Algorithms for Memory Hierarchies
Lecture 4

Lecturer: Nodari Sitchinava
Scribe: Emanuel Schrade

1 Buffer Trees

1.1 Motivation

We can build a binary search tree given N items by adding each item after another to it. This approach takes $\mathcal{O}(\log N)$ time per insertion and hence a total of $\mathcal{O}(N \log N) = sort(N)$ time for inserting N items into the binary tree. If we wanted to build a B-tree using the same approach, the I/O-complexity would be $\mathcal{O}(\log_B N)$ I/O's for inserting each element resulting in $\mathcal{O}(N \log_B N)$ I/O's for inserting N elements. Note that $\mathcal{O}(N \log_B N) \gg sort(N)$, i.e., this construction takes way more I/Os than the I/O-complexity of sorting N elements.

In contrast to an online construction of the B-tree, where each operations must be performed to completion immediately at the time when it occurs, we can achieve a better I/O-complexity for building the B-tree using a so-called batched, or offline, construction technique. For the offline-construction we require all operations to be known upfront, before building the B-tree and all op-

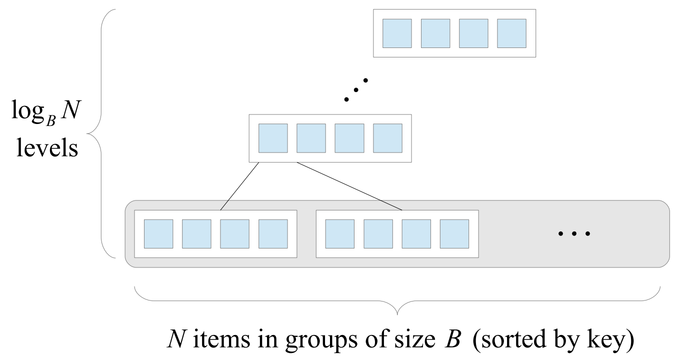

Figure 1: Batched bottom-up constriction of a B-tree given N items

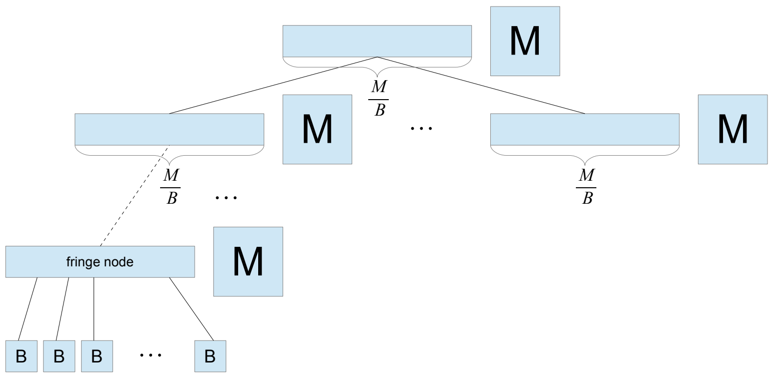

Figure 2: Buffer tree

erations must be completed eventually, i.e., before the construction is finished. Then we can build the B-tree by first sorting the leaves with an I/O-complexity of $\mathcal{O}(sort(N))$ I/O's and then building the tree in a bottom-up manner with an I/O-complexity of $\mathcal{O}(\frac{N}{B}\log_B N)$ by grouping always B leaves / nodes of a level to build a node in the next level (see Figure 1). The building process can be further improved to the I/O-complexity of sorting N elements.

However, it is not clear how to use this approach to build the persistent B-trees. *Buffer tree* is a data structure that will allow us to construct a persistent B-tree on N operations in $\mathcal{O}(sort(N))$ I/O's.

1.2 Buffer Trees

Buffer tree is an (a, b)-tree with leaves of size $\Theta(B)$ and of internal nodes of size $\Theta\left(\frac{M}{B}\right)$. Thus, a tree accomodating N elements has a height of $\mathcal{O}\left(\log_{\frac{M}{B}} \frac{N}{B}\right)$. Each internal node is augmented with a buffer of size M. From now on we refer to the parents of leaves as *fringe nodes*. An example for a buffer tree is depicted in Figure 2. The buffer tree works as follows:

- Each element contains a time stamp as in persistent B-trees.
- We collect B elements in the internal memory and write them to the root buffer.
- When the (root's) buffer gets full, we empty it by distributing its content to its childs' buffers.
- If a child's buffer is full, we recursively empty it too.

Propagating a buffer's content to fringe nodes might make rebalancing necessary.

