2PL and OCC

CS 475: Concurrent & Distributed Systems (Fall 2021)

Lecture 15

Yue Cheng

Some material taken/derived from:

- Princeton COS-418 materials created by Michael Freedman and Kyle Jamieson.
- MIT 6.824 by Robert Morris, Frans Kaashoek, and Nickolai Zeldovich.

Recap: Transaction serializability

Serializability:

Execution of a set of transactions over multiple items is equivalent to ***some*** serial execution of transactions

Some new terms

Lost update: the result of a transaction is overwritten by another transaction

Some new terms

Lost update: the result of a transaction is overwritten by another transaction

Dirty read: uncommitted results are read by a transaction

Some new terms

Lost update: the result of a transaction is overwritten by another transaction

Dirty read: uncommitted results are read by a transaction

Non-repeatable read: two reads in the same transaction return different results

Some new terms

Lost update: the result of a transaction is overwritten by another transaction

Dirty read: uncommitted results are read by a transaction

Non-repeatable read: two reads in the same transaction return different results

Phantom read: later reads in the same transaction return extra rows

Serial schedule – No problem

T1: R(A), W(A), R(B), W(B), Abort

T2: R(A), W(A), Commit

time

Quiz: Which concurrency problem is this?

T1: R(A), W(A) R(B), W(B), Abort

T2: R(A), W(A), Commit

time

Lost update Dirty read Non-repeatable read Phantom read ??

Quiz: Which concurrency problem is this?

T1: R(A)		R(A), W(A), Commit
T2:	R(A), W(A), Commit	

time →

Lost update   Dirty read   Non-repeatable read   Phantom read   ??

Quiz: Which concurrency problem is this?

T1: R(A), W(A) W(B), Commit

T2: R(A) W(A), W(B), Commit

time

Lost update Dirty read Non-repeatable read Phantom read ??

Quiz: Which concurrency problem is this?

T1: R(A), W(A) W(A), Commit

T2: R(A), R(B), W(B), Commit

time

Lost update Dirty read Non-repeatable read Phantom read ??

Q: How to ensure *correctness* when running concurrent transactions?

What does correctness mean?

Transactions should have property of *isolation*, i.e., all operations in a transaction appear to happen together at the same time

What does correctness mean?

Transactions should have property of *isolation*, i.e., all operations in a transaction appear to happen together at the same time

We need serializability

Fixing concurrency problems

Strawman: Just run transactions serially — prohibitively bad performance

Fixing concurrency problems

Strawman: Just run transactions serially — prohibitively bad performance

Observation: Problems only arise when:

1. Two transactions touch the same data
2. At least one of these transactions involves a *write* to the data

Fixing concurrency problems

Strawman: Just run transactions serially — prohibitively bad performance

Observation: Problems only arise when:

1. Two transactions touch the same data
2. At least one of these transactions involves a *write* to the data

Key idea: Only permit schedules whose effects are guaranteed to be *equivalent* to serial schedules

