On Optimistic Methods for Concurrency Control

By
H.T. KUNG and JOHN T. ROBINSON

Presented by
Lakshmi Narayana Royyala

Why another concurrency control method

- Lock management overhead.
  - Pay high price even when no conflict occurs.
  - Even read-only actions must acquire locks.
  - Locks cannot be released until the end of the transaction
  - High overhead forces careful choices about lock granularity.
- Low concurrency
  - If locks are too coarse or congested nodes
  - Aborts makes it even worse.
- Deadlocks
  - No general-purpose deadlock-free locking protocols
- Low availability
  - A client cannot make progress if the server or lock holder is temporarily unreachable.

Idea behind **Optimistic Approach**

- ”Optimistic”, because in most applications, the likelihood of two transactions accessing the same object at the same time is low

- This approach “hopes” that conflicts do not frequently occur and most transactions are allowed to proceed as if there was no conflict

- Objective is to minimize the time over which a given resource would be unavailable for use by other transactions

- A concurrency control scheme is considered pessimistic when it locks a given resource early in the data-access transaction and does not release it until the transaction is closed

Targets

- If the goal is to maximize throughput of accesses, then there are at least two cases where highly concurrent access is desirable:
  - 1) The amount of data is sufficiently great that at any given time only a fraction of the database can be present in memory
  - 2) Even if the entire database can be present in memory, there may be multiple processors

Optimistic Approach

- Unrestricted access to reads; restricted writes

- Transactions consist of three phases:
  - **Read Phase**: All writes take place on local copies of the object to be modified
  - **Validation Phase**: Determines if transaction causes a loss of integrity
  - **Write Phase**: Copies are made global if validation phase succeeds

