Correctness Proof of Euno-B+Tree

December 1, 2016

To prove that our algorithm is correct, we show that it only generates linearizable executions and acyclic serialization graph. Since deletion is treated as a normal put, and re-balancing is deferred. Here we only prove the linearizability between Get and Put operations (Algorithm 1). Put operation can be transferred to an update or an insertion (perhaps with splits) and we treat them differently here. All operations are comprised of two transactions, denoted as Tx1 (Line 3-8) and Tx2 (Line 22-32). Due to the space limitation, we only provide a proof sketch here.

We start by establishing the linearizability property between Get and Update operations. (1) If a Get operation and an Update operation share the same key. The linearization point is defined at acquiring the lock bit in conflict control module (Line 10), because their index in lock bits vector must be the same. When an operation (denoted as op1) first set its lock bit, the subsequent operation (op2) must be blocked at acquiring the lock bit until op1 releases the lock bit (Line 35). When op2 enters its Tx2 (Line 22), all the modifications made by op1 have already taken effect. That is, op1 precedes op2 in the sequential history. (2) If their keys are different, they could still be synchronized in Line 10 due to hash aliasing. In this case, since they share no common variables, the extra synchronization point could never affect the eventual output of the two operations. The proof of Update-Update and Get/Update-Insert (w/o splits) is similar as above.

The Insert operation is different. Even if it does not share the same key with other operations, it could still influence their search path due to splits. We take Get-Insert (w/ splits) as an example. (1) If the search paths of Get and the split path of Insert do not interlap (i.e., keys must be different), then it is a trivial case. (2) The search paths of Get and split path of Insert interlap at an internal node (then their keys must be different). Although the search path of a Get operation is changed, the final result (i.e., the pointer to the target leaf node) is still the same, thus the Tx2 could still proceed without any need to retry. Therefore, there is no need to define their linearization point. (3) The search paths of Get and split path of Insert interlap at a leaf node. If their keys are different and there is no hash aliasing, then Line 10 cannot be linearization point obviously. In this case, we define the linearization at Line 32. Due to the ACID properties of transaction, if an operation op1 execute Line 32 before op2, all of op1's operations in Tx2 must happen before op2 atomically to an outside observer. That is, op1 precedes op2 in sequential history. If Insert operation executes Line 32 before Get operation

(otherwise it is a trivial case), then the target leaf node of Get is split by Insert. Due to the semantics of B+Tree, the target key of Get operation (denoted as L) could be moved to the new-born sibling of L (denoted as L'). The Get operation cannot find the right key if it proceeds with the pointer attained from Tx1, which still points to L. However, in this case, the seqno of L would have already been update by Insert. The Get operation will find the inconsistency of seqno (Line 23), and this operation will retry from the root node. That is, the Insert operation much precede Get in sequential history, and no false negative result will be returned. If their keys are the same or hash aliasing happens, the linearization would be at Line 10. The proof of Update-Insert and Insert-Insert is similar as above.

In summary, (1) if two operations share the same key, their linearization point is defined at Line 10. (2) If their keys are different, the linearization is defined at Line 32 if hash aliasing does not happen. Otherwise, it is defined at Line 10.

Then we need to show that Euno-B+Tree only generate acyclic serialization graph. In a proof by contradiction we assume that there is a cycle in serialization graph, denoted by $op_i \rightarrow op_{i+1} \rightarrow ... \rightarrow op_j \rightarrow op_i$. Then, there exists an op_k ($k \in i+1 \rightarrow j$) that $op_i \rightarrow op_k \rightarrow opi$. op_k could only synchronize with the former and latter op_i both at Line 10 or Line 32. First, let's consider op_i is linearized with op_k at Line 10, in which case they access the same record. As the lock bit will not be released until op_i finish, op_i can only either be serialized before or after op_k. Then, op_i can only be linearized with op_k at Line 32. For this case, as Line 32 indicates the end of the operation, op_i can only be serialized before op_k or after, but not both. Therefore, a cycle in the serialization graph can not exist, hence our algorithm is always serializable.

Algorithm 1 Get/Put Interface (Two-Step Tree Traversal)

```
1: procedure TRAVERSE(REQ_TYPE, key, newVal)
2: RETRY:
3: XBEGIN() //upper region
4:     node = root
5:     leaf = findLeaf(node, key)
6:     seqno = leaf.seqno
7:     ccm = leaf.CCModule //get the conflict control module
8: XEND()
9:     slot = hash(key)
10:    while !CAS(ccm[slot].lockBit, 0, 1) do
11:        spin()
12:    exist = ccm[slot].marked
13:    if !exist then
14:        if REQ_TYPE == GET then
15:            ccm[slot].lockBit = 0
16:            return null
17:        else
18:            //insert the key if it does not exist when do a put
19:            insert = CAS(ccm[slot].marked, 0, 1)
20:            if insert and leaf.isNearFull() then
21:                leaf.lock() //hold the lock for split
22: XBEGIN() //lower region
23:    if seqno != leaf.seqno then
24:        consistent = false //inconsistency happens
25:    else
26:        if exist then
27:            record = leaf.getRecord(key)
28:        if REQ_TYPE == PUT then
29:            if record == null then
30:                record = INSERT(leaf, key)
31:            record.value = newVal
32: XEND()
33:    if leaf.isLocked() then
34:        leaf.unlock()
35:    ccm[slot].lockBit = 0
36:    if !consistent then
37:        goto RETRY
```