Optimistic Concurrency Control

EECS 339

Lecture 17

Basic Two Phase Locking Protocol

- Before every read, acquire a shared lock
- Before every write, acquire an exclusive lock (or "upgrade") a shared to an exclusive lock
- Only release locks after all locks have been acquired
- Lock point = point when all locks are acquired
  - Transaction can run to completion
  - Will be serialized after any conflicting transactions that already completed, and before any other conflict transaction

Motivating Example (2 x RX WX RY WY)

T1	T2

Motivating Example (2 x RX WX RY WY)

T1	T2
Slock X	

Motivating Example (2 x RX WX RY WY)

T1	T2
Slock X	
Read X	

Motivating Example (2 x RX WX RY WY)

T1	T2
Slock X	
Read X	
Xlock X	

Motivating Example (2 x RX WX RY WY)

T1	T2
Slock X	
Read X	
Xlock X	
Write X	

Motivating Example (2 x RX WX RY WY)

T1	T2
Slock X	
Read X	
Xlock X	
Write X	
Release X?	

If T1 releases X at this point, T2 could update X and Y before T1, resulting in a non-serial schedule.

Motivating Example (2 x RX WX RY WY)

T1	T2
Slock X	
Read X	
Xlock X	
Write X	
~~Release X?~~ Slock Y	

Motivating Example (2 x RX WX RY WY)

T1	T2
Slock X	
Read X	
Xlock X	
Write X	
~~Release X?~~ Slock Y	

Motivating Example (2 x RX WX RY WY)

T1	T2
Slock X	
Read X	
Xlock X	
Write X	
~~Release X?~~ Slock Y	
Release X	
	Slock X

Motivating Example (2 x RX WX RY WY)

T1	T2
Slock X	
Read X	
Xlock X	
Write X	
~~Release X?~~ Slock Y	
Release X	
	Slock X
	Read X
	Write X

Motivating Example (2 x RX WX RY WY)

T1	T2
Slock X	
Read X	
Xlock X	
Write X	
~~Release X?~~ Slock Y	
Release X	
	Slock X
	Read X
	Write X
	Acq Y?

Motivating Example (2 x RX WX RY WY)

T1	T2
Slock X	
Read X	
Xlock X	
Write X	
~~Release X?~~ Slock Y	
Release X	
	Slock X
	Read X
	Write X
	Acq Y? (Waits for T1)

Two Problems

- Cascading Aborts
- Deadlocks

Rigorous Two Phase Locking Protocol

- Before every read, acquire a shared lock
- Before every write, acquire an exclusive lock (or "upgrade") a shared to an exclusive lock
- Release locks only after the transaction commits
- Ensures cascadeless-ness, and that commit order = serialization order

Optimistic Concurrency Control

- No locks
- Check for conflict at commit time
- Each xaction stores its writes in a private workspace
- Keep track of all objs read/written by a xaction

OCC Write

```
twrite(object,value):
   if object not in write_set:  // never written, make
copy
       m = read(object)
       copies[object] = m
       write_set = write_set U {object}
   write(copies[object], value)
```

OCC Read

```
tread(object):
    read_set = read_set U {object};
    if object in write_set:
        return read(copies[object]);
    else:
        return read(object);
```

Validation Rules

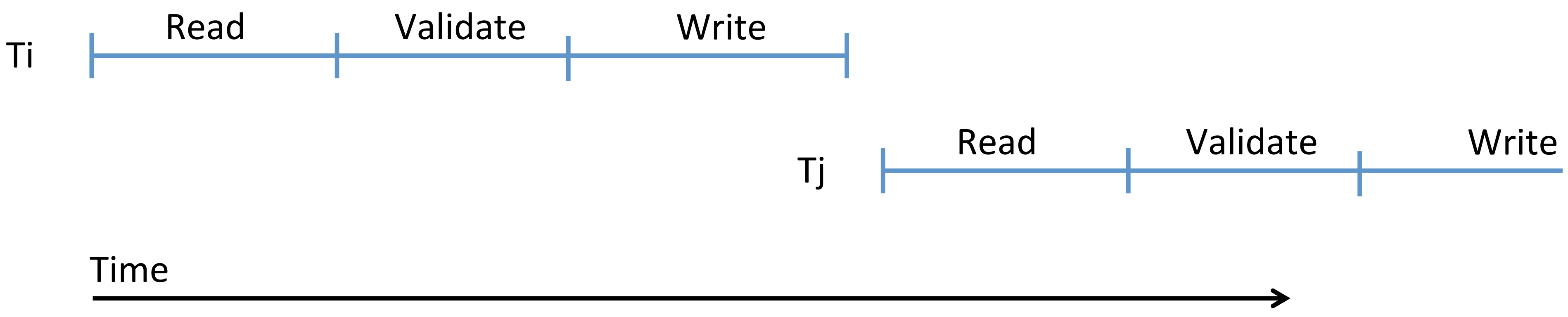
When Tj completes its read phase, require that for all Ti < Tj, one of the following conditions must be true for validation to succeed (Tj to commit):

