A B^{link} Tree latch method and protocol to support synchronous node deletion

Karl Malbrain
malbrain@yahoo.com

Introduction

A new B^{link} Tree latching method and protocol simplifies implementation of Lehman and Yao's high concurrency B^{link} Tree with Jaluta's balanced B^{link} Tree methods with increased concurrency and symmetry between insertions and deletions.

The methods presented by Jaluta compromise concurrency. As an update search proceeds down the B^{link} Tree, latches to the child node is obtained while the parent node "U" latch is held. Any delays in obtaining the child's "U" latch will curtail concurrent update searches through the parent since "U" latches are incompatible with one another. During node splits and consolidations, an "X" latch is held on the child node while the new upper fence values are posted into the parent node, shutting down all access to the node. The latch protocol presented here addresses these compromises.

The methods presented here use the underlying Operating System to implement node latches, removing this responsibility from the B^{link} Tree code. Unix/Linux/WIN32 support exclusive and sharable advisory file locking over any arbitrary portion of a file's address space, even over network file systems. The described node latch sets are implemented by dividing each node's address space into three independent segments which are given to the operating system to immediately grant, or delay granting, shared or exclusive access to the segment. The details of how these system calls implement node latches are presented in the attached source code.

The question of how to manage the change of key space in the upper tree levels after the deletion of nodes is addressed, with a resulting improvement in symmetry between insert and delete.

New latch modes

Instead of Jaluta's hierarchical "S-U-X" B^{link} Tree latch modes that each cover a node exclusively, one at a time, three independent node latch sets are used for concurrency control using advisory file locking primitives for exclusive or shared access to a sub-segment of each node's address space. These latches are implemented entirely by the underlying Operating System, not by the B^{link} Tree code, based on file system locks over file offsets and lengths. Each node is divided into three segments for this purpose, with the underlying Operating System granting requests for shared or exclusive access to each node latch:

- The first set is a sharable `AccessIntent` and exclusive `NodeDelete` for the node. `AccessIntent` is obtained during `Search` using latch-coupling from the parent node to the child node and never blocks. `NodeDelete` is obtained as the final stage before returning the node the the free list of available nodes and is requested only after all references to the node have been removed from the B^{link} Tree.

- The second set is a sharable `ReadLock` and exclusive `WriteLock` for the node. These latches are obtained during `Search` using latch-coupling with `AccessIntent` and is either `ReadLock` or `WriteLock` depending on the desired access for that level of the tree.

- The third set `ParentModification` exclusive latch is held while a node's upper fence values is posted or removed from its parent node during a `Split` or `Consolidation`. It is obtained with latch-coupling over the node's `WriteLock`. This ensures that structure modifications are serialized with respect to one another, while allowing release of the `WriteLock` on the node for increased concurrency during the upper fence key postings.

Latch Sets

The three sets each cover only a part of each node. Thus it is possible to obtain a `ParentModification` latch over a `WriteLock`, and then release the `WriteLock` later while continuing to hold `ParentModification`. The latches in each set are independent from the latches in the other set on each node. They are not equivalent to traditional "latch modes" because it is possible to hold more than one latch on a node, one from each set simultaneously.

Cost and Option to Mitigate

The cost of guaranteeing that nodes will not be referenced by concurrent tree operations after they are deleted is the requirement to obtain two node latches at each level of the tree after the root access during `Search`, instead of just one. This expense is mitigated by the absense of latch upgrades or downgrades during `Insert` or `Delete`. If some tolerance for temporarily wasted space in the B^{link} Tree file is available, the `AccessIntent` and `NodeDelete` latch steps can be eliminated as well as latch-coupling between child and parent during `Search`. The file space used by deleted nodes can be reclaimed after a suitable delay that allows for the draining away of starved operations that were underway while the node was removed from the B^{link} Tree.

