15-712:
Advanced Operating Systems & Distributed Systems

On Optimistic Methods for Concurrency Control

Prof. Phillip Gibbons

Spring 2023, Lecture 11

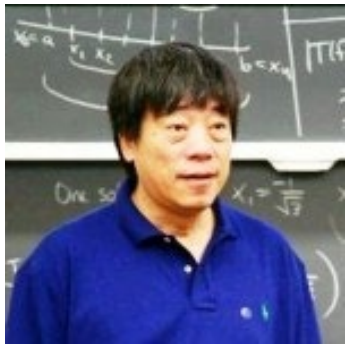
Today's Reminders / Announcements

- **Midterm #1 (in class) on Monday 2/20**
  - Covers the 12 papers you read: Lampson83, Saltzer84*, Birrell84, Lamport78, Chandy85, Li19, McKusick84, Howard88, Rosenblum92, Patterson88, Kung81, Franklin97
    (* = read but did not write a summary)
  - May be helpful to review lecture slides
  - Only need to answer 7 of 9 questions
  - 80 minutes. No notes or papers
  - Practice midterms are on course webpage

"On Optimistic Methods for Concurrency Control"

H. T. Kung, John T. Robinson 1981

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



- **John T. Robinson** (CMU PhD, IBM until 2005)
  - 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."

“On Optimistic Methods for Concurrency Control”

H. T. Kung, John T. Robinson 1981

SigOps HoF citation (2015):

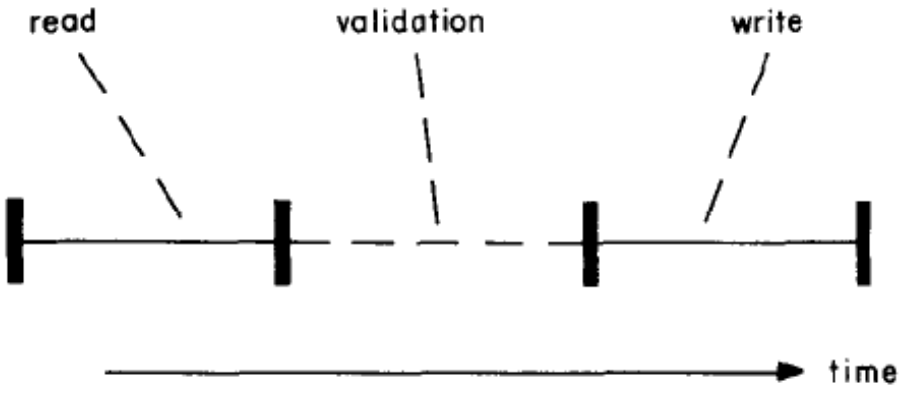
This paper introduced the notion of optimistic concurrency control, proceeding with a transaction without locking the data items accessed, in the expectation that the transaction’s accesses will not conflict with those of other transactions. This idea, originally introduced in the context of conventional databases, has proven very powerful when transactions are applied to general-purpose systems.

What's Wrong with Locks?

- **Locks are overhead vs. sequential case**
  - Even for read-only transactions; Need 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



During read phase: Any write must be to a local copy

Write phase occurs iff validation succeeds

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 $n2$.

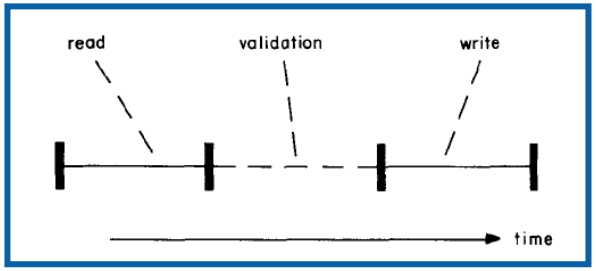
```
tread(n, i) = (
   read set := read set ∪ {n};
   if n ∈ write set
        then return read(copies[n], i)

   else
        return read(n, i))
```

Write Phase

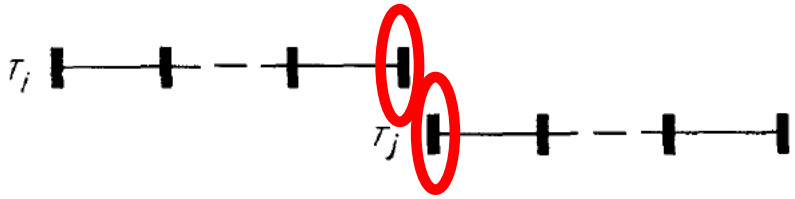
for $n \in$ *write set* **do** *exchange*$(n,$ *copies*$[n])$.