1) Ti completes its write phase before Tj starts its read phase
2) W(Ti) does not intersect R(Tj), and Ti completes its write phase before Tj starts its write phase.
3) W(Ti) does not intersect R(Tj) or W(Tj), and Ti completes its read phase before Tj completes its read phase.
4) W(Ti) does not intersect R(Tj) or W(Tj), and W(Tj) does not intersect R(Ti) [no conflicts]

These rules will ensure serializability, with Tj being ordered after Ti with respect to conflicts

Condition 1

Ti completes its write phase before Tj starts its read phase

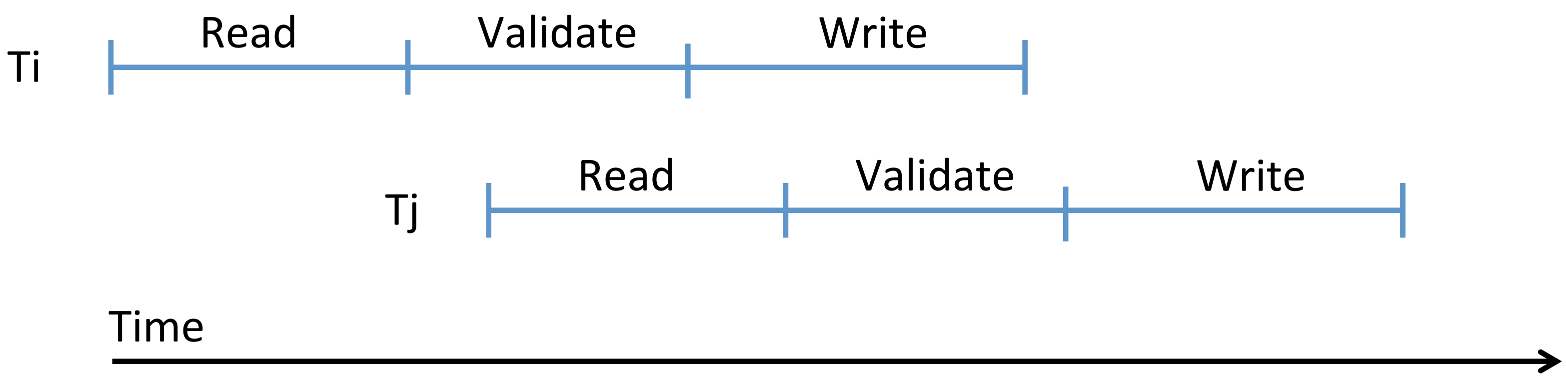


Don't overlap at all.

Condition 2

W(Ti) does not intersect R(Tj), and Ti completes its write phase before Tj starts its write phase.

$W(Ti) \cap R(Tj) = \{\ \}$ $\quad$ $R(Ti) \cap W(Tj) \neq \{\ \}$ $\quad$ $W(Ti) \cap W(Tj) \neq \{\ \}$

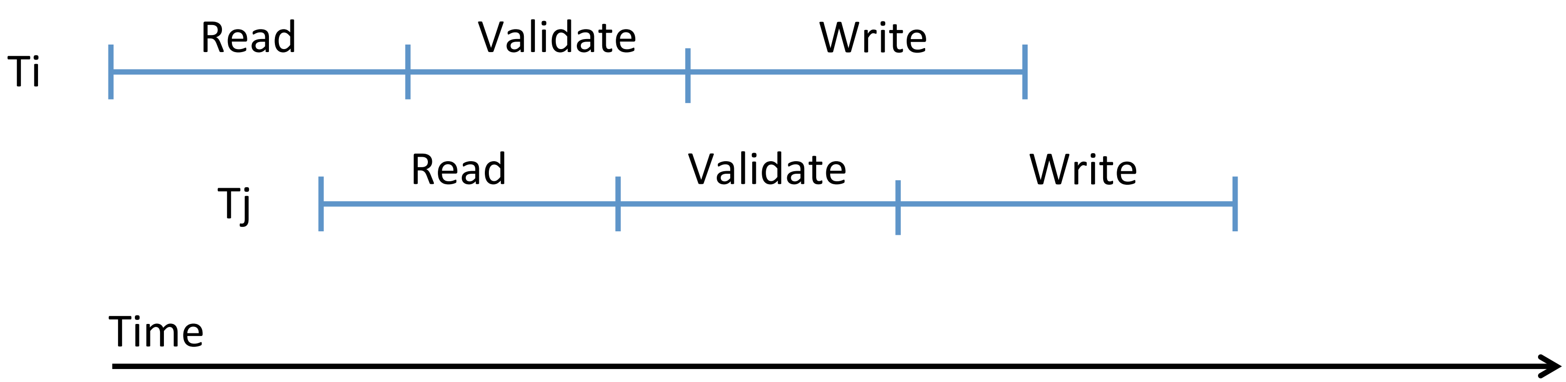


Tj doesn't read anything Ti wrote.
Anything Tj wrote that Ti also wrote will be installed afterwards.
Anything Ti read will not reflect Tjs writes

Condition 3

W(Ti) does not intersect R(Tj) or W(Tj), and Ti completes its read phase before Tj completes its read phase.

