values of m ($\sqrt{m/2}\log(M/2) \geq 2\log(2N/M)$) and is not particularly simple. Using our external priority queue we can easily develop a simple alternative algorithm without the constraint on m; when evaluating a vertex v we simply send the result forward in time by inserting elements in the priority queue. More specifically, assume that the list E of the edges (v, u) in G is given sorted by source vertex v (topologically sorted). The vertices are processed in topological order maintaining the following invariant: For each processed vertex v and each edge of the form (v, u) where u has not yet been processed, the priority queue contains a priority u element augmented with the result of f_v. Initially the priority queue is empty. When a node u is processed the priority queue contains an element for each edge of the form (v, u) and thus the results of all of u's u_d immediate predecessors can be obtained by performing

u_d deletemin operations. Then f_u can be evaluated in internal memory without further I/Os. Finally, the list E can be used to re-establish the invariant by inserting an element with priority w for each edge of the form (u, w) into the priority queue. In total the algorithm performs one scan of E, as well as $O(N)$ operations on the priority queue, for a total of $O(n \log_m n)$ I/Os (Theorem 3).

In [25] two deterministic algorithms for external list ranking are developed. One of these algorithms uses time-forward processing and therefore inherits the constraint on m not being too small. The other has a constraint on B not being too large (m too small). The list ranking algorithm is in turn used to develop efficient external algorithms for a large number of graph problems, including expression tree evaluation, centroid decomposition, least common ancestor computation, minimum spanning tree verification, connected component computation, minimum spanning forest computation, biconnected components computation, ear decomposition, and a number of problems on planar *st-graphs*. By developing an alternative time-forward processing algorithm without constraints on m, we have thus also removed the constraint from these algorithms. After the extended abstract version of this paper was published, the buffer technique and the external priority queue were used to obtain further improved algorithms for a large number of external graph algorithms (see, e.g., [34], [24], [14], and [4]).

5. Buffered Range Tree. In this section we extend the basic buffer tree with a rangesearch operation for reporting all elements in a query interval $[x_1, x_2]$. Normally, a rangesearch operation is performed on a (leaf-based) search tree by searching down the tree for x_1 and x_2, and then reporting all elements in leaves between x_1 and x_2. The query can also be answered while searching down the tree, by reporting all elements in the leaves of the relevant subtrees on the way down. This is the strategy we use on the buffer tree. The general idea in our rangesearch operation is the following. As when performing an update, we start by inserting a rangesearch element containing the interval $[x_1, x_2]$ and a time stamp in the root buffer. We then modify our buffer-emptying process in order to deal with rangesearch elements. To process a rangesearch element in a buffer-empty process on v, we first determine if a search for x_1 and x_2 would proceed to the same subtree among the subtrees rooted at the children of v. If they are we simply insert the element in the corresponding buffer. If not we "split" the element in two—one for x_1 and one for x_2—and report the elements in subtrees where *all* elements are contained in the interval $[x_1, x_2]$. The splitting only occurs once and after that the rangesearch elements are pushed downwards in the buffer-emptying processes like insert and delete elements, while elements in subtrees containing elements in $[x_1, x_2]$ are reported. As discussed previously, this means that the result of a rangesearch operation is reported in a batched way.

Several complications need to be overcome to make the above strategy work efficiently. For example, to report all elements in a subtree we cannot (as in Section 3) simply empty the buffers of the subtree by performing a buffer-emptying process on all nodes and then report the elements in the leaves—the buffers of the subtree may contain other rangesearch elements and elements contained in the intervals of these searches should also be reported. Another complication is that we need to report the elements in a subtree in $O(N_\text{a}/B)$ I/Os, where N_a is the actual number of elements in the subtree, that is, the number of elements not deleted by delete elements in the buffers of the subtree. N_a could be significantly smaller than the total number $N_\mathrm{a} + N_\mathrm{d}$ of elements in the subtree, where N_d is the number of delete elements in the subtree. However, if we can empty all buffers in the subtree in $O((N_\mathrm{a} + N_\mathrm{d})/B)$ I/Os—*and* remove all delete elements—we can charge the N_d part to the delete elements (adding $O(1/B)$ I/Os to the amortized number of I/Os used by a delete operation).

In the following we describe how to overcome the above complications and make the buffer-emptying strategy work efficiently. In Section 5.1 we describe an algorithm for emptying all buffers of a (sub-) tree in a linear number of I/Os while answering all rangesearch queries in buffers of the tree. In Section 5.2 we then describe precisely how to modify the buffer-emptying process in order to obtain an I/O-efficient rangesearch operation.

5.1. *Emptying All Buffers of a Buffered Range Tree.* In the following we assume that only previously inserted elements are deleted, that is, that only elements actually present in the buffered range tree are deleted. This assumption is natural (at least) in a batched dynamic environment and it implies the following useful property: Let i, d, and s be "matching" delete, insert, and rangesearch elements (that is, "$i = d$" and "i is contained in the interval of s"). When we say that the elements are in *time order* i, s, d we mean that i was inserted before s, and s before d. Depending on the time order of i, s, and d, we can use the following *order-changing* rules:

- Time order is i, s, d and no other relevant elements (especially not rangesearch elements) are between d and i in the time order: we can report that i is in s and remove i and d.
- Time order is i, s and no other relevant elements (especially not d) are between s and i in the time order: we can report that i is in s and interchange their time order.
- Time order is s, d and no other relevant elements (especially not i) are between d and s in the time order: we can report that d (i) is in s and interchange their time order.