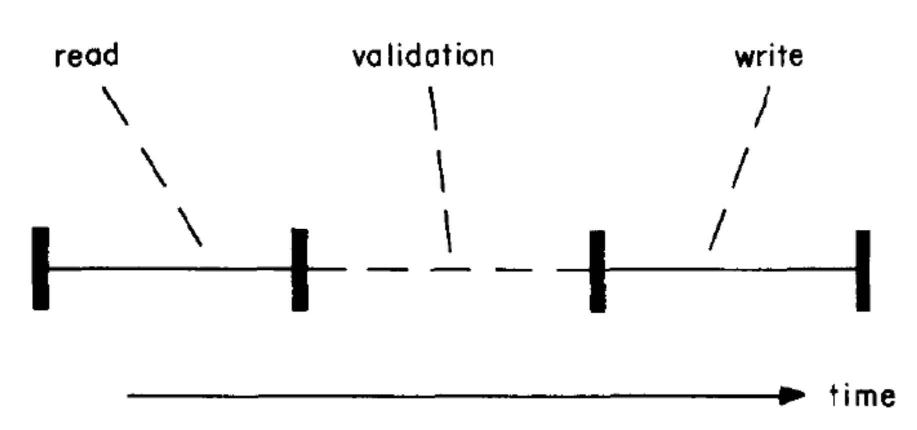
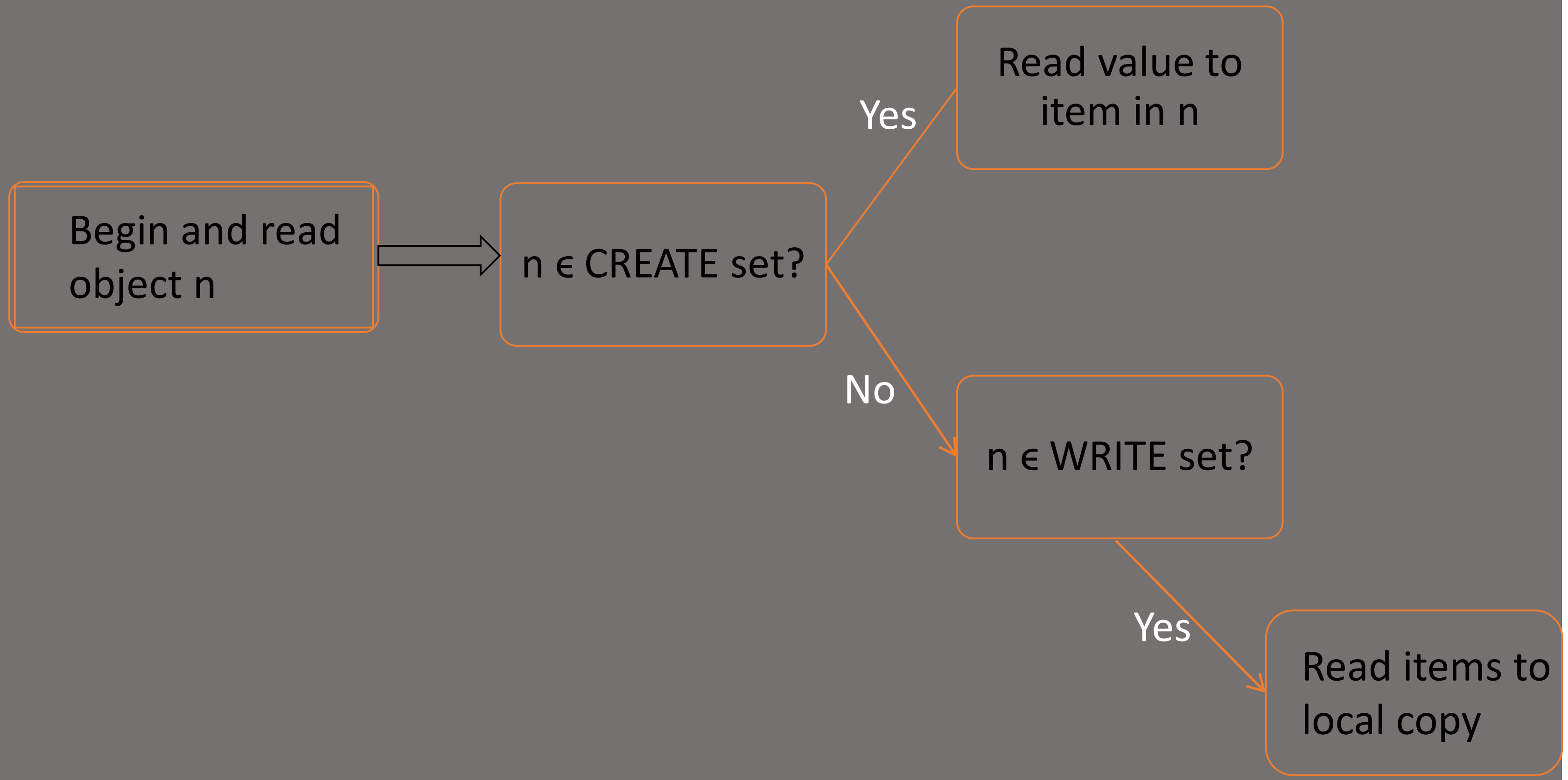


Fig. 1. The three phases of a transaction.

Read and Write Phase

- Transactions use syntactically identical procedures tcreate, tdelete, tread, and twrite. Initialized by tbegin; validation phase begins after tend call
- Each transaction has a tentative version of each of the object that it updates
- READ operations are performed immediately
- If validation succeeds, then the transaction enters the *write* phase
- After *write* phase, all written values become “global”

Read phase



Begin and read object n

n ϵ CREATE set?

