On Optimistic Methods for Concurrency Control

Phil Gibbons

15-712 F15

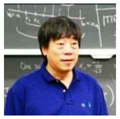
Lecture 19

Today's Reminders

- **Pick up graded midterm from Kevin, if you haven't yet**
- **Just received transcript of feedback**
  - Will reflect further and get back to you

On Optimistic Methods for Concurrency Control
[TODS 1981]

- **H.T. Kung** (Harvard)
  - NAE, CMU PhD/Prof, Guggenheim Fellow
  - www.eecs.harvard.edu/htk/phdadvice/



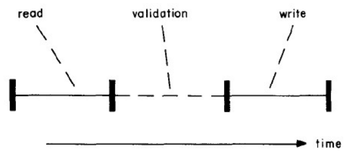
- **John T. Robinson** (IBM until 2005)
  - CMU PhD, IBM "master inventor"
  - "An interesting problem is one where it is not known in advance how (or even if) the problem can be solved…I love working on interesting problems."

What's Wrong with Locks?

- **Locks are overhead vs. sequential case**
  - Even for read-only transactions; Deadlock detection
- **No general-purpose deadlock-free locking protocols that always provide high concurrency**
- **Paging leads to long lock hold times**
- **Locks cannot be released until end of transaction (to allow for transaction abort)**
- **Locking may be necessary only in the worst case**

- **Priority inversion**
- **Lock-based programs do not compose: correct fragments may fail when combined**

Three Phases of a Transaction



Read Phase

create	create a new object and return its name.
delete(n)	delete object n.
read(n, i)	read item i of object n and return its value.
write (n, i, v)	write v as item i of object n.
copy(n)	create a new object that is a copy of object n and return its name.
exchange($n1$, $n2$)	exchange the names of objects $n1$ and $n2$.

```
tcreate = (
  n := create;
  create set := create set ∪ {n};
  return n)

twrite(n, i, v) = (
  if n ∈ create set
     then write(n, i, v)
  else if n ∈ write set
     then write(copies[n], i, v)
  else (
     m := copy(n);
     copies[n] := m;
     write set := write set ∪ {n};
     write(copies[n], i, v)))
```


Read Phase

create	create a new object and return its name.
delete(n)	delete object n.
read(n, i)	read item i of object n and return its value.
write (n, i, v)	write v as item i of object n.
copy(n)	create a new object that is a copy of object n and return its name.
exchange($n1$, $n2$)	exchange the names of objects $n1$ and $n2$.

```
tread(n, i) = (
  read set := read set ∪ {n};
  if n ∈ write set
     then return read(copies[n], i)
  else
     return read(n, i))
```

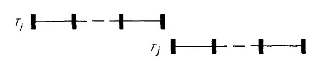
Write Phase

for $n \in$ *write set* **do** *exchange*(n, *copies*[n]).

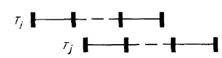
Validation Phase

- **Assign transaction number at the end of the read phase**
- **Serial equivalence:**

1)

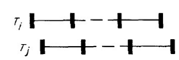


2)



$\boldsymbol{WriteSet_i}$ does not intersect $\boldsymbol{ReadSet_j}$

3)



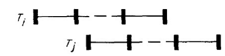
$\boldsymbol{WriteSet_i}$ does not intersect $\boldsymbol{ReadSet_j}$ or $\boldsymbol{WriteSet_j}$

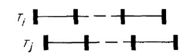
Practical Considerations

- **Write Sets are not infinite**
- **Transactions can starve**
- **If serializing write phases has acceptable performance, then validation is straightforward**
  - Place assignment of transaction number, validation, subsequent write phase all in a critical section