We define a list of insert, delete, and rangesearch elements to be in *time order representation* if the time order is such that all delete elements are "older" than (were inserted before) all rangesearch elements, which in turn are older than all insert elements, *and* the three groups of elements are internally sorted by their key—by x_1 as far as the rangesearch elements are concerned. Using the order-changing rules we can now prove two important lemmas.

LEMMA 1. *Less than M elements in a buffer can be brought into time order representation in $O(m + r)$ I/Os, where $r \cdot B$ is the number elements reported in the process.*

PROOF. We simply load the elements into internal memory and use the order-changing rules to interchange the time order of the relevant elements. Then we sort the three groups of elements individually and write them back to the buffer in time order representation. □

LEMMA 2. *Let S_1 and S_2 be subsets of a set S such that all elements in S_2 are older than all elements in S_1, and such that all other elements in S are either younger or older*

When we in the following write that we report elements, we actually accumulate elements to be reported in internal memory and report (write) them when a block has been accumulated.

1. Interchange the time order of set d_1 and set i_2 by "merging" them while removing matching insert and delete elements.
2. Interchange the time order of set d_1 and set s_2 by "merging" them while reporting the relevant elements (rangesearch-element "hits") in the following way:
   During the merge a third list of "active" rangesearch elements from s_2 is kept—except for the B most recently added elements—on disk.
   - When a *rangesearch* element from s_2 has the smallest x (that is, x_1) value, it is inserted in the active list.
   - When a *delete* element from d_1 has the smallest x-value, the active list is scanned and it is reported that the delete element is in the interval of all rangesearch elements that have not yet "expired"—that is, elements whose x_2 value is less than the value of the delete element. At the same time all expired rangesearch elements are removed from the list.
3. Interchange the time order of set s_1 and set i_2 by "merging" them and reporting "hits" as in step 2.
4. Merge i_1 with i_2, s_1 with s_2, and d_1 with d_2.

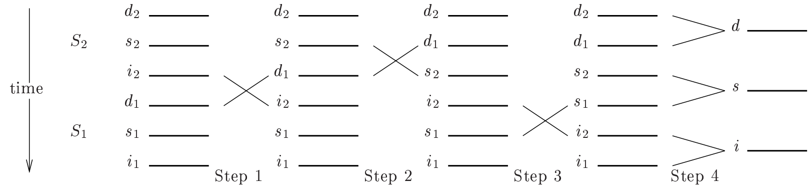

Fig. 6. Merging sets in time order representation (algorithm and illustration of algorithm).

than all elements in S_1 and S_2. Given S_1 and S_2 in time order representation, the time order representation of $S_1 \cup S_2$ can be constructed in $O(s_1 + s_2 + r)$ I/Os, where s_1 and s_2 are the sizes of S_1 and S_2, respectively, and $r \cdot B$ is the number of elements reported in the process.

PROOF. The algorithm for "merging" S_1 and S_2 is given in Figure 6 (we abuse notation slightly and use S_1 and S_2 to denote the two sets in time order representation). In step 1 we interchange the order of the delete elements d_1 in S_1 with the insert elements i_2 in S_2, and in step 2 we interchange them with the rangesearch elements s_2 in S_2 while reporting the relevant elements. That we correctly report all relevant elements (rangesearch-element "hits") follows from the order changing rules. Then in step 3 the time order of s_1 and i_2 is interchanged, such that the relevant lists can be merged in step 4. Steps 1 and 4 obviously use a linear number of I/Os and a simple amortization argument shows that steps 2 and 3 also use a linear number of I/Os in the number of elements reported. □

In our algorithm for emptying all buffers of a subtree, we utilize that all elements in buffers of nodes on a given level of a buffered range tree are in "correct" time order compared with all *relevant* elements on higher levels: an element in the buffer of node v was inserted before all elements in buffers of the nodes on the path from v to the root of

1. Make three lists for each level of the tree, consisting of the elements in time order representation: For a given level, the time order representation of elements in each buffer is constructed using Lemma 1. The resulting lists are appended after each other to obtain the total time order representation.
2. Repeatedly, starting at the root, merge the time order representation of level j with the time order representation of level $j + 1$ using Lemma 2.

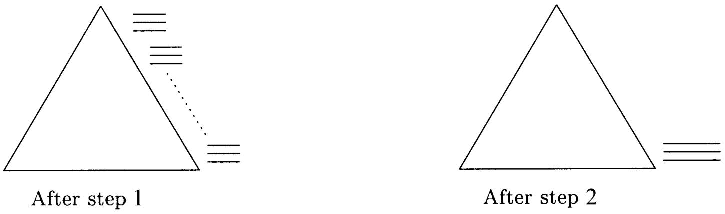

Fig. 7. Emptying all buffers and collecting the elements in time order representation (algorithm and illustration of algorithm).

the tree. Thus we can safely assume that all elements on one level were inserted before all elements on higher levels.

