>>

Optimistic Concurrency Control

- Optimistic Concurrency Control
- Validation
- Validation Check

/\ >>

❖ Optimistic Concurrency Control

Locking is a pessimistic approach to concurrency control:

- limit concurrency to ensure that conflicts don't occur

Costs: lock management, deadlock handling, contention.

In scenarios where there are far more reads than writes ...

- don't lock (allow arbitrary interleaving of operations)
- check just before commit that no conflicts occurred
- if problems, roll back conflicting transactions

Optimistic concurrency control (OCC) is a strategy to realise this.

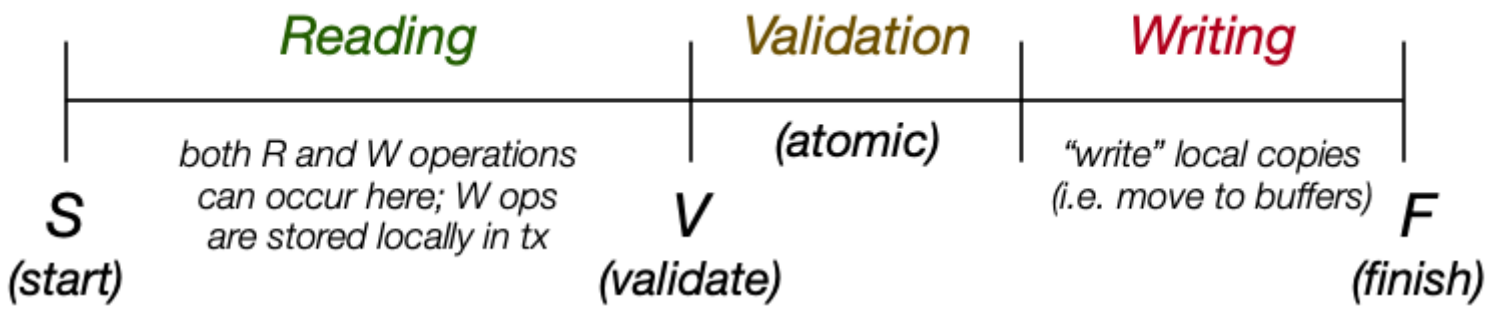
<< ^ >>

❖ Optimistic Concurrency Control (cont)

Under OCC, transactions have three distinct phases:

- Reading: read from database, modify local copies of data
- Validation: check for conflicts in updates
- Writing: commit local copies of data to database

Timestamps are recorded at points *S, V, F*:



<< ^ >>

❖ Validation

Data structures needed for validation:

- S ... set of txs that are reading data and computing results
- V ... set of txs that have reached validation (not yet committed)
- F ... set of txs that have finished (committed data to storage)
- for each T_i, timestamps for when it reached S, V, F
- $RS(T_i)$ set of all data items read by T_i
- $WS(T_i)$ set of all data items to be written by T_i

Use the V timestamps as ordering for transactions

- assume serial tx order based on ordering of $V(T_i)$'s

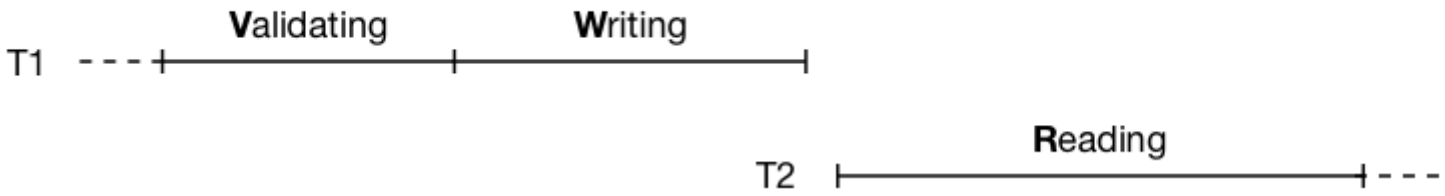
<< ^ >>

❖ Validation (cont)

Two-transaction example:

- allow transactions T_1 and T_2 to run without any locking
- check that objects used by T_2 are not being changed by T_1
- if they are, we need to roll back T_2 and retry

Case 0: serial execution ... no problem

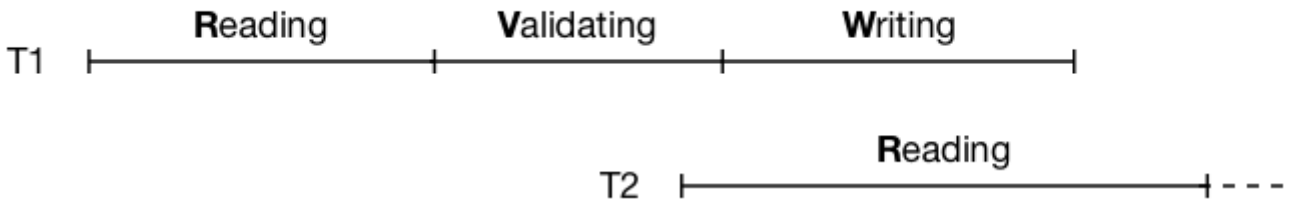


<< ^ >>

❖ Validation (cont)

Case 1: reading overlaps validation/writing

- T_2 starts while T_1 is validating/writing
- if some X being read by T_2 is in $WS(T_1)$
- then T_2 may not have read the updated version of X
- so, T_2 must start again

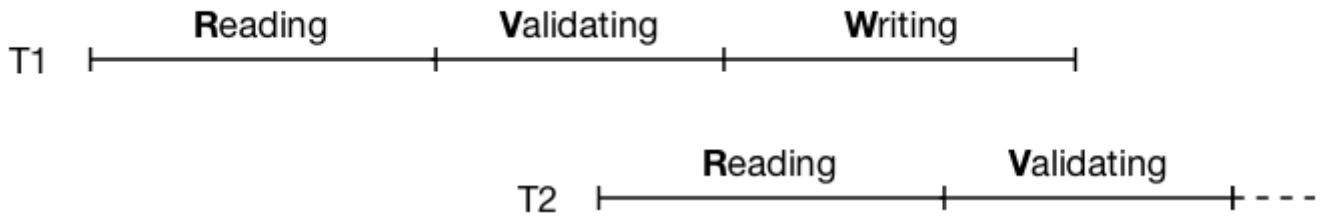


<< ^ >>

❖ Validation (cont)

Case 2: reading/validation overlaps validation/writing

- T_2 starts validating while T_1 is validating/writing
- if some X being written by T_2 is in $WS(T_1)$
- then T_2 may end up overwriting T_1's update
- so, T_2 must start again



❖ Validation Check

Validation check for transaction T

- for all transactions $T_i \neq T$
  - if $T \in S$ & $T_i \in F$, then ok
  - if $T \notin V$ & $V(T_i) < S(T) < F(T_i)$,
    then check $WS(T_i) \cap RS(T)$ is empty
  - if $T \in V$ & $V(T_i) < V(T) < F(T_i)$,
    then check $WS(T_i) \cap WS(T)$ is empty

If this check fails for any T_i, then T is rolled back.

<< ^

❖ Validation Check (cont)

OCC prevents: *T* reading dirty data, *T* overwriting T_i's changes

Problems with OCC:

- increased roll backs**
- tendency to roll back "complete" tx's
- cost to maintain *S,V,F* sets

** "Roll back" is relatively cheap

- changes to data are purely local before Writing phase
- no requirement for logging info or undo/redo (see later)

Produced: 15 Apr 2021