Yes

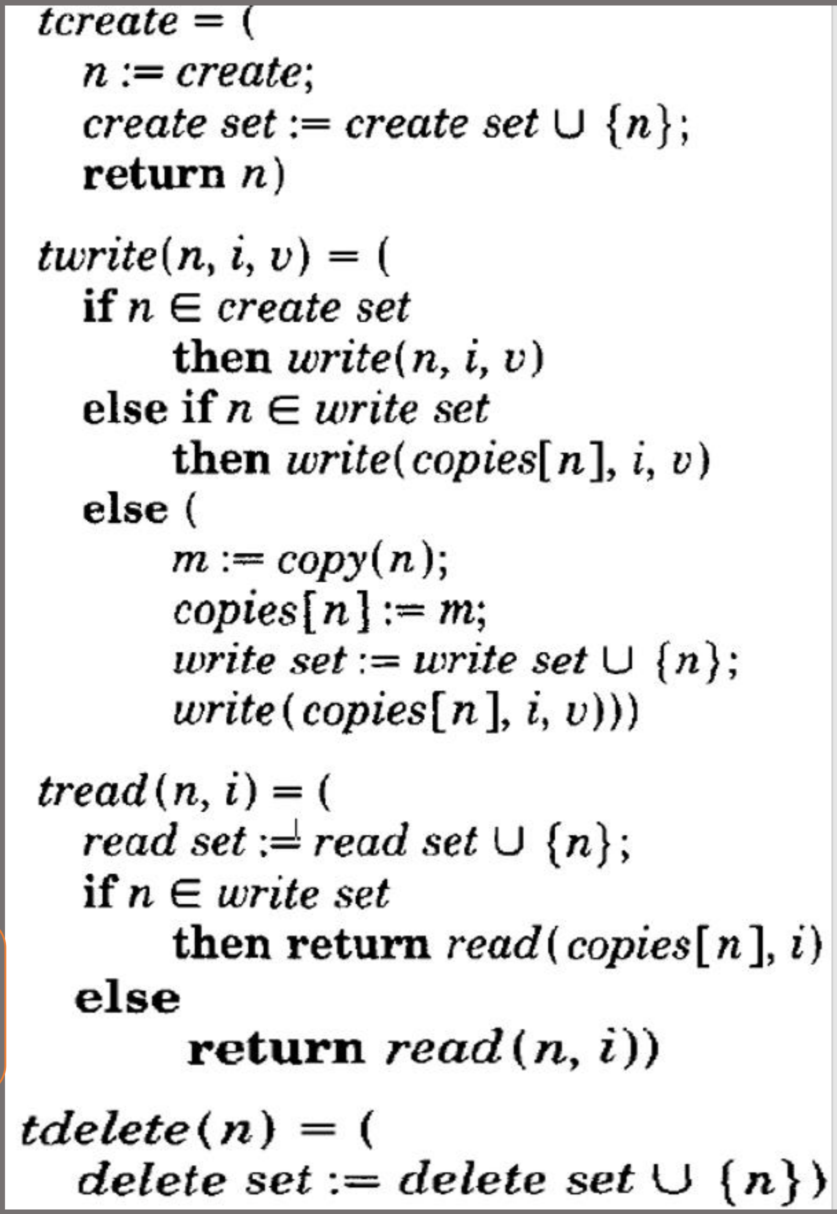
Read value to item in n

No

n ϵ WRITE set?

Yes

Read items to local copy



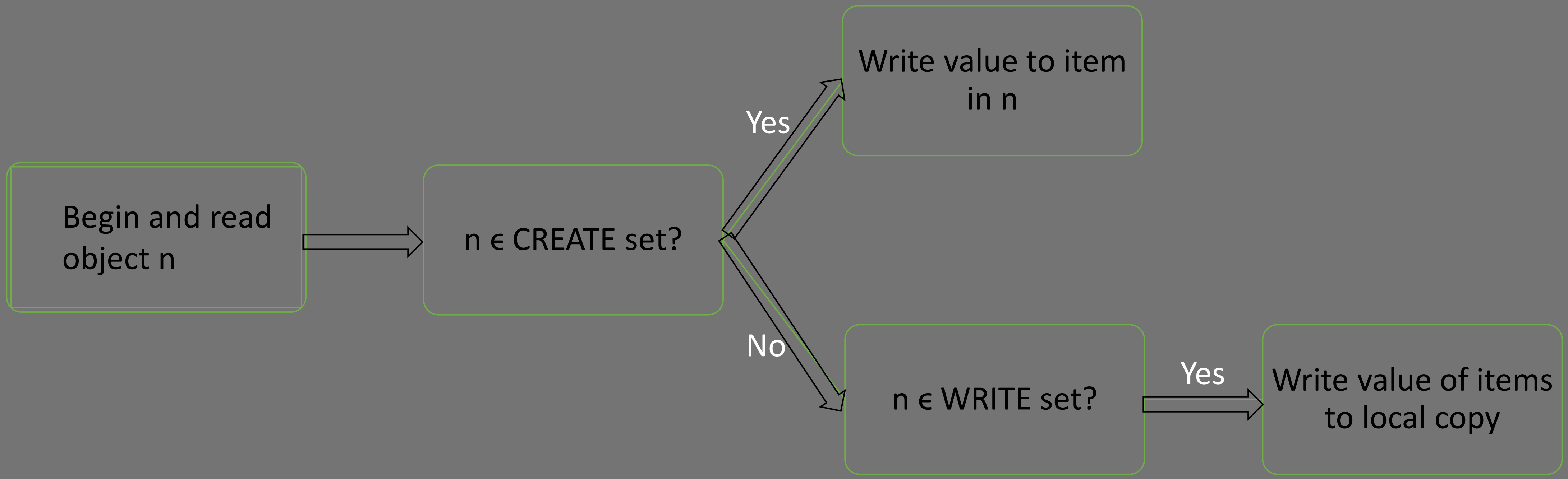
```
tcreate = (
    n := create;
    create set := create set $\cup \{n\}$;
    return n)

twrite(n, i, v) = (
    if $n \in$ create set
        then write(n, i, v)
    else if $n \in$ write set
        then write(copies[n], i, v)
    else (
        m := copy(n);
        copies[n] := m;
        write set := write set $\cup \{n\}$;
        write(copies[n], i, v)))

tread(n, i) = (
    read set := read set $\cup \{n\}$;
    if $n \in$ write set
        then return read(copies[n], i)
    else
        return read(n, i))

tdelete(n) = (
    delete set := delete set $\cup \{n\}$)
```