Serializability of schedules

Two operations conflict if

1. They belong to different transactions
2. They operate on the same data
3. One of them is a write

Serializability of schedules

Two operations conflict if

1. They belong to different transactions
2. They operate on the same data
3. One of them is a write

Two schedules are equivalent if

1. They involve the same transactions and operations
2. All *conflicting* operations are ordered the same way

Serializability of schedules

Two operations conflict if

1. They belong to different transactions
2. They operate on the same data
3. One of them is a write

Two schedules are equivalent if

1. They involve the same transactions and operations
2. All *conflicting* operations are ordered the same way

A schedule is **serializable** if it is equivalent to a serial schedule

Testing for serializability

Intuition: Swap non-conflicting operations until you reach a serial schedule

Testing for serializability

Intuition: Swap non-conflicting operations until you reach a serial schedule

T1: R(A), W(A), Commit

T2: R(A), R(B), W(B), Commit

time

Testing for serializability

Intuition: Swap non-conflicting operations until you reach a serial schedule

T1: R(A), W(A), Commit

T2: R(A), R(B), W(B), Commit

time

Testing for serializability

Intuition: Swap non-conflicting operations until you reach a serial schedule

T1: R(A), W(A), Commit

T2: R(A), R(B), W(B), Commit

time

Testing for serializability

Intuition: Swap non-conflicting operations until you reach a serial schedule

T1: R(A), W(A), Commit

T2: R(A), R(B), W(B) Commit

time

Serializable

Testing for serializability

Intuition: Swap non-conflicting operations until you reach a serial schedule

T1: R(A), W(A), W(B), Commit

T2: R(B), W(B), R(A), Commit

time

Testing for serializability

Intuition: Swap non-conflicting operations until you reach a serial schedule

T1: R(A), W(A), W(B), Commit

T2: R(B), W(B), R(A), Commit

time

Testing for serializability

Intuition: Swap non-conflicting operations until you reach a serial schedule

T1: R(A), W(A) W(B), Commit

T2: R(B), W(B), R(A), Commit

time

Testing for serializability

Intuition: Swap non-conflicting operations until you reach a serial schedule

T1: R(A), W(A), W(B), Commit

T2: R(B), W(B), R(A), Commit

time

NOT serializable

Testing for serializability

Another way to test serializability

- Draw arrows between conflicting operations
- Arrow points in the direction of time
- If no cycles between transactions, the schedule is serializable

Testing for serializability

Another way to test serializability

- Draw arrows between conflicting operations
- Arrow points in the direction of time
- If no cycles between transactions, the schedule is serializable

T1: R(A), W(A), Commit

T2: R(A), R(B), W(B), Commit

time

Testing for serializability

Another way to test serializability

- Draw arrows between conflicting operations
- Arrow points in the direction of time
- If no cycles between transactions, the schedule is serializable

T1: R(A), W(A), Commit

T2: R(A), R(B), W(B), Commit

time

Testing for serializability

Another way to test serializability

- Draw arrows between conflicting operations
- Arrow points in the direction of time
- If no cycles between transactions, the schedule is serializable

T1: R(A), W(A), Commit

T2: R(A), R(B), W(B), Commit

time

Testing for serializability

Another way to test serializability

- Draw arrows between conflicting operations
- Arrow points in the direction of time
- If no cycles between transactions, the schedule is serializable

T1: R(A), W(A), Commit

T2: R(A), R(B), W(B), Commit

time

No cycles,

serializable

Testing for serializability

Another way to test serializability

- Draw arrows between conflicting operations
- Arrow points in the direction of time
- If no cycles between transactions, the schedule is serializable

T1: R(A), W(A), W(B), Commit

T2: R(B), W(B), R(A), Commit

time

Testing for serializability

Another way to test serializability

- Draw arrows between conflicting operations
- Arrow points in the direction of time
- If no cycles between transactions, the schedule is serializable

T1: R(A), W(A), W(B), Commit

T2: R(B), W(B), R(A), Commit

time

Testing for serializability

Another way to test serializability

- Draw arrows between conflicting operations
- Arrow points in the direction of time
- If no cycles between transactions, the schedule is serializable

T1: R(A), W(A), W(B), Commit

T2: R(B), W(B), R(A), Commit

time

Testing for serializability

Another way to test serializability

- Draw arrows between conflicting operations
- Arrow points in the direction of time
- If no cycles between transactions, the schedule is serializable

T1: R(A), W(A), W(B), Commit

T2: R(B), W(B), R(A), Commit

time

Cycle exists

(T1 ⇄ T2),

NOT serializable

Linearizability vs. Serializability

- **Linearizability:** a guarantee about **single** operations on **single** objects
  - Once write completes, all later reads (by wall clock) should reflect that write
- **Serializability** is a guarantee about **transactions** over **one or more** objects
  - Doesn't impose real-time constraints

- Linearizability + serializability = **strict serializability**
  - Transaction behavior equivalent to some serial execution
    - And that serial execution agrees with real-time

Lock-based concurrency control

- Big Global Lock: Results in a **serial** transaction schedule at the cost of performance

- 2PL: Two-phase locking with finer-grain locks:
  - Growing phase when txn acquires locks
  - Shrinking phase when txn releases locks (typically commit)
  - Allows txns to execute concurrently, improving performance

