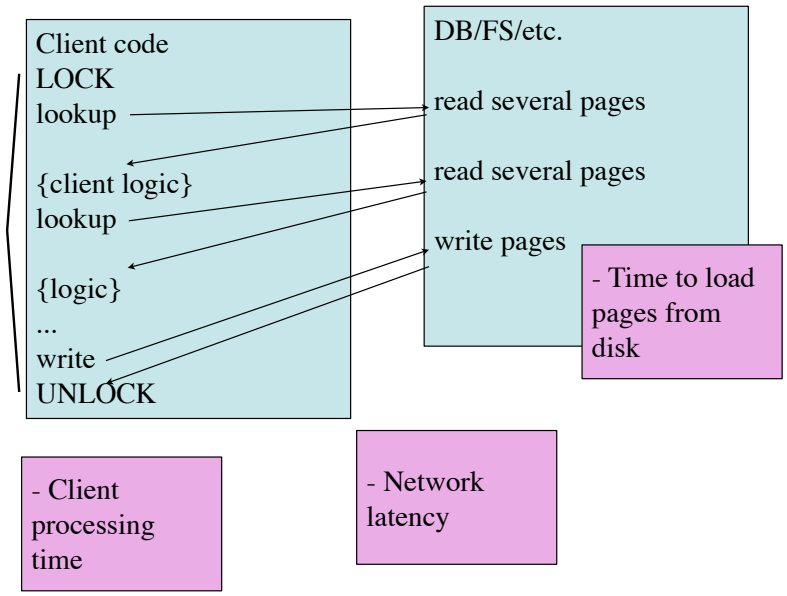
# On Optimistic Methods for Concurrency Control.

**Kung81:** H.T. Kung, John Robinson. ACM Transactions on Database Systems (TODS), vol 6, no 2, June 1981.

# Birth of Optimistic Methods

- Lovely, complex, very concurrent transactions
- Spawned much subsequent systems theory
- Basic tradeoff between go slow & safe vs go fast and clean up after yourself
- Driven by database sensibility:
  - Single threaded access to huge database means blocking all work waiting for disk pages to load
  - Instead, lots of operations in database, some waiting for disks while others work, but protect DB integrity (which is application specific, assumed correct for each transaction running serially)

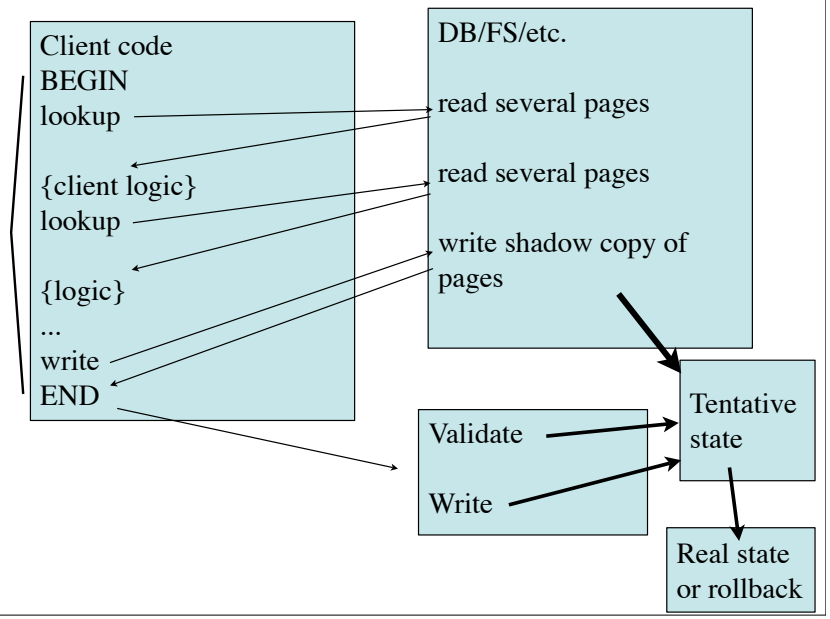



# Basic performance arg

- Locking slows down common path
  - Overhead of locking, reduced concurrency because locks too big, and, the big problem, held too long
- But given millions of different data records, probability of conflict for 2 changes is tiny
  - Cheaper to "hope for the best" but
  - check for conflict near the end of your work and abort/ clean up if real conflict occurred
- Assumes disk & memory read/write atomic
  - Real world CPUs (Eraser & Alpha) may relax this
- In virtual time, database is (passive) process
  - reads & writes are messages from and to it