LEMMA 3. *All buffers of a buffered range tree with* n *leaves can be emptied and all the elements collected into time order representation in* $O(n + r)$ *I/Os, where* $r \cdot B$ *is the number of elements reported in the process, provided that all buffers contain less than* M *elements.*

PROOF. The correctness of the algorithm given in Figure 7 follows from Lemmas 1 and 2 and the above discussion. Step 1 creates the time order representation of the elements on each level in a number of I/Os equal to $O(m)$ times the number of nodes in the tree, plus the number of I/Os used to report elements (Lemma 1). This is $O(n + r)$ since the number of nodes in the tree is $O(n/m)$. One merge in step 2 uses a linear number of I/Os in the number of elements in the merged lists, plus the I/Os used to report elements (Lemma 2). Since each level of the tree contains more nodes than all levels above it, the number of I/Os used to merge the time order representation of level j with the time order representation of all the elements above level j is bounded by $O(m)$ times the number of nodes on level j. The bound then again follows from the total number of nodes in the tree being $O(n/m)$. □

5.2. *Buffer-Emptying Process on Buffered Range Tree.* We are now ready to describe the precise details of the buffered range tree buffer-emptying process. As in the basic buffer tree, the emptying algorithm is different for internal and leaf nodes. The internal node algorithm is given in Figure 8 and the leaf node algorithm in Figure 9.

To empty the buffer of an *internal node* v, we first compute the time order representation of the elements in the buffer (we compute the time order representation of the first M elements using Lemma 1 and merge it with the time order representation of the remaining elements using Lemma 2). After distributing the delete elements (step 2) and

1. Compute the time order representation of the elements in the buffer of v using Lemmas 1 and 2.
2. Scan the delete elements and distribute them to the buffers of the relevant children of v.
3. Scan the rangesearch elements and determine which subtrees below v should have their elements reported.
4. For each of the relevant subtrees empty all but leaf buffers and report relevant elements as follows:
   (a) Remove the delete elements distributed to the subtree in step 1 and store them in temporary space.
   (b) Collect the elements in the buffers of the subtree in time order representation (empty the buffers) using Lemma 3.
   (c) Merge the time order representation with the time order representation of the delete elements removed in step (a) using Lemma 2.
   (d) Scan the insert and delete elements of the resulting time order representation and distribute a copy of the elements to the relevant leaf buffers.
   (e) Merge a copy of the elements in the leaves (in time order representation) with the time order representation using Lemma 2.
   (f) Remove the rangesearch elements.
   (g) Report the resulting elements for each of the relevant rangesearch elements in the buffer of v.
5. Scan the rangesearch elements again and distribute them to the buffers of the relevant children of v.
6. Scan the insert elements and distribute them to the buffers of the relevant children of v. Elements for subtrees that were emptied are distributed to the leaf buffers of these trees.
7. If the buffer of any of the children of v now contains more than M elements, and the children are internal nodes, then recursively apply the buffer-emptying process.

Fig. 8. Buffered range tree buffer-emptying process on internal node v.

computing what subtrees need to be emptied (step 3), we perform the following for each such tree: We empty the buffers of the subtree using Lemma 3 (step 4(b))—all buffers of the subtree are non-full since we do not include the delete elements from the buffer of v and since the buffer-emptying process is performed recursively top-down. (At the same time we automatically report the relevant elements for all the rangesearch elements in the buffers of the subtree.) Then we remove the elements deleted by delete elements in the buffer of v (step 4(c)). The resulting set of elements, together with the relevant insert elements from the buffer of v, should be inserted into or deleted from the subtree. Therefore we insert them into the relevant leaf buffers (step 4(d) and step 6), which are emptied at the very end of the algorithm. Finally, we merge the time order representation

- For each leaf node v with full buffer do the following:
  1. If needed, construct the time order representation of the elements in the buffer of v.
  2. Merge (a copy of) the time order representation with the time order representation of the elements in the children of v (leaves) using Lemma 2.
  3. Remove the rangesearch elements from the buffer of v.
- Empty all leaf buffers (and rebalance the tree) using the algorithm in Figure 3 (Section 3).

Fig. 9. Emptying the buffers of leaf nodes of a buffered range tree.

of the elements from the buffers of the subtree with the elements in the leaves of the subtree (step 4(e)). This way we at the same time report the relevant elements in the leaves for the rangesearch elements from the buffers, *and* obtain a list of all (undeleted) elements in the subtree. These elements can then be reported for each of the relevant rangesearch elements from the buffer of v (step 4(g)). Finally, after having processed the relevant subtrees, we distribute the remaining elements in the buffer of v to buffers one level down (steps 5 and 7), and then recursively empty the buffers of the relevant children.

As in the basic buffer tree case, buffers of *leaf nodes* are only emptied after all full internal buffers have been emptied. To empty a buffer of leaf node v, we first construct the time order representation of the elements in the buffer as in the internal node case. Note that a leaf buffer can become full if v is part of a subtree being emptied as a result of a buffer-emptying process on an internal node, or as a result of a buffer-emptying process being performed on $\mathit{parent}(v)$. In the first case the buffer is already in time order representation. Then we report the relevant element simply by merging the time order representation of the elements from the buffer with the time order representation of the elements in the leaves below v. Finally, we remove the rangesearch elements. After having processed all full leaf buffers in this way, all such buffers only contain insert and delete elements and we can empty the buffers (and rebalance) with the algorithm used on the basic buffer tree (Figure 3). Note that the elements in the buffer are already sorted and thus the first step of the rebalancing algorithm can be avoided.

