Decoupling indexing from correctness for improved concurrency (and other good things)

15-712 Fall 2007

Efficient Locking for Concurrent Operations on B-Trees.

Lehman81: Philip Lehman, S. Bing Yao. ACM Trans. on Database Systems (TODS), vol 6, no 4, December 1981.

Guest Appearance

- The Chord Distributed Hash Table (DHT)
  - Ion Stoica, Robert Morris, David Karger, M. Frans Kaashoek, and Hari Balakrishnan, *Chord: A Scalable Peer-to-peer Lookup Service for Internet Applications*, ACM SIGCOMM 2001, San Deigo, CA, August 2001, pp. 149-160.

B-tree Index Access

- B-Trees common concurrent data structure
  - Indices for databases of all kinds
  - Good at fixed, short depth of tree for fast lookup
    - Based on all nodes having a min and max number of keys, and splitting or redistributing keys to nodes (rebalancing)
  - Insertion & deletion can change many nodes
    - if the tree becomes unbalanced, parent nodes must be updated, which can split or merge them, continuing up to the root

Eg. Splitting a B-Tree

- Add item with key 47
  - Node for 47 is full
  - Split node into two
  - Add entry in parent
- If parent is full
  - Propogate split up
- Delete is messier still
  - To preserve constant depth, may need to rotate keys from a sibling or compress a whole level

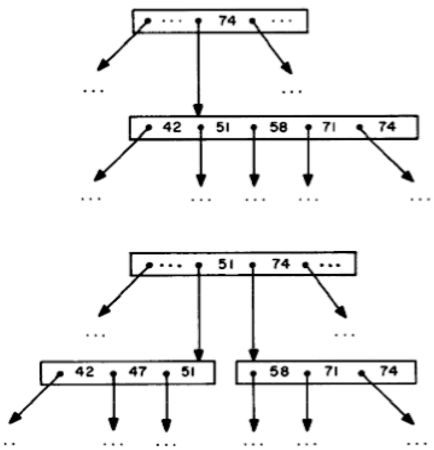

Fig. 4. Splitting a node after adding "47" ($k = 2$).

Concurrent Access Problems

- Indices are shared data structures with high concurrency
  - specialize concurrency control

	search(15)	insert(9)
1.	$C \leftarrow$ read(x)	
2.		$A \leftarrow$ read(x)
3.	examine C; get ptr to y	
4.		examine A; get ptr to y
5.		$A \leftarrow$ read(y)
6.		insert 9 into A; must split into A, B
7.		put(B, y')
8.		put(A, y)
9.		Add to node x a pointer to node y'.
10.	$C \leftarrow$ read(y)	
11.	*error*: 15 not found!	

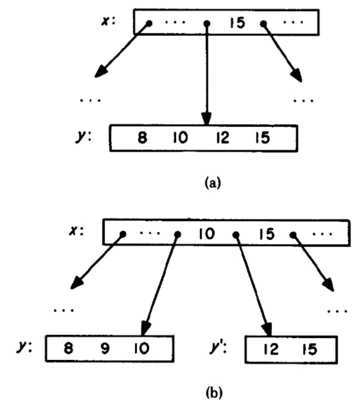

Fig. 5. Counterexample to naive approach.

Lock-based Concurrency Control

- Used by most databases today for all applic code
  - Inside any transaction, all accessed data is protected by read/write locks & stored in shadow pages or undo logs (later lecture) until changes are committed & written
  - All locks acquired are held until transaction is done (!)
  - So concurrent transactions sharing any page are serialized by page locks, that is, with respect to shared pages, execute one at a time
  - Beware deadlocks -- if locks cannot be hierarchicalized, then detect lock cycles and break with abort & rollback

Concurrent B-tree Access

- Simple: lock all nodes that might change as you look for point of insertion
  - But this locks top of tree, blocking everything else
- Bayer77: don't write lock top of tree on first try; hope splitting will not need to change top of tree
  - If wrong, abort and retry holding all write locks

