Variant B^{ε}-Tree: How to implement efficient B^{ε}-Tree in Key Value SSD

Seri Lee
Seoul National University
sally20921@snu.ac.kr

***Abstract*—**

***Index Terms*—B^{ε}-Tree, Key Value SSD**

I. Introduction

II. Background and Motivation

A. B-Tree

Binary search trees are not good for locality because a given node of the binary tree occupies only a fraction of any cache line. B-trees are a way to get better locality by putting multiple elements into each tree node. B-tree was originally invented for storing data structures on disk, where locality is even more crucial than with memory. Because the height of the tree is uniformly the same and every node is at least half full, we are guaranteed that the asymtotic performance is $O(\log n)$ where n is the size of the collection.

Search

Algorithm 1 Get(K)

```
Require: K: key
  start at the root
  recursively traverse from top to bottom
```

Insertion

An improved algorithm supports a single pass down the tree from the root to the node where insertion will take place, splitting any full nodes encountered on the way. This prevents the need to recall the parent nodes into memory, which may be expensive if the nodes are on secondary storage.

Algorithm 2 Put(K, V)

```
Require: K: key, V: Value
  find the leaf node $L$ where the new element should be added
  if $L$ ¡ $max$ then          ▷ This process happens recursively
      insert
      keep the node sorted
  else                          ▷ $L$ = full
      single median $m$ is chosen
      if value ¡ $m$ then
          put in to new leaf node
      $m$ inserted into $L$'s parent
```

Deletion

There are two popular strategies for deletion, but below uses the former strategy. First, to locate and delete the item, then restructure the tree to regain its invariants. Or, Do a single pass down the tree, but before entering a node, restructure the tree so that once the key to be deleted is encountered, it can be deleted without triggering the need for further restructuring.

Variants

In the narrow sense, a B-tree stores keys in its internal nodes but need not store those keys in the records at the leaves. However, in B^{+}-tree, copies of the keys are stored in the internal nodes and the keys and records are stored in leaves. A leaf node also include a pointer to the next leaf node to speed sequential access.

Algorithm 3 Delete(K)

```
Require: K: key
  find the node $N$ that needs to be deleted
  if $N$ = leaf node then
      delete it from $N$
      if underflow then
          fuse siblings together
          function REBALANCE(tree)
  else                                        ▷ $N$ = internal node
      if both the child nodes have the minimum number of elements then ▷ this process happens recursively
          join into a single node
      if one of the two child nodes contain more than minimum number of elements then
          find new separator
          replace                             ▷ $N$ = special case
```

Algorithm 4 Rebalance(tree)

```
Require: tree: B-tree where deletion happened
  if leaf node ¡ $min$ then    ▷ make the root node be the only deficient node
      add separator to end of deficient node from siblings
      replace separator in the parent
```

B. B^{ε}-Tree

B^{ε}-Tree is a B-tree that uses per-node buffering to improve inserts. By using these buffers, B^{ε}-trees are able to batch insert operations to amortize their cost. ε is a design-time constant between [0,1] and is the ratio used for buffering, where as the rest of the space in each node $(1-\varepsilon)$ is used for storing pivots. The main distinction between B^{ε}-tree and B-tree is that interior B^{ε}-tree nodes are augmented to include message buffers. A B^{ε}-tree models all changes (insert, deletes, upserts) as messages. A key technique behind write-optimization is that message can accumulate in a buffer, and are flushed down the tree in larger batches, which amortize the costs of rewriting a node. Most notably, this batching can improve the costs of small, random writes by orders of magnitude. In order to make searching and inserting into buffers efficient, the message buffers within each node are typically organized into a balanced binary search tree, such as a red-black tree. Messages in the buffer are sorted by their target key, followed by timestamp. The timestamp ensures that messages are applied in the correct order. Thus, inserting a message into a buffer, searching within a buffer, flushing from one buffer to another are all fast.

Insertion

Insertions are encoded as ”insert messages”. When enough messages have been added to a node to fill the node’s buffer, a batch of messages are flushed to one of the node’s children.

Query

Messages addressed to a key k are guaranteed to be applied to k’s leaf or in some buffer along the root-to-leaf path towards key k. This invariant assures that point and range queries in B^{ε}-tree have a similar I/O cost to a B-tree. **Point Query** Visit each node from the root to the correct leaf. Also check the buffers in nodes on this path for messages, applying relevant messages before returning the results of the query. **Range Query** The messages for the entire range of keys must be checked and applied as the appropriate subtree is traversed.

Deletion

B^{ε}-Tree delete items by inserting ”tombstone messages” into the tree. These tombstone messages are flushed down the tree until they reach a leaf. When a tombstone message is flushed to a leaf, the tree discards both the deleted item and the tombstone message.

C. LSM-Tree

State-of-the-art key-value stores which typically use an LSM-tree structure have been designed primarily for HDDs. LSM-trees are a write-optimized structure that is a good fit for HDDs where there is a large difference in performance between random and sequential accesses. However, the drawback for LSM-tree derives from the fact that they keep large sorted containers (which later compactions are performed in). This incurs high CPU overhead and results in I/O amplification for reads and writes.

D. Why use B^{ε}-Tree

Compared to LSM-trees, B^{ε}-trees incur less I/O amplification. B^{ε}-tree use an index, compared to LSM-trees, in order to remove the need for sorted containers. This results in smaller and more random I/Os. As device technology reduces the I/O size required to achieve high throughput, using a B^{ε}-tree instead of an LSM-tree is a reasonable decision.

III. Conclusion

References

[1]
[2]