$W(Ti) \cap R(Tj) = \{\}$ $R(Ti) \cap W(Tj) \neq \{\}$ $W(Ti) \cap W(Tj) = \{\}$



Tj doesn't read or write anything Ti wrote (but Ti may read something Tj writes).

Ti definitely won't see any of Tj's writes, because it finishes reading before Tj starts validation, so Ti ordered before Tj.

Ti will always complete its read phase before Tj b/c timestamps assigned after read phase

Applying Conds 1 & 2: Serial Validation

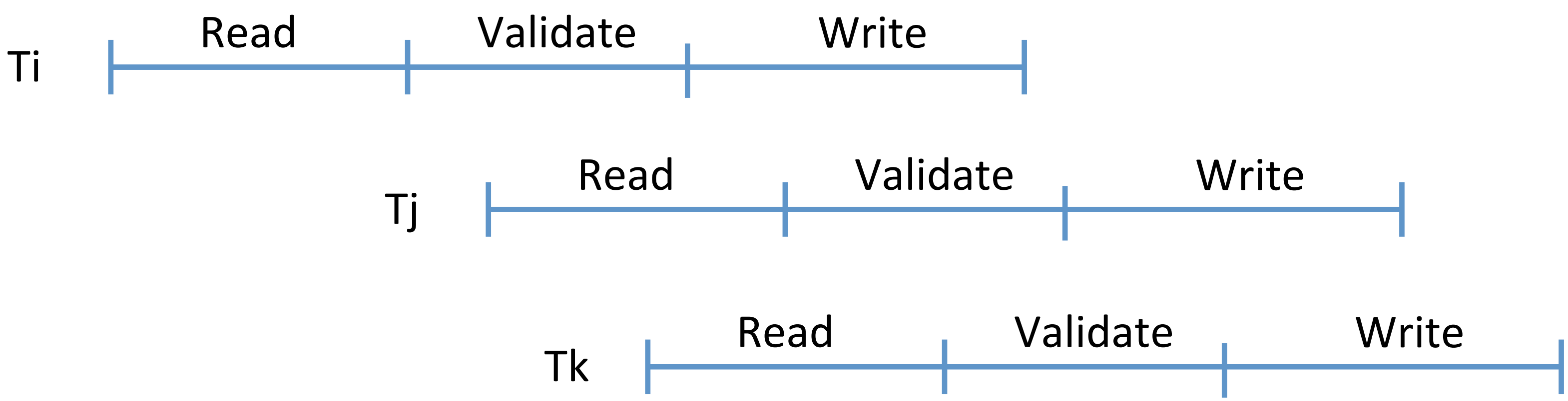
- To validate Xact T:

```
valid = true;
// S = set of Xacts that committed after Begin(T)
< foreach  Ts in S do {
  if ReadSet(Ts) does not intersect WriteSet(Ts)
    then valid = false;
  }
  if valid then { install updates; // Write phase
                  Commit T } >
                else Restart T
```

end of critical section

Study Break: Optimistic Concurrency

- What transaction(s) succeed? Which of the three conditions make it possible?



R(Ti) = d
W(Ti) = a, b, c

R(Tj) = d
W(Tj) = b, c

R(Tk) = d
W(Tk) = d

Comments on Serial Validation

- Applies Test 2, with T playing the role of Tj and each Xact in Ts (in turn) being Ti.
- Assignment of Xact id, validation, and the Write phase are inside a **critical section**!
  - I.e., Nothing else goes on concurrently.
  - If Write phase is long, major drawback.
- Optimization for Read-only Xacts:
  - Don't need critical section (because there is no Write phase).

Serial Validation (Contd.)

- Multistage serial validation: Validate in stages, at each stage validating T against a subset of the Xacts that committed after Begin(T).
  – Only last stage has to be inside critical section.
- Starvation: Run starving Xact in a critical section (!!)
- Space for WriteSets: To validate Tj, must have WriteSets for all Ti where Ti < Tj and Ti was active when Tj began. There may be many such Xacts, and we may run out of space.
  – Tj's validation fails if it requires a missing WriteSet.

Overhead of Optimistic CC

- Must record read/write activity in ReadSet and WriteSet per Xact.
  - Must create and destroy these sets as needed.
- Must check for conflicts during validation, and must make validated writes ``global''.
  - Critical section can reduce concurrency.
  - Scheme for making writes global can reduce clustering of objects.
- Optimistic CC restarts Xacts that fail validation.
  - Work done so far is wasted; requires clean-up.

Timestamp Validation

- Maintain timestamps for when each xaction begins and the last time an object was read/written
- Detect conflicts at write time by comparing timestamps
  - Abort as needed

Multiversion Concurrency Control

- To minimize aborts maintain multiple versions of the same object as writes are completed
- Each transaction sees a snapshot of the data at a given time
- If Ti tries to write to something that was read after the xaction started, it will abort

Summary

- Many schemes for managing db concurrency
- Locking is pessimistic – lose time waiting for shared objs
- Optimistic concurrency control detects conflicts when or shortly after they happen
- Best scheme depends on conflict rate
  - OCC best for low conflict – 2PL for mostly overlapping xactions