B-Link-Tree Example

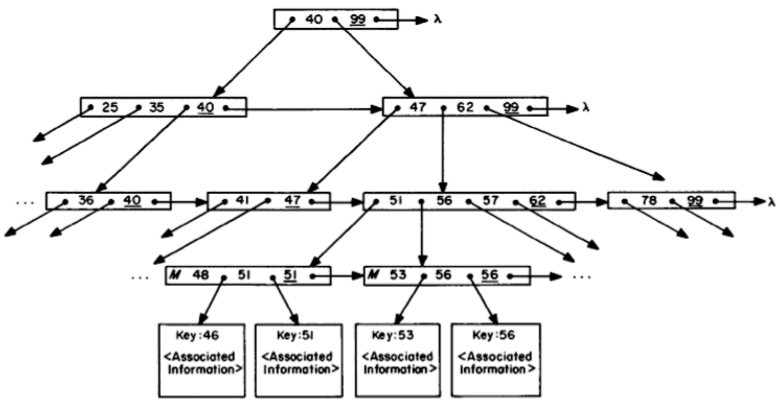

Fig. 7. A B^{link}-tree.

Lehman81: more concurrency

- Small maximum number of locks held (3)
- Readers are never blocked; tree is always valid
  - Reader may "miss" concurrent insertion
- B-link-tree adds same-level next-node link
  - New pointer points to node at same level with next key
  - Reader searching a node being split may see new "highest value" smaller than search key, & go on to next node
  - Effectively hides splitting from concurrent readers until the atomic update of the parent pointer
  - Writers only lock an individual node wrt other writers
  - Careful coding of split/rebalance needed
  - Useful for fast range scans too
- "Disk" ops get(), put() are indivisable (atomic)

Readers never lock!

- What atomic ops make not locking work?

$x \leftarrow scannode(v, A)$ denotes the operation of examining the tree node in memory block A for value v and returning the appropriate pointer from A (into x).

```
procedure search(v)
current ← root;                                  /* Get ptr to root node */
A ← get(current);                                /* Read node into memory */
while current is not a leaf do
begin                                            /* Scan through tree */
   current ← scannode(v, A);                     /* Find correct (maybe link) ptr */
   A ← get(current)                              /* Read node into memory */
end;
                                                 /* Now we have reached leaves. */
while t ← scannode(v, A) = link ptr of A do
                                                 /* Keep moving right if necessary */
begin
   current ← t;
   A ← get(current)                              /* Get node */
end;
                          /* Now we have the leaf node in which v should exist. */
if v is in A then done "success" else done "failure"
```

Reading re-written

```
node n = root
while (n.type != TYPE_LEAF) {
    n = n.scan_for(key)
} /* log(n) */

while ((c = n.scan_for(key)) == n.link) {
    n = c
} /* linked list traversal */

if (n.contains(key)) return n.lookup(key)
return NULL
```

Insertion

- Inserted value appears at step c, although f still not updated
- Is tree always balanced?
  - no? so why is approach used?
  - trade weakening of maximal comparisons for more concurrency

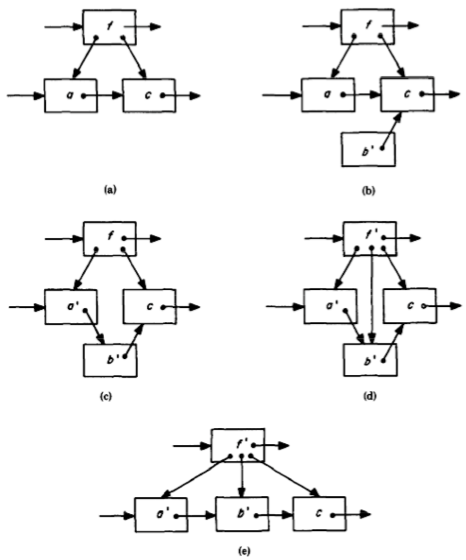

Fig. 8. Splitting node *a* into nodes *a'* and *b'*. (Note that (d) and (e) show identical structures.)