2PL

- 2PL guarantees serializability by disallowing cycles between txns

- There could be dependencies in the waits-for graph among txns waiting for locks:
  - Edge from T2 to T1 means T1 acquired lock first and T2 has to wait
  - Edge from T1 to T2 means T1 acquired lock first and T1 has to wait
  - Cycles mean DEADLOCK, and in that case 2PL won't proceed

2PL

T1: R(A), W(A), W(B), Commit

T2: R(B), W(B), R(A) Commit

time

Deal with deadlocks by aborting one of the twn txns (e.g., detect with timeout)

2PL

Lock_X(A)

T1: R(A), W(A), W(B), Commit

T2: R(B), W(B), R(A) Commit

time

2PL

Lock_X(A)

T1: R(A), W(A), W(B), Commit

T2: R(B), W(B), R(A) Commit

Lock_X(B)

time

2PL

Lock_X(A)

T1: R(A), W(A), W(B), Commit

T2: R(B), W(B), R(A) Commit

Lock_X(B) Lock_S(A)

time

2PL

Lock_X(A) Lock_X(B)

T1: R(A), W(A), W(B), Commit

T2: R(B), W(B), R(A) Commit

Lock_X(B) Lock_S(A)

time

2PL

Lock_X(A) Lock_X(B)

T1: R(A), W(A), W(B), Commit

T2: R(B), W(B), R(A) Commit DEADLOCK!

Lock_X(B) Lock_S(A)

time

2PL

Lock_X(A) Lock_X(B)

T1: R(A), W(A), W(B), Commit

T2: R(B), W(B), R(A) Commit DEADLOCK!

Lock_X(B) Lock_S(A)

time

Deal with deadlocks by aborting one of the two txns (e.g., detect with timeout)

2PL: Releasing locks too soon?

What if we release the lock as soon as we can?

2PL: Releasing locks too soon?

What if we release the lock as soon as we can?

T1: R(A), W(A), Abort

T2: R(B), W(B), R(A) Abort

time

2PL: Releasing locks too soon?

What if we release the lock as soon as we can?

Lock_X(A)

T1: R(A), W(A), Abort

T2: R(B), W(B), R(A) Abort

time

2PL: Releasing locks too soon?

What if we release the lock as soon as we can?

Lock_X(A) Unlock_X(A)

T1: R(A), W(A), Abort

T2: R(B), W(B), R(A) Abort

time

2PL: Releasing locks too soon?

What if we release the lock as soon as we can?

Lock_X(A) Unlock_X(A)

T1: R(A), W(A), Abort

T2: R(B), W(B), R(A) Abort

Lock_X(B)

time

2PL: Releasing locks too soon?

What if we release the lock as soon as we can?

Lock_X(A) Unlock_X(A)

T1: R(A), W(A), Abort

T2: R(B), W(B), R(A) Abort

time

Lock_X(B) Lock_S(A)

2PL: Releasing locks too soon?

What if we release the lock as soon as we can?

Lock_X(A)   Unlock_X(A)

T1: R(A), W(A),   Abort

T2:   R(B), W(B), R(A)   Abort

time

Lock_X(B)   Lock_S(A)

Rollback of T1 requires rollback of T2, since T2 reads a value written by T1

2PL: Releasing locks too soon?

What if we release the lock as soon as we can?

Lock_X(A) Unlock_X(A)

T1: R(A), W(A), Abort

T2: R(B), W(B), R(A) Abort

Lock_X(B) Lock_S(A)

time

Rollback of T1 requires rollback of T2, since T2 reads a value written by T1

Cascading aborts: the rollback of one txn causes rollback of another

Strict 2PL

- Release locks at the end of the transaction

- Variant of 2PL implemented by most DBs in practice

Q: What if access patterns rarely, if ever, conflict?

Today

- Optimistic concurrency control (OCC)
  - Be optimistic, or opportunistic that conflicts rarely happen

Be optimistic!