Parallel Validation

```
tend = (
  ⟨finish tn := tnc;
   finish active := (make a copy of active);
   active := active ∪ {id of this transaction}⟩;
   valid := true;
   for t from start tn + 1 to finish tn do
       if (write set of transaction with transaction number t intersects read set)
          then valid := false;
   for i ∈ finish active do
       if (write set of transaction T_i intersects read set or write set)
          then valid := false;
   if valid
       then (
          (write phase);
          ⟨tnc := tnc + 1;
           tn := tnc;
           active := active—{id of this transaction}⟩;
          (cleanup))
       else (
          ⟨active := active—{id of transaction}⟩;
          (backup))).
```





Use of OCC for Concurrent Insertions in B-Trees

- **Read/Write sets bounded by depth of tree, which is small**
- **Due to page faults in Reads, Validation+Write time incurs minimal overhead versus Read time**
- **One (random) insertion unlikely to cause another insertion to fail its validation**

Thoughts on this argument?

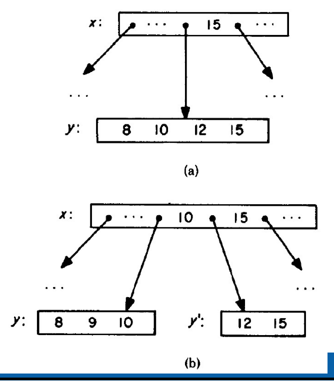
Locks are Bad for B-Trees?

- **Locks are overhead vs. sequential case**
  - Even for read-only transactions; Deadlock detection
- **No general-purpose deadlock-free locking protocols that always provide high concurrency**
- **Paging leads to long lock hold times**
- **Locks cannot be released until end of transaction (to allow for transaction abort)**
- **Locking may be necessary only in the worst case**
- **Priority inversion**
- **Lock-based programs do not compose**

Which arguments hold?

Efficient Locking for Concurrent Operations on B-Trees

[Philip Lehman & Bing Yao, TODS 1981]



Problem Scenario

Thread 1: search for 15
Reads x, gets ptr to y

Thread 2: insert 9
Reads x; Splits y: inserts 9, adds ptr in x to y'
[see Fig (b)]

Thread 1:
Reads y; 15 not found!

Splitting in B-link Tree

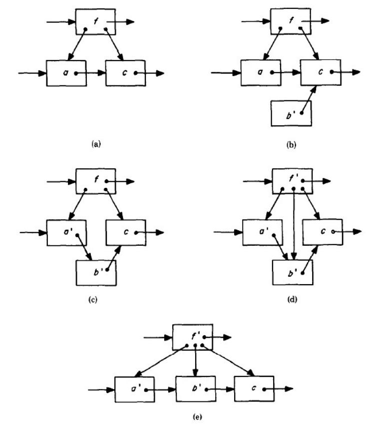


Fig. 8. Splitting node a into nodes a' and b'. (Note that (d) and (e) show identical structures.)

Insert
Keep track of rightmost node visited at each level

Lock a node before modifying it

Corner case requires 3 locks

Search
Follows link ptrs as needed

No locking!

Use of OCC for Concurrent Insertions in B-link-Trees

How does B-link-tree change this argument?

- **Read/Write sets bounded by depth of tree, which is small**
- **Due to page faults in Reads, Validation+Write time incurs minimal overhead versus Read time**
- **One (random) insertion unlikely to cause another insertion to fail its validation**

Locks are Bad for B-link-Trees?

- X **Locks are overhead vs. sequential case**
  - Even for read-only transactions; Deadlock detection
- n/a **No general-purpose deadlock-free locking protocols that always provide high concurrency**
- X **Paging leads to long lock hold times**
- X **Locks cannot be released until end of transaction (to allow for transaction abort)**
- X **Locking may be necessary only in the worst case**
- X **Priority inversion?**
- n/a **Lock programs do not compose**

While OCC is good, example is bad

Locks, OCC, etc Today

- **Fine-grained locking still challenging to get right**
- **Software Transactional Memory (STM)**
  - Some Hardware Transactional Memory [Haswell, 2013]
- **Hardware Lock Elision (HLE)**
- **Heavy use of Multiversion Concurrency Control**

Wednesday's Class

Concurrency Control and Recovery

Mike Franklin

[Computer Science & Engineering Handbook 1997]