1 B-Trees (Cont.)

For a B-tree, all leaves form a linked list:

- scan all elements $i \log_B N + \frac{N}{B}$
- scan $k$ records $i \log_B N + \frac{K}{B}$

2 $B^\epsilon$-Trees (Buffered B-Trees)

**Definition 1.** Associate a *buffer* of elements with each internal node in the tree, containing elements to be inserted into that node’s subtree. Whenever a buffer become full, its content are pushed down to the node’s children using small number of I/Os.

Given $0 \leq \epsilon \leq 1$, $B^{\epsilon}$ is the number of childrens and $B^{1-\epsilon}$ is additional space for buffer. And each internal node must have $\geq \frac{B^{\epsilon}}{2}$ children.

2.1 Insert

- Insertion is the only operation required for sorting.
• _Insert_ in root takes constant time at right buffer.

• _Lookups_ must examine buffers along the way.

The top-level insertion algorithm simply records the new item in a block in internal memory. When the block is full, sort the block, write it into buffer at the root and start a new block.

![Figure 2.1: Buffer Tree](image)

### 2.2 Flush from Internal Node to Leaf

If the root buffer is full, the buffer is emptied and the operations in the buffer are _flushed_ to the next buffer levels as follows:

- First, read the first M items of the buffer into main memory, sort them and merge with the remainder of the buffer (which is already sorted).

- Second, distribute items in the sorted buffer to the buffers in the children of the root, maintaining the invariant that each buffer contains at most one non-full block.

- Finally, if any of the child buffers is full, empty it recursively.

Because the leaves do not have buffers, the parents of leaves have a slightly different buffer-empty process. Assume the node has leaves as children, then:

- First, read the first M items of the buffer into main memory, sort them and merge with the remainder of the buffer. The items stored in the leaves.

- Next, write the first blocks of items of the expanded buffer into the leaves.

- Finally, add the remaining blocks of the expanded buffer as new leaves to the node one at a time. If at any time, the node has too many leaves, split the node as the normal B-tree insertion algorithm.
The amortized analysis computes the total time for \( N \) operations divided by \( N \).

The amortized analysis of Buffered B-tree is:

- Total number of items in tree is \( N \)
- Tree depth is \( O \left( \log \frac{2^\epsilon}{\epsilon} N \right) = O \left( \frac{\log N}{\log B^\epsilon} \right) = O \left( \frac{1}{\epsilon} \log_B N \right) \)
- Each item can be flushed at most \( O \left( \frac{1}{\epsilon} \log_B N \right) \)
- \( \leq O \left( \frac{1}{\epsilon} \log_B N \right) \) item-flushes
- Each I/O accomplishes \( B^{1-\epsilon} \) item-flushes
- So total I/O is \( O \left( \frac{N \cdot \frac{1}{\epsilon} \log_B N}{B^{1-\epsilon}} \right) \)
- Amortized work: \( O \left( \frac{N \cdot \frac{1}{\epsilon} \log_B N}{N \cdot B^{1-\epsilon}} \right) = O \left( \frac{\log_B N}{B^{1-\epsilon}} \right) \) is the operation cost of insert performed

The lookups depends on depths of tree \( O \left( \frac{1}{\epsilon} \log_B N \right) \) and inserts is \( O \left( \frac{\log_B N}{B^{1-\epsilon}} \right) \).

When \( \epsilon = 1 \), then the operation of lookups is \( O \left( \log_B N \right) \) and inserts is \( O \left( \log_B N \right) \). When \( \epsilon = \frac{1}{2} \), the operation of lookups is \( O \left( 2 \log_B N \right) \) and inserts is \( O \left( \frac{2 \log N}{\sqrt{B}} \right) \). Thus, in some circumstance, giving up some look up speed can tremendously increase the insert speed.