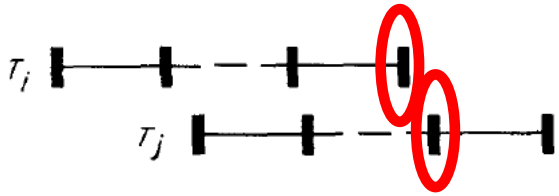
Validation Phase

- **Assign transaction number at the end of the read phase**
  - There are optimizations that assign it later

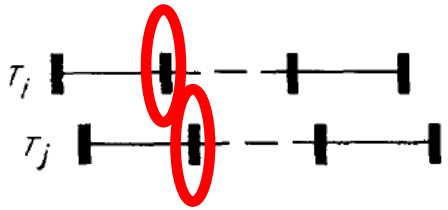


- **Serial equivalence: T_i before T_j**

1) 

2) 

$+ \boldsymbol{WriteSet_i}$ doesn't intersect $\boldsymbol{ReadSet_j}$

3) 

$+ \boldsymbol{WriteSet_i}$ doesn't intersect $\boldsymbol{ReadSet_j}$
nor $\boldsymbol{WriteSet_j}$

Practical Considerations

- **Can only maintain finitely many Write Sets**
  - Validation fails if old write set no longer available
- **Transactions can starve**
  - Restart without releasing the critical section semaphore
- **Serial Validation (condition 3 disallowed): If this has acceptable performance, then validation is straightforward**
  - Place assignment of transaction number, validation, subsequent write phase all in a critical section
- **At end of read phase, can read global transaction counter & eagerly validate against write sets (start_tn, mid_tn]**
  - Outside of critical section

Discussion: Summary Question #1

- **State the 3 most important things the paper says.** These could be some combination of their motivations, observations, interesting parts of the design, or clever parts of their implementation.

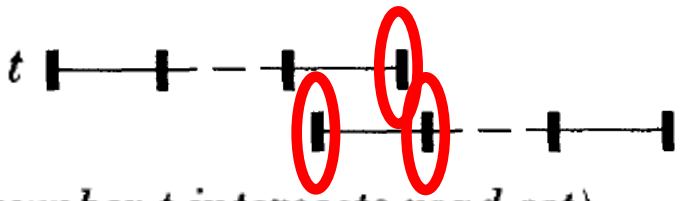
Read-Only Transactions

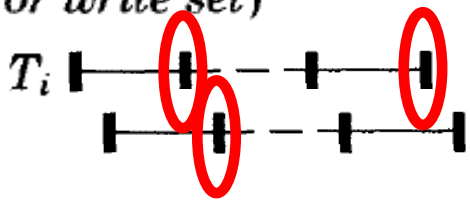
- **At end of read phase,** *finish_tn := tnc*
- **Validate against write sets for transactions in (***start_tn,finish_tn***]**
  - Can be done outside of any critical section
  - Especially fast when *start_tn=finish_tn*

Parallel Validation

Finished read phase, but not write phase

```
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))).
```





Discussion: Summary Question #2

- **Describe the paper's single most glaring deficiency.** Every paper has some fault. Perhaps an experiment was poorly designed or the main idea had a narrow scope or applicability.

Aside: Locks, OCC, etc. Today

- **Fine-grained locking still challenging to get right**
- **Software Transactional Memory (STM)**
- **Hardware Transactional Memory (HTM)**
- **Hardware Lock Elision (HLE)**
- **Heavy use of Multiversion Concurrency Control**
  - Next lecture (optional reading)

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?

Discussion: Summary Question #3

- **Describe what conclusion you draw from the paper as to how to build systems in the future.** Most of the assigned papers are significant to the systems community and have had some lasting impact on the area.

Wednesday's Paper

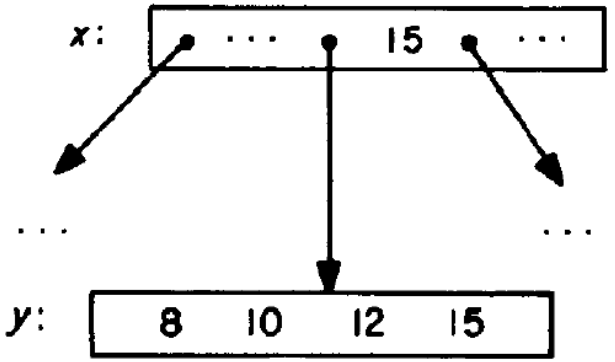
Transactions and Databases (II)

"Concurrency Control and Recovery"

