A Symmetric Concurrent B-Tree Algorithm

Part 2: Implementation
Overview: B-Link Tree

- Like a B+ tree, actual keys are in leaf nodes. “Separators” in internal nodes are for navigation only.
- Contains extra links (“rightlinks” and “outlinks”) so that threads searching for something will still be able to find it if the tree is reconfigured during their search.
  - This allows it to use less locking than other types of concurrent B-tree algorithm.
Example of a B-Link Tree (diagram from Lanin & Shasha):

Root with only one separator (formerly had more). Skipped by using anchor’s *fast* pointer in place of actual root.

- **fast pointer**
- **anchor**
- **top pointer**

**outlink** (link from an empty node to a non-empty node)

**rightlinks** (link all nodes in a level)

Some nodes are only accessible through rightlinks.
Structure Properties

• Form is less strict than a normal B-tree or B+ tree.
• Most important properties:
  – All keys in each level must be sorted.
  – All leaves must be on the same level.
  – No key can move left after tree modification, or appear to the left of where an above separator indicates.
  – If a key is located (or moved) farther to the right, it can be found by following rightlinks.
• A search can be started at any level.
  – Starting at the leaf level is equivalent to searching a sorted list.
Implementation

• The original algorithm, written in 1986 with Pascal in mind, is structured in a very non-object oriented way.
  – E.g. Static methods call other static methods which in turn call other static methods, with enumerator parameters that choose their behavior.
• My implementation is a combination of object-oriented and non-object-oriented techniques.
Original Algorithm: **search**
(Each box represents a static method.)
Java Implementation

• Many methods in the original algorithm take only a Node, or Node & lock type, as a parameter.
• For my implementation, these were assembled into Node objects.
• Most other static methods were placed in a BLinkTree object.
Implementation: Concurrency

• The algorithm details a fine-grained locking scheme, using read and write locks.
  – Recall: The purpose of the algorithm is to minimize locking.
• This was translated directly to Java using `ReentrantReadWriteLock`.
• `CyclicBarrier` was used to coordinate threads during testing.
• `Synchronized` methods were used for things external to the algorithm itself.
  – E.g. Writing to `System.out`, verifying correctness of the structure.
Java Implementation: Classes
(shows methods from original algorithm only)

Node
node : Node
tvalue : int
check_key(int) : boolean
numberofchildren() : int
rightsep() : int
empty() : boolean
outlink() : Node
rightlink() : Node
too_crowded() : boolean
too_sparse() : boolean
lock(LockType)
unlock(LockType)

Anchor
anchorlock : ReentrantReadWriteLock
fast : Node
fastHeight : int
top : Node
topHeight : int
lock(LockType)
unlock(LockType)

TreeNode
package::java.lang

DescendThread
t : Task
sep : int
child : Node
theight : int
descend : Stack<NodeValuePair>
tree : BLinkTree
run() : void

Critic
anchor : Anchor
run() : void

BLinkTree
anchor : Anchor
critic : Critic
search(int) : boolean
insert(int) : boolean
delete(int) : boolean
add_or_remove_link(Task, int, Node, Node, int, Stack<NodeValuePair>) : boolean
grow() : void
locate_internal(int, int, Stack<NodeValuePair>) : Node
locate_leaf(int, LockType, Stack<NodeValuePair>) : LeafNode
normalize(Node, Stack<NodeValuePair>, int) : void

LeafNode
dkeys : Vector<Integer>
singleLeafSep : int
add_key(int) : boolean
remove_key(int)

InternalNode
downlinks : Vector<Node>
separators : Vector<Integer>
add_link(int, Node) : boolean
find(int, int) : NodeValuePair
leftmostchild() : Node
remove_link(int, Node) : boolean
rightsep(Node) : int
Correctness Testing

• The most important factor is that every item in the tree can be found at all stages of the tree’s life, including while it is being modified by other threads.
• Each test run was started with a set of odd numbers inserted into the tree.
• About 50% of the threads searched for these numbers during initial testing and during the performance tests.
• Insertion and deletion were made to only modify even numbers, so that the search threads would always find what they were looking for, provided the code was correct.
• Any time a search thread did not find a key, it kept a record of this.
• The total number of keys not found was counted after each test.
• After initial debugging was complete, every single test run finished without any un-found keys.
Performance Testing

• Used:
  – MTL
  – My PC (4 core, 4GB RAM)

• Tested against:
  – Sequential version
    • All locks and thread-spawning were removed.
  – No-merge version
    • Merge operation was removed.
    • Tree is not rebalanced to correct for nodes that have too few keys after deletion.
    • Similar to Lehman & Yao’s original B-link algorithm, which this algorithm was based on.
B-tree with merge:

B-tree without merge:
Variables

• Size of nodes = 8
• Number of threads
• Number of cores
• Size of tree
• Maximum key in tree
  – Affects maximum size (no duplicate keys)
• Proportion of threads for each of:
  – Search
  – Insert
  – Delete
Average Throughput

- concurrent - no merge
- concurrent

**Number of Cores**

**Average Throughput (operations/millisecond)**
Average Throughput
(including sequential)
Throughput Using 1 Core

- No Merge
- Concurrent
- Sequential

Average throughput (operations/millisecond)
Increasing Number of Deleted Keys
(maximum key: 100, keys/thread: 100, with 10 search threads and 20 threads total)

average throughput (operations/millisecond)

- Concurrent - No Merge
- Concurrent
- Sequential

number of delete threads (out of 20)
Questions?