	create	create a new object and return its name.
	delete(n)	delete object n.
Terminology:	*read(n, i)*	read item i of object n and return its value.
	write (n, i, u)	write u 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 nl and n2.

ReadSet: Set of objects read by Transaction T.

WriteSet: Set of objects modified by Transaction T.

Write phase



Begin and read object n

n ϵ CREATE set?

Yes

Write value to item in n

No

n ϵ WRITE set?

Yes

Write value of items to local copy

Correctness Criterion for validation

- Same effect on database as if all the transactions ran one after the other
- Easy way to validate that every transaction preserves integrity
- If *Tinitial(d)* satisfies all integrity criteria and the concurrent execution of T1, T2,…,Tn are serially equivalent, then *Tfinal(d)* satisfies all integrity criteria

$$d_{\text{final}} = T_{\pi(n)} \circ T_{\pi(n-1)} \circ \cdots \circ T_{\pi(2)} \circ T_{\pi(1)}(d_{\text{initial}})$$

- Easier to verify serial equivalence than check integrity after every interleaving concurrent transactions

Validation of Serial Equivalence

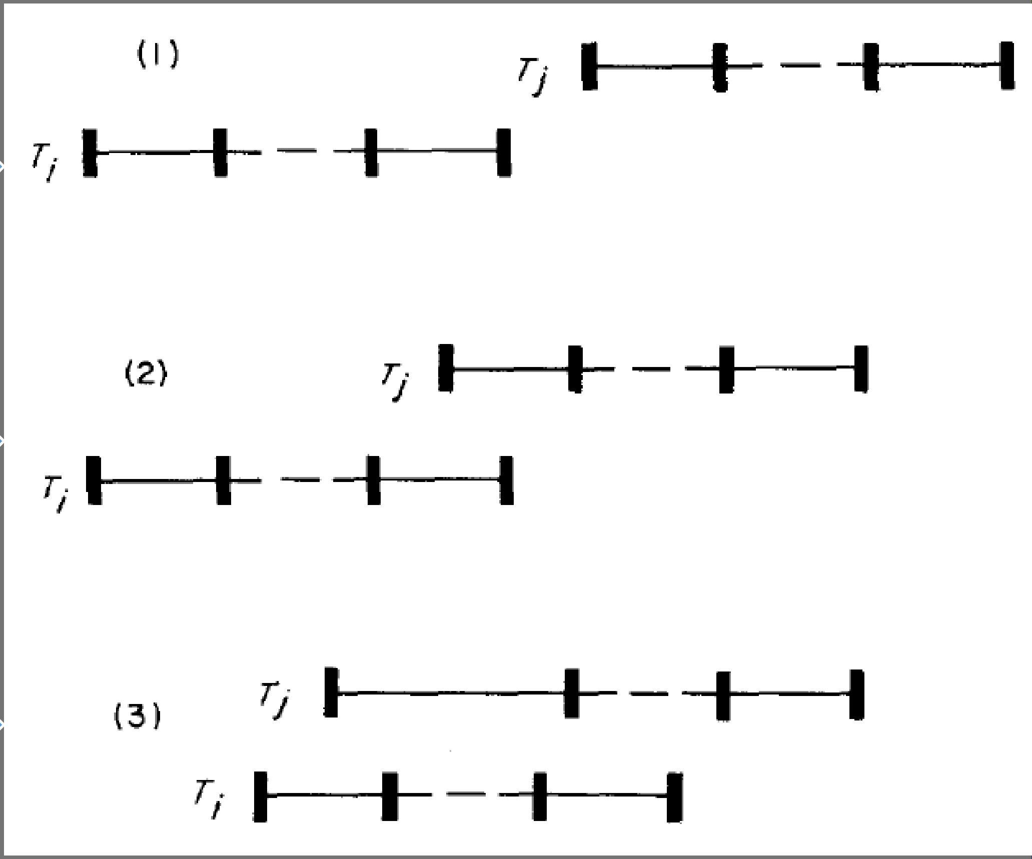
- *Transaction Number* t(i) assigned at the end of the *read* phase
- Transaction numbers are integers assigned in ascending sequence; global transaction number counter
- Transaction number defines its position in time
- If the transaction is validated and completed successfully, number is retained for re-use
- If transaction fails the validation checks and is aborted, or if the transaction is read-only, the number is released for reassignment

Validation conditions

T1 completes its write phase before T2 starts its read phase.

T1 write set does not intersect T2 read set, and T1 completes its write phase before T2 starts its write phase.

T1 write set does not intersect T2 read or write set, and T1 completes its read before T2 completes its read phase.



Why transaction numbers are assigned at the end of read phase instead of beginning?

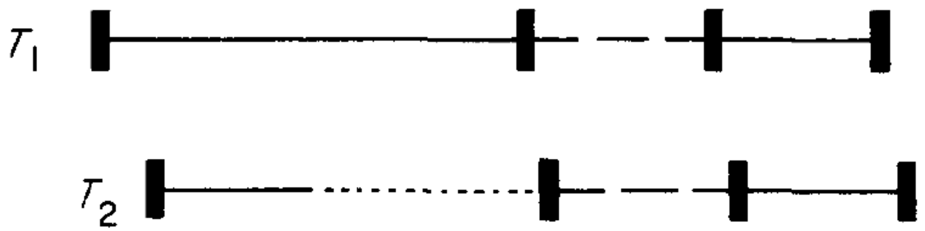


Fig. 3. Transaction 2 waits for transaction 1 in . . . .

