Optimistic Concurrency Control

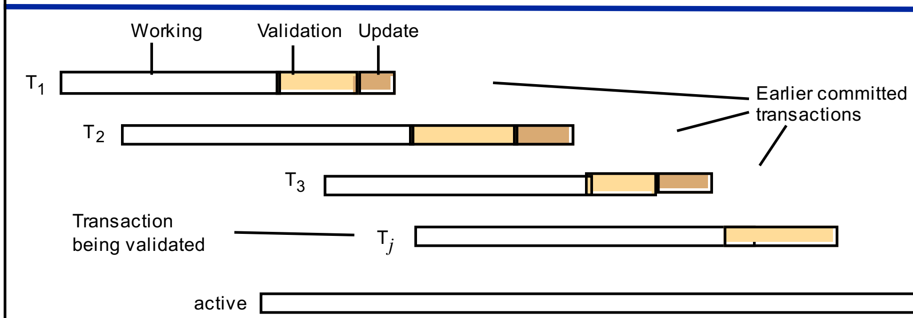
Optimistic CC

- Locking is conservative
  - ☆ Locking overhead even if no conflicts
  - ☆ Deadlock detection/resolution (especially problematic in distributed environment)
- Optimistic concurrency control
  - ☆ Perform operation first
  - ☆ Check for conflicts only later (e.g., at commit time)
- Centralized systems never used textbook optimistic CC
  - ☆ More popular: snapshot isolation
- Interesting alternative for distributed environment

Kung-Robinson Model

- Working Phase:
  - If first operation on X, then access the last committed version of X, make local copy and read/write local copy
  - Otherwise read/write read/write local copy
  - Keep WriteSet containing objects written
  - Keep ReadSet containing objects read
- Validation Phase
  - Check whether transaction conflicts with other transactions
- Update Phase
  - Upon successful validation, local copies of updated objects are are made public. They are committee
- Assumption (for simplicity):
  - Only one transaction may be in validation phase and update phase at a time (e.g., implemented as critical section)

Example



- Validation order determines serialization order
  - if serialization order not possible according to validation order then abort validation transaction
- Validation and update serially
  - All updates are automatically executed in order transactions are validated
  - Validation only needs to check write/read conflicts

Backward Validation

- Maintain transaction counter TC, initialize TC := 0
- Upon begin of transaction T_i (could be first read/write request)
  - StartTC(T_i) := TC
  - CommitTS(T_i) := undefined
  - WriteSet(T_i) := RreadSet(T_i) := {}
- Upon $r_i(x)$, $w_i(x)$ request of T_i
  - If first operation on X, access X (latest committed version) and make local copy
  - read/write local copy of X
  - If $r_i(x)$, then ReadSet(T_i) := ReadSet(T_i) $\cup$ {x}
  - If $w_i(x)$, then WriteSet(T_i) := WriteSet(T_i) $\cup$ {x}
- At time of validation of transaction T_i (in critical section)
  - For all StartTC(T_i) < j <= TC
    - Let WS be the write set of transaction T with CommitTS(T) = j
    - If WS $\cap$ ReadSet(T_i) $\neq$ {}, then abort T_i
  - If not aborted
    - TC = TC + 1
    - CommitTS(T_i) = TC
    - for each x in WriteSet(T_i) local copy of x becomes the official last committed version

Discussion

- Validation and Update Phase in Critical section
  - Reduced concurrency
  - Optimizations possible
- Space overhead:
  - To validate T_i, must have WriteSets for all T_j where $T_j < T_i$ and T_j was active when T_i began.
  - Each transaction must record read/write activity
- No deadlock but potentially more aborts.

Snapshot Isolation

- Multiple versions
- Check write/write conflicts instead of read/writes
- Read snapshot as of start of transactions

Snapshot Isolation

- Maintain transaction counter TC, initialize TC := 0
- Upon begin of transaction T_i (could be first read/write request)
  - ☆ StartTC(T_i) := TC
  - ☆ CommitTS(T_i) := undefined
  - ☆ WriteSet(T_i) := RreadSet(T_i) := {}
- Upon $r_i(x)$, $w_i(x)$ request of T_i
  - ☆ If first operation on X, access version with label TS(X) such that
    - TS (x) <= StartTC(T_i) and
    - No TS'(x) with TS(x) < TS'(x) <= StartTC(T_i)
  - ☆ read/write local copy of X
  - ☆ If $w_i(x)$, then WriteSet(T_i) := WriteSet(T_i) $\cup$ {x}
- At time of validation of transaction T_i (in critical section)
  - ☆ For all StartTC(T_i) < j <= TC
    - Let WS be the write set of transaction T with CommitTS(T) = j
    - If WS $\cap$ WriteSet(T_i) $\neq$ {}, then abort T_i
  - ☆ If not aborted
    - TC = TC + 1
    - CommitTS(T_i) = TC
    - for each x in WriteSet(T_i) write version X with label TC into the data store

Snapshot Isolation vs. Classical Optimistic CC

- ❑ Do not need to keep track of reads
  - ☆ Typically many more reads than writes
  - ☆ Predicate reads difficult to grasp
- ❑ Read-only transactions never need validation
  - ☆ Are serialized at the time point they started
  - ☆ Readers never conflict with writers
- ❑ Natural for append-only stores
  - ☆ Have become popular in the recent past
- ❑ No serializability!!
- ❑ Oracle, PostgreSQL, Microsoft SQL Server, …