Bε-tree: Data structure to organized on disk storage

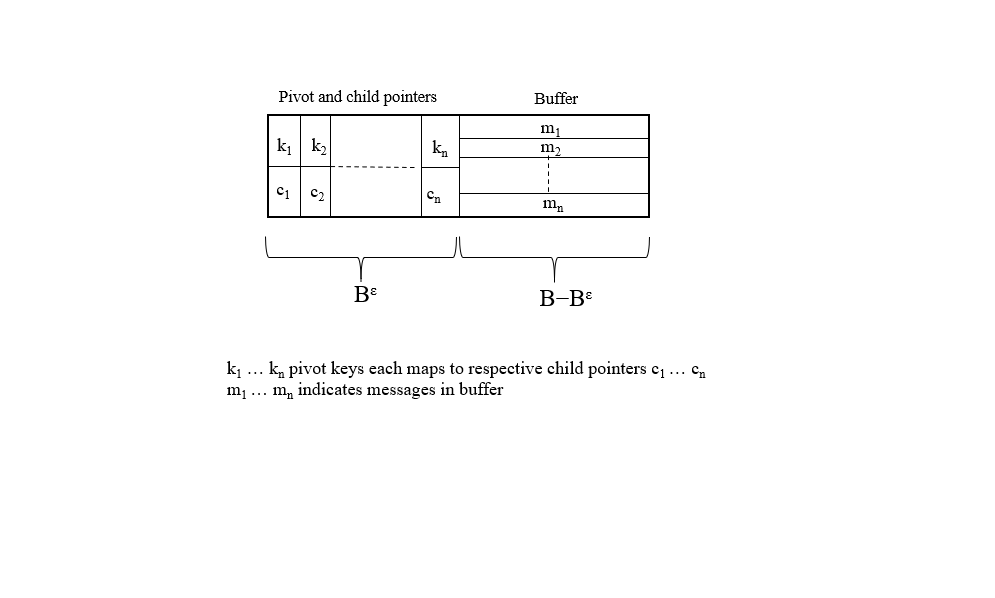
**Introduction:**

Bε-tree is a data structure which can be used to organized on disk storage for application such as databases or file system. A B epsilon uses key-value pair as similar to B tree but provides better performance, particularly for inserts, range queries, and key-value updates. It is composition of B tree and buffered repository tree where B tree favors query operation and a buﬀered repository tree favors insertion.

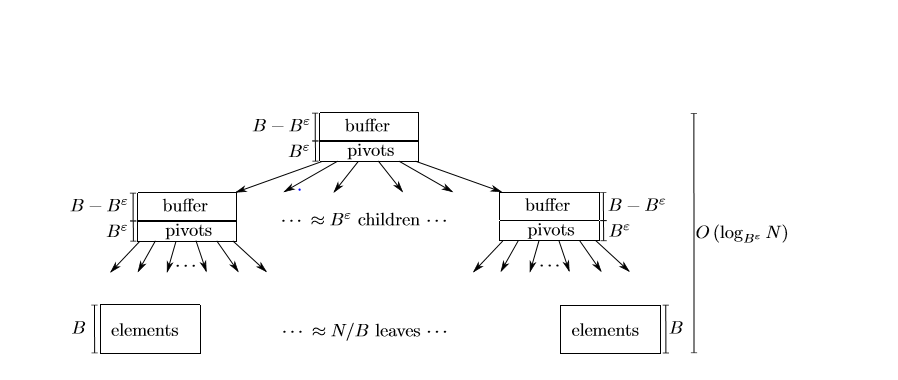
**Basic structure:**

The node size of B epsilon is chosen to be multiple of block size of underlying storage device. Similar to B-tree, in Bε -tree , internal node consists of pivot keys and pointer to children and leaves store key value pair sorted by key. The distinguishing feature of a Bε -tree is that internal node also allocated some space for a buffer which can be used to store messages for operations. Such encoded updates will eventually be applied to leaves under this node. The size allocated to buffer will depends on the value of ε, which must be between 0 and 1. It is a tuning parameter for selection of space for picots pivots (≈ Bε) and buﬀer (≈ B−Bε).

Node structure:



Bε-tree :



**Implementation:**

Insertion:

Insertion are encoded messages, which will contain key, value and operation parameters and added to buffer of root node of the tree. When buffer size exceeds than allocated buffer size, it will flush the messages to one of the node’s children with the most pending messages. Over the course of flushing, each message is ultimately delivered to the appropriate leaf node. When the leaf node becomes full, it splits , just as a B-tree. When an interior node gets too many children, it splits and the messages in its buﬀer are distributed between the two new nodes.

Deletion:

Similar to insertion, deleted messages are encoded and are flushed down the tree until they reach a node with corresponding key which needs to be deleted. When a message is ﬂushed to the such node, the Bε-tree discards both the deleted item and the message. Thus, a item which needs to be deleted can continue to exist until a message reaches to that item.

**Performance:**

Moving messages down the tree in batches is the key to the Bε-tree’s insert performance. By storing newly-inserted messages in a buﬀer near the root, a Bε-tree can avoid seeking all over the disk to put elements in their target locations. The Bε-tree only moves messages to a subtree when enough messages have accumulated for that subtree.

Moreover, the Bε -tree ﬂushing strategy is designed to ensure that it can always move elements in large batches. Messages are only ﬂushed to a child when the buﬀer of a node is full, containing B − Bε ≈ B messages. When a buﬀer is ﬂushed, not all messages are necessarily ﬂushed— messages are only ﬂushed to children with enough pending messages to oﬀset the cost of rewriting the parent and child nodes. Consequently, any node in a Bε-tree is only rewritten if a suﬃciently large portion of the node will change.

**The role of ε:**

Parameter ε ranges between 0 and 1. As, ε is deciding factor for size allocation for pivots and child pointer (≈ Bε) and buffer (≈ B − Bε),it will influence the insertion and point query performance. As ε increases, so does the branching factor (Bε), causing the depth of the tree to decrease and searches to run faster. As ε decreases, the buﬀers get larger, batching more inserts for every ﬂush and improving overall insert performance.

For example, when ε = 1, a Bε -tree is just a B-tree, since interior nodes contain only pivot keys and child pointers. On the other, when ε = 0, a Bε -tree is a binary search tree with a large buﬀer at each node, called a buﬀered repository tree.

**Example:**