# Basic Implementation

- Organize transactions to work on private copies
- At end of transaction, "validate" correctness of private copy changes
- If valid, then make copies permanent
- Divides each transaction into:
  - DB read phase (making copies), validation phase & write (copyback) phase
- Doesn't lock DB during client logic computation
- Groups reads & writes of same blocks close in time
- "Pre-fetches" blocks for validation/write phase




# Validation: Serializability

- Separately test all transactions consistent
  - Running alone on database, transaction assumed correct
- Make transaction codes independent of each other
  - Happens-before only property of (data) values in db
- All serial orderings of "concurrent" transactions are valid
- Allow concurrent running if transformations are same as transforms possible from a serial order
  - Serializable

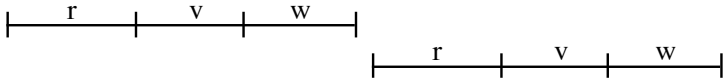
# Basic approach: transaction IDs

- Serializability means a serial ordering exists
- Select ID for each transaction a priori
  - Force DB updates to be equivalent to serial execution in numeric order of ID values
  - In validation phase, abort & retry transactions that would not meet this condition
- Transforms lock queueing slowdowns for the risk of not making progress
  - ID selected at "begin" less concurrent if some read phases run much longer
  - Delay ID choice until validation or later to reduce aborts

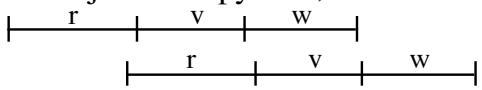
# Conditions for validity

- For i<j, Transaction Ti must "precede" Tj
- 1) Ti ends copyback before Tj reads start, or
  - Actually serial ordering
- 2) Tj reads nothing Ti writes and Ti ends copyback before Tj starts copyback, or
  - Overlap of Tj reading with Ti copyback is harmless & Tj copyback is serialized after Ti copyback
- 3) Tj reads & writes nothing Ti writes and Ti ends reading before Tj ends reading
  - Overlap of Ti copyback with Tj reads because Ti can't hurt Tj, but Tj might hurt Ti if its writes start too soon

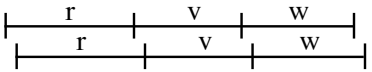
- 1) Ti ends copyback before Tj reads start




- 2) Tj reads nothing Ti writes and Ti ends copyback before Tj starts copyback, or




- 3) Tj reads & writes nothing Ti writes and Ti ends reading before Tj ends reading




# Simple testing of sets

- Long critical section if copyback slow

```
tbegin = (
   create set := empty;
   read set := empty;
   write set := empty;
   delete set := empty;
   start tn := tnc)

tend = (                                              Lock
   (finish tn := tnc;
    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;
    if valid
         then ((write phase); tnc := tnc + 1; tn := tnc));
    if valid                                          Unlock
         then (cleanup)
         else (backup)).
```

# More parallelism

- Test clearly preceding (no longer changing) transactions outside critical section

```
tend := (
   mid tn := tnc;
   valid := true;
   for t from start tn + 1 to mid tn do
   if (write set of transaction with transaction number t intersects read set)
         then valid := false;
   (finish tn := tnc;
   for t from mid tn + 1 to finish tn do
         if (write set of transaction with transaction number t intersects read set)
            then valid := false;
   if valid
         then ((write phase); tnc := tnc + 1; tn := tnc));
   if valid
         then (cleanup)
         else (backup)).
```

# Even more parallelism

- Add ordered testing of condition 3

```
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 Ti 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))).
```

# Space issues

- What if run out of space for sets?
  - Abort & retry
- What about repeated abort & retry?
  - Hold critical section in a retry (ugh)