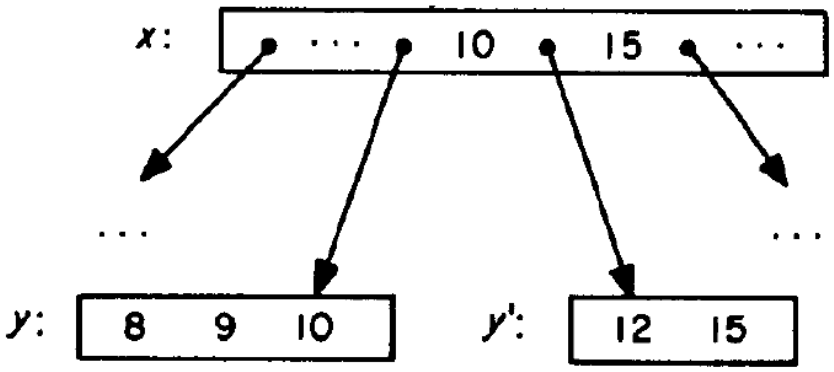
Michael J. Franklin 1997

Aside: Efficient Locking for Concurrent Operations on B-Trees

[Philip Lehman & Bing Yao, TODS 1981]



(a)



(b)

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!

Aside: Splitting in B-link Tree

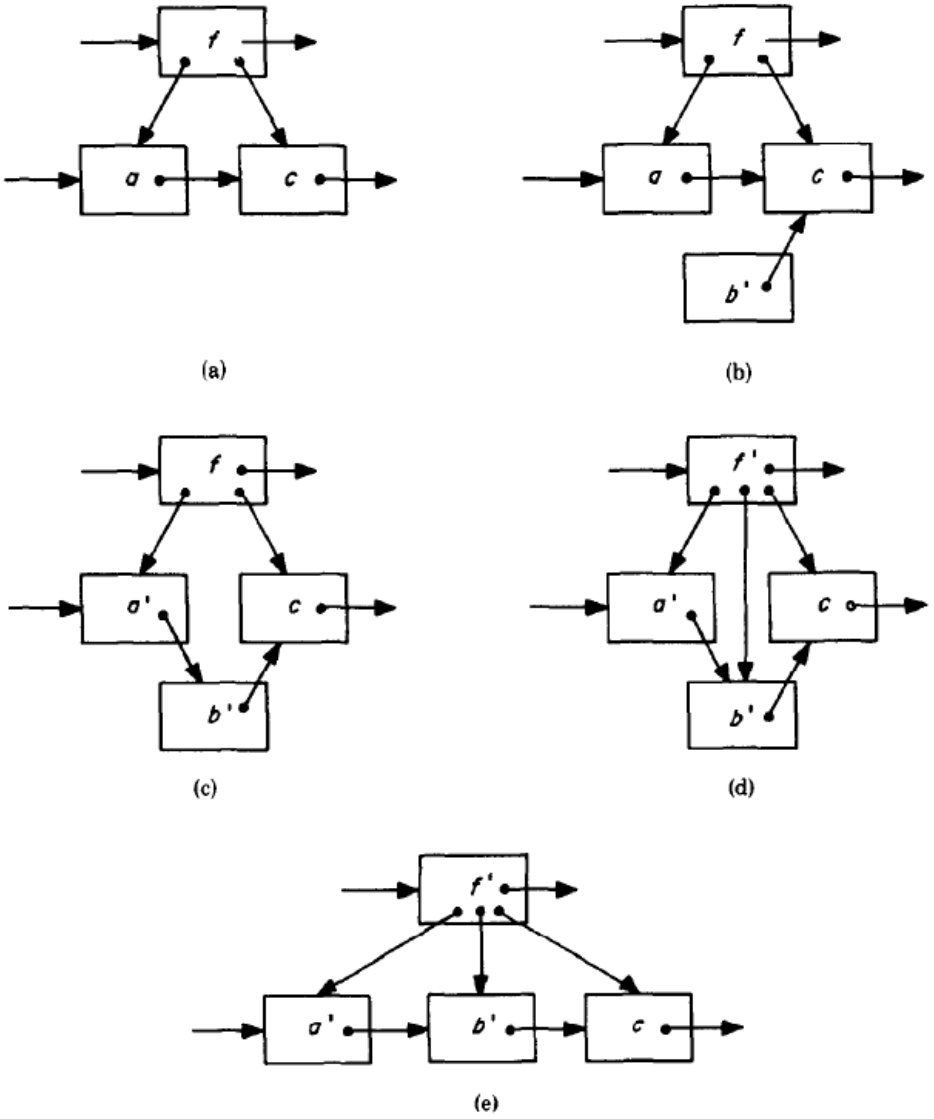


Fig. 8. Splitting node a into nodes a' and b'. (Note that (d) and (e) show identical structures.)

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!

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)

true, but… • Locking may be necessary only in the worst case

X • Priority inversion

n/a • Lock-based programs do not compose

While OCC is good, example is bad