THEOREM 4. *The total cost of an arbitrary sequence of N intermixed insert, delete, and rangesearch operations performed on an initially empty buffered range tree is $O(n \log_m n + r)$ I/Os, where $r \cdot B$ is the number of reported elements. The tree uses $O(n)$ space.*

PROOF. The correctness of the algorithm follows from the above discussion. Using Lemmas 1–3, it is easy to realize that a buffer-emptying process on an internal node uses a linear number of I/Os in the number of elements in the emptied buffer and the number of elements in the leaves of the emptied subtrees, plus the number of I/Os used to report elements. The only I/Os we cannot account for using the standard argument in Section 2 are the ones used on emptying subtrees. However, as discussed, this linear cost can be divided between the reported elements and the deleted elements, so that the deleted elements pay for their own deletion—once the elements in the buffers of internal nodes of a subtree are removed and inserted into the leaf buffers, they will only be touched again when they are inserted into or deleted from the tree by the rebalancing algorithm (leaf buffer-emptying). The cost of emptying a leaf buffer, not counting rebalancing, is linear in the number of elements processed and reported. Thus the standard amortized argument applies. The rebalancing cost can be analyzed as in the buffer tree case (Section 3) and the theorem follows. □

5.3. *Application*: *Orthogonal Line Segment Intersection*. The problem of *orthogonal line segment intersection reporting* is defined as follows: Given N line segments parallel to the axes, report all intersections of orthogonal segments. In internal memory the optimal plane-sweep algorithm for the problem (see, e.g., [38]) makes a vertical sweep with a horizontal line, inserting the x-coordinate of a vertical segment in a search tree T when its top endpoint is reached, and deleting it again when its bottom endpoint is reached. When the sweep-line reaches a horizontal segment, a rangesearch operation with the two endpoints of the segment is performed on T in order to report intersections. Using precisely the same algorithm and our range tree data structure (remembering to empty the tree after performing the sweep), we immediately obtain the following:

COROLLARY 2. *The orthogonal line segment intersection reporting problem can be solved in* $O(n \log_m n + r)$ *I/Os.*

As mentioned, an alternative orthogonal line segment intersection algorithm is developed in [30]. However, unlike our algorithm that "hides" all I/O in the buffered range tree, the algorithm is very I/O specific. That both algorithms are optimal follows from the $\Omega(N \log_2 N + R)$ comparison model lower bound and the results in [9] and [10].

6. Buffered Segment Tree.

In this section we illustrate how the buffer technique can be used to develop an external batched version of the segment tree [21], [38]. We also discuss how this structure can be used to solve the batched range searching and the pairwise rectangle intersection problems in the optimal number of I/Os.

The segment tree is a data structure for maintaining a set of segments (intervals) such that given a query point q all segments containing q can be reported efficiently. Such queries are normally called *stabbing queries*. We assume without loss of generality that the segment endpoints are distinct. Assume also that the endpoints belong to a fixed set E of size N. In internal memory, a segment tree consists of a static binary *base tree* on the sorted set E, and a segment s is stored in up to two nodes on each of the $O(\log_2 N)$ levels of the tree. Specifically, an interval X_v is associated with each node v in a natural way, covering all endpoints stored in the leaves in the subtree rooted at v, and s is stored in v if it contains X_v but not the interval associated with $parent(v)$. All segments associated with v are stored in an unordered list. To answer a stabbing query q we simply have to search down the base tree for the leaf containing q, reporting all segments stored in nodes encountered during the search.

6.1. *Definition of Buffered Segment Tree.*

To use the buffer technique to obtain a segment tree supporting batched operations I/O efficiently, we first have to modify the internal segment tree in order to obtain a tree with higher fan-out. (The intuitive idea of grouping the internal nodes together into super-nodes of size $\Theta(m)$ does not work since a segment can be stored in $O(\log_2 N)$ nodes—thus even after grouping the nodes together we can be forced to use many I/Os to store a segment in $O(\log_2 N)$ lists.) An external buffered segment tree consists of a balanced base tree T with branching factor $\sqrt{m}$ on the sorted set E. Refer to Figure 10. We assume without loss of generality that $n = (\sqrt{M})^c$ for some integer $c > 0$. The root r of T naturally partitions E into $\sqrt{m}$ *slabs* σ_i, separated by dotted lines in Figure 10, and we define a *multislab* to be a contiguous range of slabs, such as, for example, $[\sigma_1, \sigma_4]$. There are $m/2 - \sqrt{m}/2$ multislabs associated with r and a segment list is associated with each of them. Segments are now stored in these lists as follows. A segment s completely spanning one or more slabs is called a *long segment*

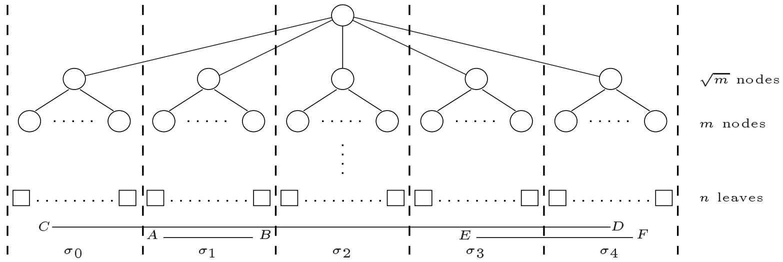

Fig. 10. An external segment tree based on a set of N segments. The long segment $\overline{CD}$ is stored in the root as well as recursively in trees corresponding to slabs σ_0 and σ_4. The short segments $\overline{AB}$ and $\overline{EF}$ are only stored recursively in σ_1 and in σ_3 and σ_4, respectively.