# Applic to B-trees

- Models B-trees with lots of entries in each page (199), uniform key insertion, interior nodes cacheable and leaf pages not cacheable
- Does consider splitting a leaf, but apparently not rotating tree to maintain balance (not needed if inserts are uniform :-)
- Concludes for such B-trees that conflict, abort and restart will be rare (0.07%)

# Eval

- Thought experiment
  - Not fair as really a database theory paper then
- Analysis of B-tree not appropriate
  - Really should have modeled rotations as “randomness” is far too unlikely
- Very influential: “optimistic methods” is current label of anything trying first then checking if it had a conflict and undoing it

# Lock-based Concurrency Control

- Used by most databases today
  - Inside any transaction, all accessed data is protected by read/write locks & stored in shadow pages or undo logs (later lecture) until changes are commited & 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



```
Client code
BEGIN
lookup

{client logic}
lookup

{logic}
...
write
END
```

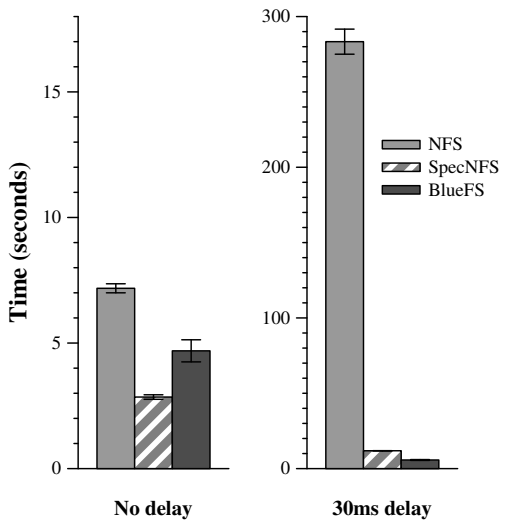
- If no conflicting writes, longer Tx better
- But possibly harder for app to rollback
- What about non-transactional apps?



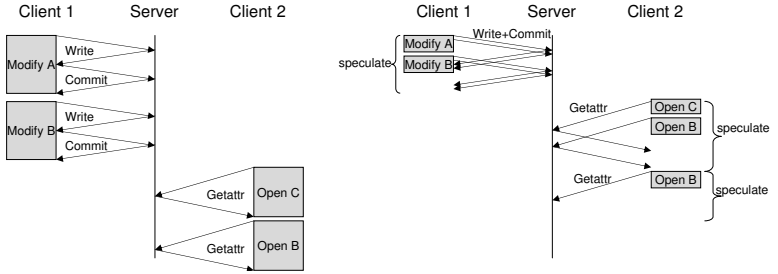
# Speculative Execution

- Example: Unmodified apps using an NFS server
  - Clients cache data, but
  - For consistency, NFS ops are all sync
- Client-server latency greatly slows things down






From Nightingale, SOSP 2005




(a) Unmodified NFS (b) Speculative NFS

From Nightingale, SOSP 2005



# How?

- create_speculation
  - Normal client code goes here
- commit_speculation || fail_speculation
- create_speculation:
  - Create copy-on-write fork of current process, save it away (don't run it)
  - On commit_speculation: Delete copy
  - On fail: Replace original with saved copy, returning "speculation failed"



# Simple idea, but...

- Lots and lots of details
  - What if process does something that would affect system/user visible state?
    - For any op, by default, mark as "wait until speculative ops get resolved"
    - Optimize the common ones by letting them speculate
  - what if process writes while executing speculatively?
    - Makes shadow copy of file (if possible), etc.
  - Propagate speculative bits across fork(), writes to other processes on pipes, etc.
  - If anything too hard (sysV shm), just punt and wait for speculation to end



# Optimizing NFS

- Propagate speculative writes across protocol
  - Make NFS server transactional -- much as in Kung81
  - NFS server keeps shadow copy and can invalidate/write it on commit
- In common case, little sharing in {NFS, AFS, etc.}
- Prior approaches (e.g., Coda) use optimism to allow operations, but punt conflict resolution to {user, app}
  - Designed for longer-term speculation (hours)
  - Nice to make it transparent