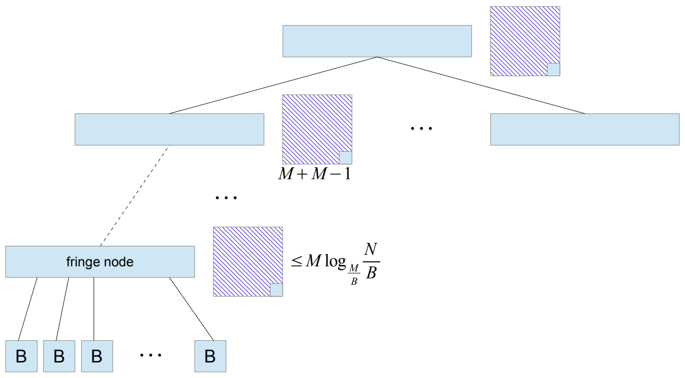

Figure 3: buffers growing beyond M elements during buffer emptying

1.3 Emptying non-fringe node buffers

The (full) buffers are emptied in the following way:

1. Load the M elements of the full buffer and sort them primarily by their key and secondarily by their timestamp.
2. Merge them with other elements.
3. Process elements with the same key.
4. Distribute remaining elements to child buffers in sorted order.

For now we can just delete matching insertions and deletions in step 3. When it comes to persistent B-trees, we would mark elements as deleted, rather than deleting them, as we did for persistent B-trees.

Sorting the elements might be a problem as shown in Figure 3. In case all elements are propagated to the same fringe node more than M elements might need to be emptied from a buffer i.e. sorted (see Figure 3). However, note that elements have been propagated from the parent in sorted order and at most M elements in the buffer are unsorted. That is why we sort only the first M elements and merge them with the rest.

Analysis of the emptying algorithm's I/O-complexity (for each node):

1. Loading M elements: $\mathcal{O}(\frac{M}{B})$ I/O's
2. Merging X elements with the M new ones: $\mathcal{O}(\frac{X}{B} + \frac{M}{B})$ I/O's
3. Processing and distributing the elements: $\mathcal{O}(\frac{X}{B} + \frac{M}{B})$ I/O's

In total, emptying a full buffer takes $\mathcal{O}(\frac{X}{B})$ I/O's if $X > M$.

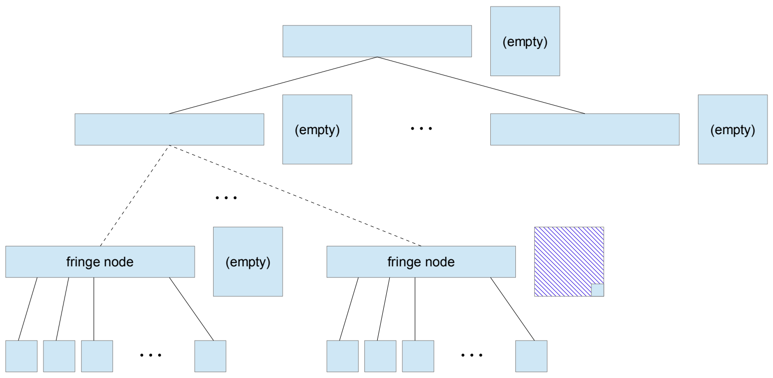

Figure 4: Invariant

1.4 Emptying fringe node buffers

As mentioned in Section 1.2 rebalancing may be required when emptying fringe node buffers. We have to adapt our aproach from Section 1.3 a bit:

1. Load the M elements of the full buffer, sort them, and merge them with other elements.
2. Merge all elements of leaves with elements from the buffer
3. Process elements with the same key. Let K' be the number of elements that remain after this step.
4. *if* $K' < K$ create $K - K'$ "dummy" elements to delay the rebalancing till later.
   else create one leaf at a time and rebalance as in B-trees.

1.5 Buffer emptying

For the whole buffer emptying algorithm we require the following invariant to hold at any time: **All buffers on the path from the buffer emptying node to the root are empty.** Also we empty all non-fringe nodes, that have to be emptied, first, before emptying any fringe node.

By this invariant we avoid special cases that might appear when rebalancing the tree. Figure 4 shows an example where we are currently emptying the leftmost fringe node's buffer. The buffer emptying algorithm results in:

1. Empty all non-fringe full buffers.
2. Empty fringe buffers (dealing with node splits)
3. Remove each dummy node and rebalance
   If fusion is required, empty the buffer of sibling node and fuse nodes. This

step might be necessary to be executed recursively upwards. The sibling's buffer has to be emptied even though it isn't full, in this case.