and we store a copy of s in the list associated with the largest multislab it spans. For example, $\overline{CD}$ in Figure 10 is a long segment and it is stored in the list associated with the multislab $[\sigma_1, \sigma_3]$. Segments not spanning slabs are called *short segments* and they are passed down to lower levels of T where they may span recursively defined slabs. $\overline{AB}$ and $\overline{EF}$ in Figure 10 are examples of short segments. Additionally, the portions of long segments that do not completely span slabs are treated as short segments. On each level of the tree there are at most two synthetically generated short segments for each long segment (since a segment can only be short in nodes encountered in searches for its endpoints), and thus a segment is stored in at most two lists on each level of T. Segments passed down to a leaf are stored in just one list. Note that at most one block of segments is stored in each leaf, since we assumed that all segments have distinct endpoints.

As in the internal segment tree, a stabbing query q on a buffered segment tree (with empty buffers) can be answered by searching down T, reporting all segments containing q in nodes encountered. In a node v we report all segments stored in lists associated with each of the multislabs containing q. Answering queries on an individual basis is of course not I/O efficient.

6.2. *Batched Operations on Buffered Segment Trees*. We now show how to perform insertions and batched queries on the buffered segment tree. We do not support deletions, but instead we require that a delete time is given when a segment is inserted in the tree. This requirement is fulfilled in the applications discussed below. In general the requirement, along with the assumption that the set E is fixed, means that our structure can only be used to solve batched dynamic problems as discussed in the Introduction. We do not support general deletions because the lists associated with a node can be much larger than $\Theta(M)$, which means that we cannot afford to load them when performing a buffer-emptying process.

Given the sorted set E, the base tree T can be built in $O(n)$ I/Os by first constructing the leaves in a scan through the sorted list, and then repeatedly constructing the next level of the tree by scanning through the previous level. An insert or a query is then performed basically as discussed in Section 2. We construct a new element with the segment or query point in question, a time stamp, and—if the element corresponds to an insertion—a delete time. When we have collected a block of elements we insert them in the buffer of the root. If the buffer of the root now contains more than M elements we perform

On internal node v:

- Repeatedly load $m/2$ blocks of elements into internal memory and do the following:
    1. Store segments:
       Collect all segments to be stored in v (long segments). For every multislab list in turn, insert the relevant long segment in the list (maintaining the invariant that at most one block of a list is non-full). Finally, replace every long segment with the short (synthetic) segments to be stored recursively.
    2. Answer queries:
       For every multislab list in turn, decide if the segments in the list need to be reported (are stabbed by any query point). Scan through the relevant lists, reporting the relevant segments and removing segments that have expired (segments for which all the relevant queries are inserted after their delete time).
    3. Distribute segments and queries to the buffers of the nodes on the next level.
- Apply the buffer-emptying process recursively to children with buffers now containing more than m blocks.

On leaf node v:

- Do exactly the same as on internal nodes, except that in step 3 *segments* distributed to a leaf l are just inserted in the segment block associated with l and *queries* are removed after reporting the relevant segments in leaves below v.

Fig. 11. Buffer-emptying process on buffered segment tree.

a buffer-emptying process. The buffer-emptying process is presented in Figure 11. We have the following:

THEOREM 5. *Given the endpoint set E of size N and a sequence of insertions* (*with delete time*) *and stabbing queries*, *we can build a buffered segment tree T on E and perform all the operation on T in $O(n \log_m n + r)$ I/Os using $O(n \log_m n)$ space.*

PROOF. To prove that the $O(N)$ operations can be performed in $O(n \log_m n + r)$ I/Os, we need to argue that a buffer-emptying process on a buffer containing $x > m$ blocks is performed in $O(x + r')$ I/Os. First consider the buffer-emptying process on an *internal node*. Loading and distributing (step 3) $M/2$ segments to buffers one level down is obviously performed in $O(m)$ I/Os as previously. *Step* 1 is also performed in $O(m)$ I/Os since the $O(m)$ segments are distributed to $O(m)$ multislab lists. If we charge $O(m)$ I/Os to the buffer-emptying process in *step* 2 (the number of I/Os used to load non-full multislab list blocks), the rest of the I/Os used to scan a multislab list can either be charged to query output or to the deletion of a segment, assuming that each segment holds $O(1/B)$ credits to pay for its deletion. This credit can be accounted for (deposited) when inserting a segment in a multislab list. Overall only $O(m)$ I/Os are charged to the buffer-emptying process every time $M/2$ segments are processed, for a total of $O(x)$ I/Os as required. Similarly, we can easily argue that a *leaf node* buffer-emptying process uses $O(x + r')$ I/Os. Thus by the discussion in Section 2 all operations are performed in $O(n \log_m n + r)$ I/Os.