- Goal: Low overhead for non-conflicting txns
- Assume success!
  - Process transaction as if would succeed
  - Check for serializability only at commit time
  - If fails, abort transaction
- Optimistic Concurrency Control (OCC)
  - **Higher performance** when few conflicts vs. locking
  - **Lower performance** when many conflicts vs. locking

OCC: Three-phase approach

- Begin: Record timestamp marking the transaction's beginning

OCC: Three-phase approach

- Begin: Record timestamp marking the transaction's beginning
- Modify phase:
  - Txn can read values of committed data items
  - Updates only to local copies (versions) of items (in DB cache)

OCC: Three-phase approach

- Begin: Record timestamp marking the transaction's beginning
- Modify phase:
  - Txn can read values of committed data items
  - Updates only to local copies (versions) of items (in DB cache)
- Validate phase

OCC: Three-phase approach

- Begin: Record timestamp marking the transaction's beginning
- Modify phase:
  - Txn can read values of committed data items
  - Updates only to local copies (versions) of items (in DB cache)
- Validate phase
- Commit phase
  - If validates, transaction's updates applied to DB
  - Otherwise, transaction restarted
  - Care must be taken to avoid "TOCTTOU" issues

OCC: Three-phase approach

- **Begin:** Record timestamp marking the transaction's beginning
- Modify phase: Execute optimistically!
  - Txn can read values of committed data items
  - Updates only to local copies (versions) of items (in DB cache)
- Validate phase
- Commit phase
  - If validates, transaction's updates applied to DB
  - Otherwise, transaction restarted
  - Care must be taken to avoid "TOCTTOU" issues

OCC: Three-phase approach

- **Begin:** Record timestamp marking the transaction's beginning
- **Modify** phase:
  - Txn can read values of committed data items
  - Updates only to local copies (versions) of items (in DB cache)

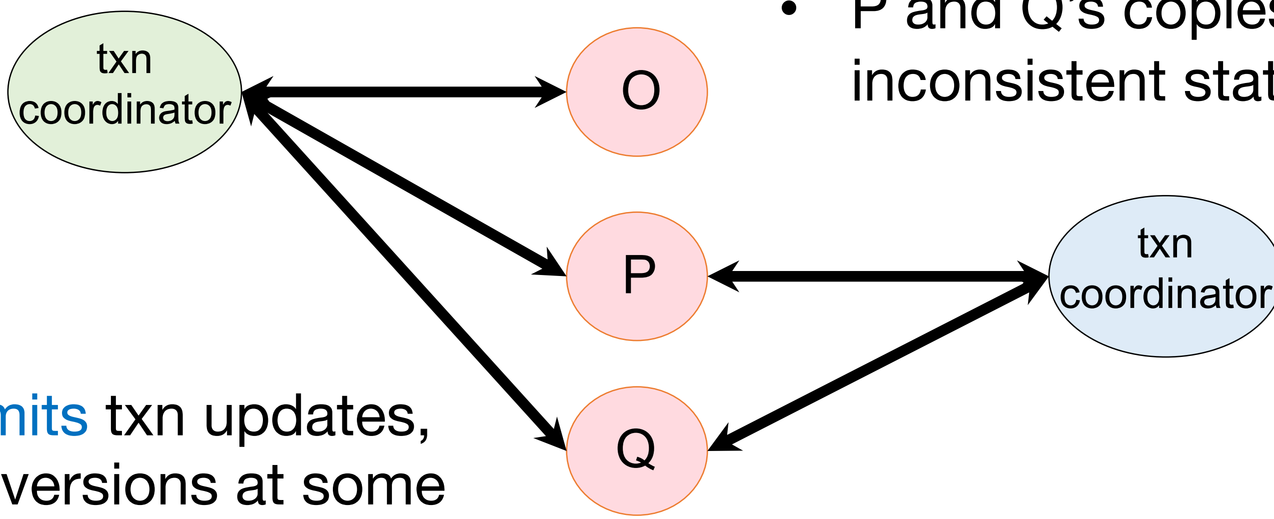
Execute optimistically!

- **Validate** phase
- **Commit** phase
  - If validates, transaction's updates applied to DB
  - Otherwise, transaction restarted
  - Care must be taken to avoid "TOCTTOU" issues

These should happen together!

OCC: Why validation is necessary!