Latch Compatibility Matrix

A "y" (yes) entry means that two latches of the types specified by the row and column labels for that entry can be simultaneously held on a data item by two different B^{link} Tree operations, i.e., the two latch types do not conflict. An "n" (no) entry means that the two latch types cannot be concurrently held on a data item by distinct operations, i.e., the latch types conflict. There is never a case where a node is being deleted twice and this is marked N/A. Note that after `NodeDelete` is requested, no further `AccessIntent` requests are possible since all references to it have been removed from the tree and this is also marked N/A. The table indicates the latch type requested across the X axis, and the existing latch(es) being held down the Y axis.

```
                    Latch Requested

                 AI    ND    RL    WL    PM
E AccessInten t   Y     N     Y     Y     Y
X NodeDelete     N/A   N/A    Y     Y     Y
I ReadLock        Y     Y     Y     N     Y
S WriteLock       Y     Y     N     N     Y
T ParentModification  Y  Y    Y     Y     N
```

Implementation Details

There are five latch types for each node organized into three independent sets:

- (set 1) `AccessIntent`: The latch on the parent node will be released, while the child node `ReadLock` or `WriteLock` will be requested next. This temporary latch will never stall since the `NodeDelete` that it is incompatible with will not be requested until after all

references to the node have been removed from the tree.

- (set 1) `NodeDelete`: The node has been removed from the tree, wait for previously granted `AccessIntent` latches to drain away, then set the `NodeDelete` latch.

- (set 2) `ReadLock`: Read the node. Incompatible with `WriteLock`.

- (set 2) `WriteLock`: Allow modification of the node. Incompatible with `ReadLock` and other `WriteLocks`.

- (set 3) `ParentModification`: Serialize the change of the node's upper fence values in the parent nodes. Incompatible with another `ParentModification`, but compatible with the other four latches on the node.

Tree Operations

`Search`: While descending the B$^{\text{link}}$ Tree from the root to the target node at the requested tree level, a `ReadLock` on a parent is latch-coupled with obtaining `AccessIntent` on the child node, which never blocks since `NodeDelete` is obtained only after all references have been removed from the B$^{\text{link}}$ Tree. `AccessIntent` is latch-coupled with obtaining `ReadLock` on the child node. Requests for key values beyond the node's upper fence value are directed to the node's right sibling, which is processed with the same `AccessIntent` and `ReadLock` and upper fence key check as above. Accessing a `Deleted` node results in a sibling traversal using the link field to the `consolidated` node to its left. At the requested tree level at the end of the `Search`, if Updating/Insert/Deleting, a child node `WriteLock` is obtained instead of a `ReadLock`.

`Update/Insert`: A key value is added or updated to a node under the `WriteLock` obtained during `Search` for the root/branch/leaf node being posted and the `WriteLock` is released. The same `Update/Insert` module is used for root, branch, and leaf nodes. If the new key doesn't fit in the node because it is full, the node is `Split` first and the `Update/Insert` is restarted.

`Split`: A `ParentModification` latch is obtained for the node over the existing `WriteLock.` Note that this stalls if a previous `Split/Consolidation` for this node is still underway. The upper half of the contents are split into a new right sibling node (which is not latched), and the `WriteLock` on the left node is released. The new median fence value for the left node is `Inserted` into the appropriate parent, and the new right sibling node's upper fence key is `Updated` in its appropriate parent node to switch it from the left node to the new right node. Note that different parent nodes for the left and right nodes are possible. The `ParentModification` latch on the left node is then released.

`Delete`: A key is removed from a node under a `WriteLock` obtained during `Search` by setting its key delete bit. The `WriteLock` is released if the node is not empty, otherwise the node is `Consolidated` with its existing right sibling, if any.

`Consolidation`: A `WriteLock` is obtained for the right sibling node, and its contents (including its right sibling link field) replace the empty left node's contents under its existing `WriteLock.` The right sibling node's deleted bit is set, and the right node's link field is set to point to the left node, which will direct subsequent searches to the consolidated node. A `ParentModification` latch is obtained for the consolidated node. Note that this stalls if another `ParentModification` is already underway due to concurrent B$^{\text{link}}$ Tree operations. The `WriteLock` for the left node and the `WriteLock` on the right node are released allowing further updates by concurrent tree operations. The old left node upper fence key is `Deleted` from its parent, and the old right node upper fence key is `Updated` in its parent to point to the left node.