As previously, in order to answer all queries we need to empty all buffers after performing all operations. As in Section 3, we simply perform a buffer-emptying process on all nodes in BFS order. As there are $O(n/\sqrt{m})$ nodes in the tree, the argument used previously would lead to an $O(n\sqrt{m})$ I/O bound for emptying all buffers (plus the number of I/Os used to report stabbings). The problem seems to be that T has $n/\sqrt{m}$ leaf nodes containing $O(m)$ multislab lists each, and that we can be forced to use an I/O for each of these lists when emptying the buffers of a leaf node. However, since only $O(n \log_m n)$ I/Os have been performed before the final buffer-emptyings, at most $O(n \log_m n)$ multislab lists can actually contain *any* segments. Therefore only $O(n \log_m n + r)$ I/Os are used to empty buffers of all the leaf nodes. Furthermore, as the number of internal nodes is $O(n/m)$, the buffers of these nodes can all be emptied in $O(n)$ I/Os. □

6.3. *Applications of the Buffered Segment Tree.* The *batched range searching problem*—given N points and N axis parallel rectangles in the plane, report all points inside each rectangle—can be solved with a plane-sweep algorithm in almost the same way as the orthogonal line segment intersection problem. The algorithm makes a vertical sweep with a horizontal line, inserting a segment (rectangle) in a segment tree when the top of a rectangle is reached, and deleting it when the bottom is reached. A stabbing query is performed when a point is encountered. Using the buffered segment tree in this algorithm immediately yields the following:

COROLLARY 3. *The batched range searching problem can be solved in $O(n \log_m n + r)$ I/Os.*

The problem of pairwise orthogonal rectangle intersection is defined similar to the orthogonal line segment intersection problem. Given N rectangles in the plane, the problem is to report all intersecting pairs. In [21] it is shown that the rectangle intersection problem is equivalent to a constant number of orthogonal line segment intersection and batched range searching problem instances on the points and segments defining the N rectangles. Thus we obtain the following:

COROLLARY 4. *The pairwise rectangle intersection problem can be solved in $O(n \log_m n + r)$ I/Os.*

That both algorithms are optimal follows from the internal comparison lower bound and the results in [9] and [10]. Like in the orthogonal line segment intersection case, optimal algorithms for the two problems are also developed in [30].

The buffered segment tree, as well as the idea of decreasing the fan-out of the tree to $\sqrt{m}$ and introducing the notion of multislabs, have recently been used in a number of other results (e.g., [16], [13], [12], [11], [17], [1], and [15]). Examples include algorithms for processing line segments in GIS [16] and a general technique for solving (higher-dimensional) batched dynamic searching problems [13]. The practical efficiency of an algorithm for the spatial join problem, based on the rectangle intersection algorithm discussed in this section, has been demonstrated in [12] and [11].

7. Extending the Results to the D-Disk Model. As mentioned in the Introduction, an approach to increase the throughput of I/O systems is to use a number of disks in parallel. One method for using D disks in parallel is *disk striping*—each disk reads or writes a block in the same location as each of the others in an I/O, thus effectively simulating a single large disk with block size $B' = DB$. Even though disk striping does not achieve theoretical asymptotic optimality when D is large, it is often the method of choice in practice for using parallel disks. While sorting N elements using disk striping and one of the one-disk sorting algorithms requires $O((n/D)\log_{m/D} n)$ I/Os, the optimal bound is $O((n/D)\log_m n)$ I/Os [2].

Nodine and Vitter [37] develop a theoretically optimal D-disk sorting algorithm based on merge sort, and later an optimal algorithm based on distribution sort [36]. Several (simple) randomized algorithms have also been developed (e.g., [18], [19], [39], and [44]). The distribution sort algorithm by Nodine and Vitter [36] requires that $4DB \leq M - M^{1/2+\beta}$ for some $0 < \beta < \frac{1}{2}$ (a non-restrictive assumption in practice). The algorithm works as normal distribution sort by repeatedly distributing the set of elements into $\sqrt{m}$ sets of roughly equal size, such that all elements in the first set are smaller than all elements in the second set, and so on. The distribution is performed in $O(n/D)$ I/Os.

To obtain the results in this paper we essentially only used two "paradigms": *distribution* and *merging*. In the batched range tree we used distribution when inserting elements from the buffer of v in the buffers of children of v. In the buffered segment tree we also distributed segments to multislab lists. We used merging of two lists when emptying all buffers in a (sub-) buffered range tree. While it is easy to merge *two* lists in the optimal number of I/Os on parallel disks, we need to modify our use of distribution to make it work optimally with D disks. As mentioned, Nodine and Vitter [36] developed an optimal D-disk distribution algorithm, but only when the distribution is performed $O(\sqrt{m})$-wise. As already mentioned, we can make our buffered range tree work with fan-out and buffer size $\Theta(\sqrt{m})$ instead of $\Theta(m)$, and thus we can use the algorithm by Nodine and Vitter [36] to make our buffered range tree work effectively in the D-disk model. The buffered segment tree already has fan-out $\sqrt{m}$, but we still distribute segments to $\Theta(m)$ multislab lists. Thus to make the buffered segment tree work in the D-disk model, we decrease the fan-out to $m^{1/4}$ thereby decreasing the number of multislab lists to $O(\sqrt{m})$. To summarize, our buffered structures work in the general D-disk model under the non-restrictive assumption that $4DB \leq M - M^{1/2+\beta}$ for some $0 < \beta < \frac{1}{2}$.