1.6 Analysis of the buffer tree

Let's analyze the I/O complexity of a buffer tree after processing N elements. As shown in Section 1.3 emptying full buffers takes $\mathcal{O}(\frac{N}{B})$ I/O's per level. Hence, excluding the rebalancing, we get an I/O-complexity of

$$\mathcal{O}\Bigg(\underbrace{\frac{N}{B}}_{per\ level} \cdot \underbrace{\log_{\frac{M}{B}} \frac{N}{B}}_{height}\Bigg)$$

for emptying full buffers on all levels. However, in case of rebalancing non-full buffers have to be emptied. Each time a non-full buffer is emptied, we spend $\mathcal{O}(\frac{M}{B})$ I/O's. Let's count how many times this will happen. The number of rebalances is $\frac{N}{B}$ times for the leaves (level 0), $\frac{\frac{N}{B}}{\frac{M}{B}}$ times for their parents (level 1), $\dots$, $\frac{N}{B \cdot (\frac{M}{B})^k}$ times for the nodes at the k-th level from the bottom.

Hence the number of rebalances sums up to

$$\sum_{i=1}^{\#levels} \frac{N}{B \cdot (\frac{M}{B})^i} = \mathcal{O}\left(\frac{N}{B \cdot (\frac{M}{B})}\right)$$

which is the number of how often we empty non-full buffers, each taking $\mathcal{O}(\frac{M}{B})$ I/O's. Thus the total I/O-complexity of emptying non-full buffers is

$$\mathcal{O}\left(\frac{N}{B \cdot (\frac{M}{B})}\right) \cdot \mathcal{O}\left(\frac{M}{B}\right) = \mathcal{O}\left(\frac{N}{B}\right) \text{ I/O's}$$

In total the I/O-complexity of processing N elements in the buffer tree is:

$$\mathcal{O}\left(\frac{N}{B} \cdot \log_{\frac{M}{B}} \frac{N}{B}\right) \in sort(N).$$

1.7 Sorting

We can sort N items by inserting them into the buffer tree, emptying all buffers that haven't been emptied yet and reporting all elements in the leaves of the buffer tree.

We need to compute the I/O-complexity of the emptying of the remaining non-full buffers after all insertions. We empty these buffers level by level starting from root. There are a total of $\mathcal{O}\left(\frac{N}{M}\right)$ buffers in the buffer tree. Emptying each buffer takes $\mathcal{O}\left(\frac{M}{B}\right)$ I/O's. Thus, the total complexity of this step is $\mathcal{O}\left(\frac{N}{M} \cdot \frac{M}{B}\right) = \mathcal{O}\left(\frac{N}{B}\right)$ I/O's. Thus, sorting using buffer trees is dominated by other operations of the buffer tree and is $\mathcal{O}\left(\frac{N}{B} \log_{\frac{M}{B}} \frac{N}{B}\right)$.

1.8 Priority Queue

We implement a priority queue using a buffer tree as follows. Insertion of elements into the priority queue is a standard insertion into the buffer tree as described above. When performing a delete-min operation observe that the smallest element is in the leftmost leaf or in any of the buffers on the path from the root to the lefmost leaf. Thus, to perform a delete-min operation we must empty all buffers on this path, which takes

$$\mathcal{O}\left(\underbrace{\frac{M}{B}}_{\text{to empty each buffer}} \cdot \underbrace{\log_{\frac{M}{B}} \frac{N}{B}}_{\text{height}}\right) \text{ I/O's.}$$

As buffer emptying operations are expensive, we keep $\Theta(M)$ smallest elements in the internal memory and maintain them during insertions. Then, the next $\Theta(M)$ delete-min operations are free in terms of I/O-complexity. Thus, one delete-min operation takes

$$\mathcal{O}\left(\frac{1}{B} \cdot \log_{\frac{M}{B}} \frac{N}{B}\right)$$

I/O's amortized. To delete an element defined by a *key*, we insert a sentinel with the same *key*, representing a deletion. If a deleted element ever becomes the smallest element in the priority queue, the sentinel will be the smallest as well and we can delete them from the priority queue instead of reporting.

1.9 Additional applications of buffer trees

- **EM segment trees:** report all intersecting rectangles
- **Range Queries:** find points in rectangle; for geometric applications or databases
- **EM interval trees:** report all intersections of line segments
- **EM string sorting:** lexicographic sorting of strings, or sorting numbers with variable length keys