Practical considerations

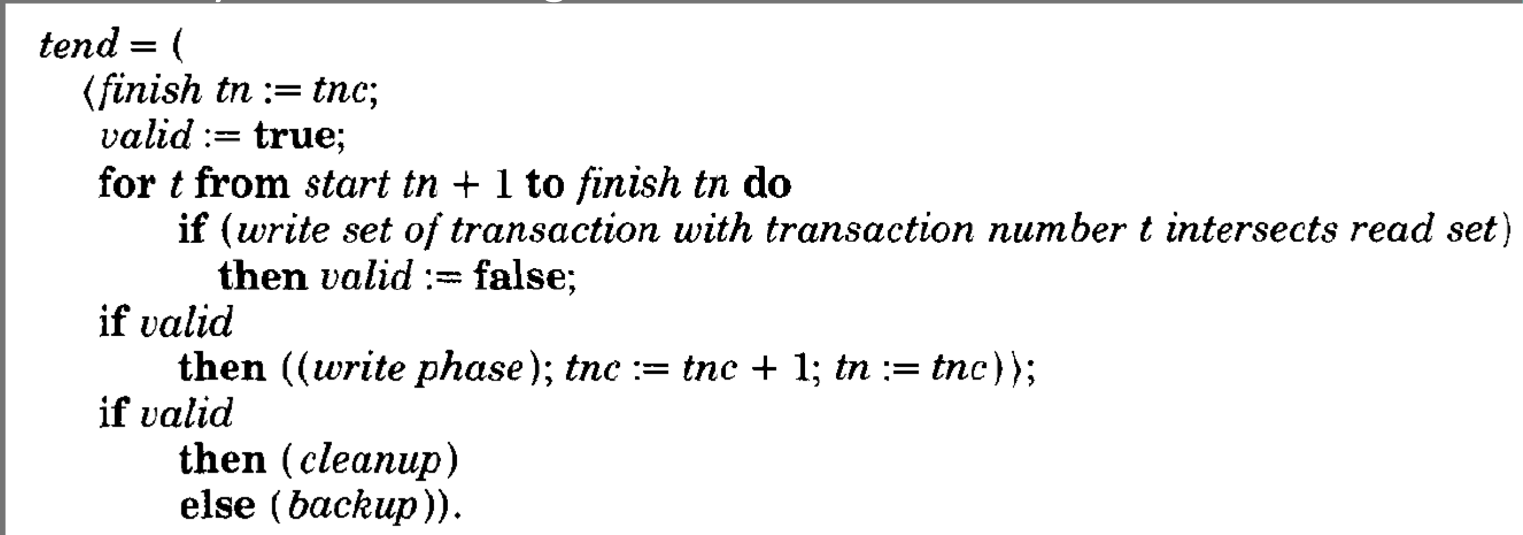
- What happens when validation fails?

  Transaction is aborted and restarted with new transaction number

- What should be done if validation continually fails?

Serial Validation

- This model implements validation conditions (1) and (2) of serial equivalence
- Transaction numbers are assigned only if validation is successful.
- Implementation consists of placing the assignment of Tid, validation, and subsequent write phase all in a section
- Ideal for query dominant systems or systems with single CPU



```
tend = (
  ⟨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
       then (cleanup)
       else (backup)).
```

Parallel Validation

- Concurrency control that uses all three of the validation conditions
- Retains optimization properties of Serial Validation
- Transaction numbers assigned after write phase, if validation succeeds
- *Active* transaction id’s were maintained - transactions which completed read but not yet write
- Extends validation to allow multiple transactions to be in the validation phase at the same time

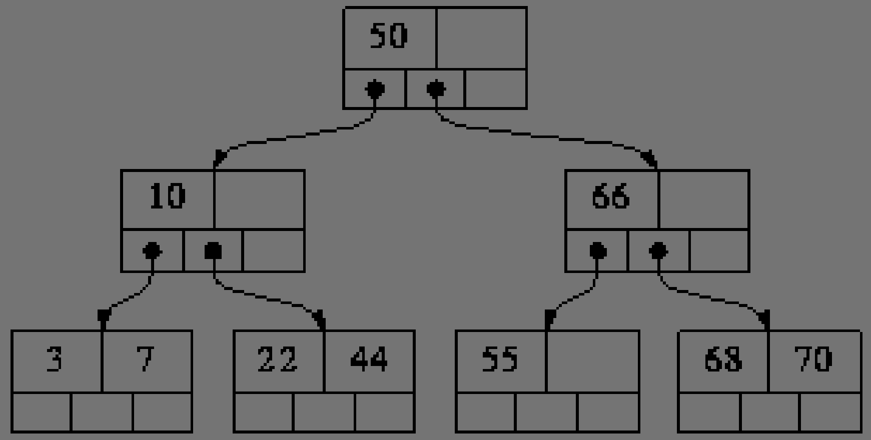
Applications

- Very large Tree structured indexes

  Ex: B-Trees

  B-tree of order 199 contain 200 million keys, the depth is just 5 because 1+ log (N + 1)/2

  Probability of conflict during insertions is very low



Conclusion

Locking approach	Optimistic approach
Controlled by locking	Relies on back up
Serial equivalence by ordering the transactions by first access time	Transactions are ordered by transaction number assignment
Major difficultly: Deadlocks	Major difficulty: Starvation
Suitable for high-conflict concurrent write systems	Best for query dominant systems and very large tree-structured indexes.
Better consistency and isolation	Faster throughput. Parallel validation benefits from multiprocessor environment

Questions Time!