Note that these upper fence keys might be in different parent nodes. The `ParentModification` latch on the left node is released. A `NodeDelete` latch is obtained for the deleted right node which waits for all existing `AccessIntent` latches to drain, and a `WriteLock` is obtained for the right node which waits for all `ReadLock` or `WriteLock` to drain.. There are now no pointers and no other latches for the right node, and it is placed onto a free-list for use in subsequent splits. The `NodeDelete` and `WriteLock` latches are released from the now free node.

The structure modifications to the upper tree levels are serialized by the `ParentModification` latch while the node remains available for subsequent `Searches`, `Update/Inserts`, and `Deletes` while the structure modification proceeds.

Deleting Fence Keys

When deleting a child's upper fence key from a parent which is also the parent's upper fence key, three courses of action are possible. This occurs whenever a `Consolidated` node and its right sibling have different parent nodes.

1) The entire child node delete can be ignored, leaving the empty child node in the B^{link} Tree for later use by `Inserts`. This is the approach taken by Jaluta, Lomet, and others, and leaves the key ranges intact for the parent nodes. It suffers from a lack of symmetry with `Insert`, and shuts down node `Deletes` at the grandparent level which are never emptied. The grandparent's key space partitions are never changed to reflect the current contents of the B^{link} Tree.

2) The change in the parent's upper fence key can be propagated up the B^{link} Tree to the grandparent level. This approach will permanently narrow the key range of the parent node, pushing any and all re-inserted keys under the parent's right sibling. This is also not symmetrical with `Insert`.

3) The deleted key can remain as the parent's upper fence key, while the child node is removed from the tree. This approach leaves the key ranges for parent nodes intact, without freezing the grandparent level. Further `Searches` for keys in the deleted range will be redirected to the parent's right sibling and any subsequent split from `Inserts` there will occur back into the original parent node. The cost to achieve this symmetry is an extra right sibling parent node traversal for key `Searches` in the deleted range. This is the approach taken here.

Latch Coupling

At several points in the implementation, latch coupling is used to acquire a new latch on a node and release another. During `Splits/Consolidations` this overlap is extended to include additional node processing before the previous latch is released so that two latches might be held simultaneously. During `Search`, a parent node will remain fully available during any delay in obtaining a child `ReadLock/WriteLock` over `AccessIntent`. This guarantees that the operations conducted under the `ParentModification` latch remain deadlock free.

Source Code URL

Full implementation in the C language is available at www.code.google.com/p/high-concurrency-btree and consists of 1700 lines of code and comments. It will compile under either unix/linux or WIN32.

Key Size and Node Layout

The node size is specifiable in bits, from 9 (512) thru 20 (1 MB) and stores variable sized keys, each from 1 thru 255 bytes in length. Each node has a tree level value (0 for leaf) and a delete bit which redirects searches to the link (left) node. Each key has a deleted bit which allows keeping the upper fence key value on the page even after it's deleted. The space within the node from deleted keys is reclaimed during node cleanup, which is attempted before deciding to `split` during `insert`.

An array of keys and their values is kept sorted in each node by key value, with the highest array slot reserved for the node's upper key fence value, even if it is deleted.

Each node also has a link field which points to a sibling node. An active node points to its right sibling under guidance of its upper fence key, while a deleted node has its link changed to point to the `Consolidated` node on the left which is used under guidance of the node's delete bit. The link field is updated during `Split` and `Consolidation`.

Buffer Pool Manager

A Buffer Pool Manager is included in the source code that utilizes the underlying Operating System file mapping interface to map B^{link} Tree nodes to program virtual memory. Up to 65536 nodes are buffered (this is a Linux and WIN32 system limitation). The Buffer Manager will also run without a pool or file mapping, using standard file system reads and writes top access nodes. Note that file mapping is not supported over network file systems.

Acknowledgements

References

Jaluta, I., Sippu, S., Soisalon-Soininen, E., 2005: Concurrency control and recovery for balanced B-link trees. VLDB J. 14(2): 257-277.

Lehman, P. L., Yao, S. B., 1981: Efficient locking for concurrent operations on B-trees. ACM TODS 6(4): 650-670.

Lomet, D. B., 2004: Simple, robust and highly concurrent B-trees with node deletion. ICDE: 18-28.