8. Conclusion. In this paper we presented a technique for designing efficient batched external data structures. Using the technique we developed an efficient external priority queue and batched versions of range trees and segment trees. We illustrated how these structures allow us to design I/O-efficient algorithms from known internal algorithms in a straightforward way, such that all the I/O-specific parts of the algorithms are "hidden" in the data structures. We also used our priority queue to develop a simple alternative implementation of the so-called time-forward processing technique, improving on the previously known algorithms and consequently a large number of graph algorithms. It remains a challenging open problem to design a priority queue where an element can be updated (have its key decreased) I/O efficiently without knowing the current key.

Our buffer technique and the developed structures have subsequently been used in the development of algorithms for many other problems (refer to recent surveys [6], [5], [43], and [42]). Some experimental results on the practical performance of our structures have also been reported [32], [22]. Even though the buffer technique has been used to solve many problems in batched data structure design, the technique has limitations. For example, the laziness introduced by the buffers creates problems when the elements stored in a buffered structure are drawn from a partial order (see, e.g., [16]).

Acknowledgments. We thank Mikael Knudsen for discussions that lead to many of the results in this paper, Sven Skyum for many computational geometry discussions, and Peter Bro Miltersen, Erik Meineche Schmidt, Darren Vengroff, and Jeff Vitter, as well as an anonymous referee, for discussions and comments that led to improvements in the presentation of the results in this paper.

References

[1] P. K. Agarwal, L. Arge, G. S. Brodal, and J. S. Vitter. I/O-efficient dynamic point location in monotone planar subdivisions. In *Proc. ACM–SIAM Symp. on Discrete Algorithms*, pages 1116–1127, 1999.

[2] A. Aggarwal and J. S. Vitter. The input/output complexity of sorting and related problems. *Communications of the ACM*, 31(9):1116–1127, 1988.

[3] L. Arge. The buffer tree: a new technique for optimal I/O-algorithms. In *Proc. Workshop on Algorithms and Data Structures*, pages 334–345. LNCS 955. Springer-Verlag, Berlin, 1995.

[4] L. Arge. The I/O-complexity of ordered binary-decision diagram manipulation. In *Proc. Internat. Symp. on Algorithms and Computation*, pages 82–91. LNCS 1004, Springer-Verlag, Berlin, 1995. A complete version appear as BRICS Technical Report RS-96-29, University of Aarhus.

[5] L. Arge. External-memory algorithms with applications in geographic information systems. In M. van Kreveld, J. Nievergelt, T. Roos, and P. Widmayer, editors, *Algorithmic Foundations of GIS*, pages 213–254. LNCS 1340. Springer-Verlag, Berlin, 1997.

[6] L. Arge. External memory data structures. In J. Abello, P. M. Pardalos, and M. G. C. Resende, editors, *Handbook of Massive Data Sets*, pages 313–358. Kluwer, Dordrecht, 2002.

[7] L. Arge, P. Ferragina, R. Grossi, and J. Vitter. On sorting strings in external memory. In *Proc. ACM Symp. on Theory of Computation*, pages 540–548, 1997.

[8] L. Arge, K. H. Hinrichs, J. Vahrenhold, and J. S. Vitter. Efficient bulk operations on dynamic R-trees. *Algorithmica*, 33(1):104–128, 2002.

[9] L. Arge, M. Knudsen, and K. Larsen. A general lower bound on the I/O-complexity of comparison-based algorithms. In *Proc. Workshop on Algorithms and Data Structures*, pages 83–94. LNCS 709. Springer-Verlag, Berlin, 1993.

[10] L. Arge and P. B. Miltersen. On showing lower bounds for external-memory computational geometry. In J. Abello and J. S. Vitter, editors, *External Memory Algorithms and Visualization*, pages 139–160. DIMACS series, volume 50. American Mathematical Society, Providence, RI, 1999.

[11] L. Arge, O. Procopiuc, S. Ramaswamy, T. Suel, J. Vahrenhold, and J. S. Vitter. A unified approach for indexed and non-indexed spatial joins. In *Proc. Conf. on Extending Database Technology*, pages 413–429, 1999.

[12] L. Arge, O. Procopiuc, S. Ramaswamy, T. Suel, and J. S. Vitter. Scalable sweeping-based spatial join. In *Proc. Internat. Conf. on Very Large Databases*, pages 570–581, 1998.

[13] L. Arge, O. Procopiuc, S. Ramaswamy, T. Suel, and J. S. Vitter. Theory and practice of I/O-efficient algorithms for multidimensional batched searching problems. In *Proc. ACM–SIAM Symp. on Discrete Algorithms*, pages 685–694, 1998.

[14] L. Arge, L. Toma, and J. S. Vitter. I/O-efficient algorithms for problems on grid-based terrains. In *Proc. Workshop on Algorithm Engineering and Experimentation*, 2000.

[15] L. Arge and J. Vahrenhold. I/O-efficient dynamic planar point location. In *Proc. ACM Symp. on Computational Geometry*, pages 191–200, 2000.

[16] L. Arge, D. E. Vengroff, and J. S. Vitter. External-memory algorithms for processing line segments in geographic information systems. In *Proc. European Symp. on Algorithms*, pages 295–310. LNCS 979. Springer-Verlag, Berlin, 1995. To appear in special issues of *Algorithmica* on Geographical Information Systems.

[17] L. Arge and J. S. Vitter. Optimal dynamic interval management in external memory. In *Proc. IEEE Symp. on Foundations of Computer Science*, pages 560–569, 1996.