- New txn creates shadow copies of P and Q
- P and Q's copies at inconsistent state



When commits txn updates, create new versions at some timestamp t

OCC: Validate phase

- Transaction is about to commit. System must ensure:
  - Initial consistency: Versions of accessed objects at start consistent
  - No conflicting concurrency: No other txn has committed an operation at object that conflicts with one of this txn's invocations

- Consider transaction 1: For all other txns N either committed or in validation phase, one of the following holds:
  A. N completes commit before 1 starts modify
  B. 1 starts commit after N completes commit, and ReadSet 1 and WriteSet N are disjoint
  C. Both ReadSet 1 and WriteSet 1 are disjoint from WriteSet N, and N completes modify phase

- When validating 1, first check (A), then (B), then (C). If all fail, validation fails and 1 aborted

Atomic commit for OCC

- Use two-phase commit (2PC) to achieve atomic commit (validate + commit writes)

- Recall 2PC protocol:
  1. Coordinator sends *prepare* messages to all nodes, other nodes vote *yes* or *no*
     a. If all nodes accept, proceed
     b. If any node declines, abort
  2. Coordinator sends *commit* or *abort* messages to all nodes, and all nodes act accordingly

Atomic commit for OCC

- Execute optimistically: Read committed values, write changes locally
- Validate: Check if data has changed since original read
- Commit (Write): Commit if no change, else abort

Phase 1

Phase 2

Atomic commit for OCC

- Execute optimistically: Read committed values, write changes locally
- Validate: Check if data has changed since original read — Phase 1
- Commit (Write): Commit if no change, else abort — Phase 2

- Phase 1: send *prepare* to each shard: include buffered write + original reads for that shard
  - Shards validate reads and acquire locks (exclusive for write locations, shared for read locations)
  - If this succeeds, respond with *yes*; else respond with *no*

Atomic commit for OCC

- Execute optimistically: Read committed values, write changes locally
- Validate: Check if data has changed since original read — Phase 1
- Commit (Write): Commit if no change, else abort — Phase 2

- Phase 1: send *prepare* to each shard: include buffered write + original reads for that shard
  - Shards validate reads and acquire locks (exclusive for write locations, shared for read locations)
  - If this succeeds, respond with *yes*; else respond with *no*
- Phase 2: collect votes, send result (*abort* or *commit*) to all shards
  - If commit, shards apply buffered writes
  - All shards release locks

Two ways of implementing serializability: 2PL, OCC

- 2PL (pessimistic):
  - Assume conflict, always lock
  - High overhead for non-conflicting txn
  - Must check for deadlock

- OCC (optimistic):
  - Assume no conflict
  - Low overhead for low-conflict workloads (but high for high-conflict workloads)
  - Ensure correctness by aborting txns if conflict occurs

Lock_X(A) <granted>	
Read(A)	Lock_S(A)
A := A-50	
Write(A)	
Unlock(A)	<granted>
	Read(A)
	Unlock(A)
	Lock_S(B) <granted>
Lock_X(B)	
	Read(B)
<granted>	Unlock(B)
Read(B)	
B := B +50	
Write(B)	
Unlock(B)	

Is this a 2PL schedule?

No

Is this a serializable schedule?

No

Lock_X(A) <granted>	
Read(A)	Lock_S(A)
A := A-50	
Write(A)	
Lock_X(B) <granted>	
Unlock(A)	<granted>
	Read(A)
	Lock_S(B)
Read(B)	
B := B +50	
Write(B)	
Unlock(B)	<granted>
	Unlock(A)
	Read(B)
	Unlock(B)

Is this a 2PL schedule?

Yes, and it is serializable

Is this a Strict 2PL schedule?

No, cascading aborts possible

Lock_X(A) <granted>	
Read(A)	Lock_S(A)
A := A-50	
Write(A)	
Lock_X(B) <granted>	
Read(B)	
B := B +50	
Write(B)	
Unlock(A)	
Unlock(B)	<granted>
	Read(A)
	Lock_S(B) <granted>
	Read(B)
	Unlock(A)
	Unlock(B)

Is this a 2PL schedule?

Yes, and it is serializable

Is this a Strict 2PL schedule?

Yes, cascading aborts not possible