Insertion

- Nasty code!
- Livelock possible
- Eraser would ….

```
procedure insert(v)
initialize stack;                                   /* For remembering ancestors */
current ← root;
A ← get(current);
while current is not a leaf do
begin                                               /* Scan down tree */
  t ← current;
  current ← scannode(v, A);
  if new current was not link pointer in A then
    push(t);                                        /* Remember node at that level */
  A ← get(current)
end;
lock(current);                                      /* We have a candidate leaf */
A ← get(current);
move.right;                                         /* If necessary */
if v is in A then stop "v already exists in tree";  /* And t points to its record */
w ← pointer to pages allocated for record associated with v;
Doinsertion:
if A is safe then
begin
  A ← node.insert(A, w, v);                         /* Exact manner depends if current is a leaf */
  put(A, current);
  unlock(current);                                  /* Success—done backtracking */
end else begin                                      /* Must split node */
  u ← allocate(1 new page for B);
  A, B ← rearrange old A, adding v and w, to make 2 nodes,
    where (link ptr of A, link ptr of B) ← (u, link ptr of old A);
  y ← max value stored in new A;                    /* For insertion into parent */
  put(B, u);                                        /* Insert B before A */
  put(A, current);                                  /* Instantaneous change of 2 nodes */
  oldnode ← current;                                /* Now insert pointer in parent */
  v ← y;
  w ← u;
  current ← pop(stack);                             /* Backtrack */
  lock(current);                                    /* Well ordered */
  A ← get(current);
  move.right;                                       /* If necessary */
  unlock(oldnode);
  goto Doinsertion                                  /* And repeat procedure for parent */
end

procedure move.right
while t ← scannode(v, A) is a link pointer of A do
begin                                               /* Move right if necessary */
  lock(t);                                          /* Note left-to-right locking */
  unlock(current);
  current ← t;
A ← get(current);
end
```

Deletion

- "allow fewer than k entries in a leaf node"
  - far simpler than one that requires underflows ad concatenations
  - uses a "little" extra storage (and comparisons)
  - if needed defragment with batch reoganization locking whole tree
    - Possibly a cop-out: Slower worst-case
    - But may avoid unnecessary merges w/later insert
- it would be better if there was a GFS-like background process renormalizing the B-link tree

Eval

- Proof-based, part of theoretical DB work
  - Makes assumptions about primitives, OS functions and invariants (so proof must be tested :-)
- Some subsequent use of these methods
  - Boxwood at MSR
- Fragile code!
  - Not a database transaction ….
- Gives up balance properties for unknown duration
  - Worst case analysis gets worse, normal case better

Context & Comparison

- Not transaction locking
  - May still need to ensure consistency btwn multiple read/writes (Kung does this; Lehman does not!)
- Thoughts?

Technique

- Correctness depends on link pointers
  - Some operations could (very rarely!) degrade to a linked-list traversal
- Similar in "feel" to some DHT techniques, like Chord
  - Consistent Hashing
  - Node IDs = hash(node IP), mapped in circular 128-bit (or whatever) key space
  - Items "belong" to successor node

Operation

- Node insertion:
  - Search for yourself in the ring
    - succ = ring_search(me)
  - Update your predecessor
    - me.next = succ
    - pred.next = succ.prev
  - Quick insertion == *correctness*, but

Optimizing searches

- Finger table
  - Points 1/2, 1/4, 1/8th of way around the ring
    - How to find? Search for items in those spaces!
- Finger table ("index!") correctness not critical for integrity
  - Can always fall back to linear search
  - But provides eventual efficiency

Points

- Decoupled optimization and correctness good for distributed implementation
  - Sagiv's B*-link tree variant
  - Chord's "finger table"
  - Skip-list based approaches