[18] R. D. Barve, E. F. Grove, and J. S. Vitter. Simple randomized mergesort on parallel disks. *Parallel Computing*, 23(4):601–631, 1997.

[19] R. D. Barve and J. S. Vitter. A simple and efficient parallel disk mergesort. In *Proc. ACM Symp. on Parallel Algorithms and Architectures*, pages 232–241, 1999.

[20] R. Bayer and E. McCreight. Organization and maintenance of large ordered indexes. *Acta Informatica*, 1:173–189, 1972.

[21] J. L. Bentley and D. Wood. An optimal worst case algorithm for reporting intersections of rectangles. *IEEE Transactions on Computers*, 29:571–577, 1980.

[22] K. Brengel, A. Crauser, P. Ferragina, and U. Meyer. An experimental study of priority queues in external memory. *ACM Journal on Experimental Algorithmics*, 5, article 17, 2001.

[23] G. S. Brodal and J. Katajainen. Worst-case efficient external-memory priority queues. In *Proc. Scandinavian Workshop on Algorithms Theory*, pages 107–118. LNCS 1432. Springer-Verlag, Berlin, 1998.

[24] A. L. Buchsbaum, M. Goldwasser, S. Venkatasubramanian, and J. R. Westbrook. On external memory graph traversal. In *Proc. ACM–SIAM Symp. on Discrete Algorithms*, pages 859–860, 2000.

[25] Y.-J. Chiang, M. T. Goodrich, E. F. Grove, R. Tamassia, D. E. Vengroff, and J. S. Vitter. External-memory graph algorithms. In *Proc. ACM–SIAM Symp. on Discrete Algorithms*, pages 139–149, 1995.

[26] D. Comer. The ubiquitous B-tree. *ACM Computing Surveys*, 11(2):121–137, 1979.

[27] A. Crauser and K. Mehlhorn. LEDA-SM: extending LEDA to secondary memory. In *Proc. Workshop on Algorithm Engineering*, 1999.

[28] H. Edelsbrunner and M. Overmars. Batched dynamic solutions to decomposable searching problems. *Journal of Algorithms*, 6:515–542, 1985.

[29] R. Fadel, K. V. Jakobsen, J. Katajainen, and J. Teuhola. Heaps and heapsort on secondary storage. *Theoretical Computer Science*, 220(2):345–362, 1999.

[30] M. T. Goodrich, J.-J. Tsay, D. E. Vengroff, and J. S. Vitter. External-memory computational geometry. In *Proc. IEEE Symp. on Foundations of Computer Science*, pages 714–723, 1993.

[31] S. Huddleston and K. Mehlhorn. A new data structure for representing sorted lists. *Acta Informatica*, 17:157–184, 1982.

[32] D. Hutchinson, A. Maheshwari, J.-R. Sack, and R. Velicescu. Early experiences in implementing the buffer tree. In *Proc. Workshop on Algorithm Engineering*, pages 92–103, 1997.

[33] D. E. Knuth. *The Art of Computer Programming*, volume 3. Addison-Wesley, Reading, MA, second edition, 1998.

[34] V. Kumar and E. Schwabe. Improved algorithms and data structures for solving graph problems in external memory. In *Proc. IEEE Symp. on Parallel and Distributed Processing*, pages 169–177, 1996.

[35] K. Mehlhorn. *Data Structures and Algorithms*, 1: *Sorting and Searching*. EATCS Monographs on Theoretical Computer Science. Springer-Verlag, New York, 1984.

[36] M. H. Nodine and J. S. Vitter. Deterministic distribution sort in shared and distributed memory multiprocessors. In *Proc. ACM Symp. on Parallel Algorithms and Architectures*, pages 120–129, 1993.

[37] M. H. Nodine and J. S. Vitter. Greed Sort: optimal deterministic sorting on parallel disks. *Journal of the ACM*, 42(4):919–933, 1995.

[38] F. P. Preparata and M. I. Shamos. *Computational Geometry: An Introduction*. Springer-Verlag, New York, 1985.

[39] P. Sanders, S. Egner, and J. Korst. Fast concurrent access to parallel disks. In *Proc. ACM–SIAM Symp. on Discrete Algorithms*, pages 849–858, 2000.

[40] J. van den Bercken and B. Seeger. An evaluation of generic bulk loading techniques. In *Proc. Internat. Conf. on Very Large Databases*, pages 461–470, 2001.

[41] J. van den Bercken, B. Seeger, and P. Widmayer. A generic approach to bulk loading multidimensional index structures. In *Proc. Internat. Conf. on Very Large Databases*, pages 406–415, 1997.

[42] J. S. Vitter. Online data structures in external memory. In *Proc. Internat. Colloq. on Automata, Languages, and Programming*, pages 119–133. LNCS 1644. Springer-Verlag, Berlin, 1999.

[43] J. S. Vitter. External memory algorithms and data structures: dealing with MASSIVE data. *ACM Computing Surveys*, 33(2):209–271, 2001.

[44] J. S. Vitter and D. A. Hutchinson. Distribution sort with randomized cycling. In *Proc. ACM–SIAM Symp. on Discrete Algorithms*, pages 77–86, 2001.

[45] J. S. Vitter and E. A. M. Shriver. Algorithms for parallel memory, I: Two-level memories. *Algorithmica*, 12(2–3):110–147, 1994.