Concurrency Control II (OCC, MVCC)

COS 418: *Distributed Systems*
Lecture 18

Michael Freedman

Serializability

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

2

Lock-based concurrency control

- **Big Global Lock:** Results in a **serial** transaction schedule at the cost of performance
- **Two-phase locking with finer-grain locks:**
  - **Growing phase** when txn acquires locks
  - **Shrinking phase** when txn releases locks (typically commit)
  - Allows txn to execute concurrently, improving performance

3

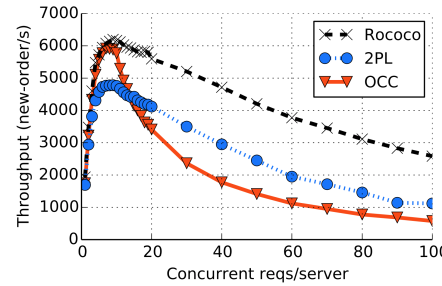
Q: What if access patterns rarely, if ever, conflict?

4

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

2PL vs OCC



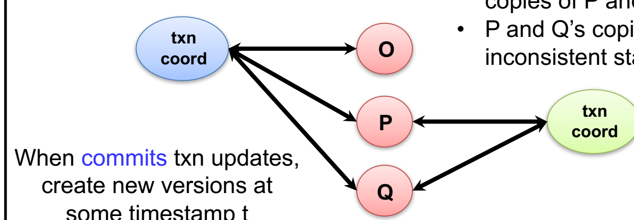
- From "Rococo" paper in OSDI 2014. Focus on 2PL vs. OCC.
- Observe OCC better when write rate lower (fewer conflicts), worse than 2PL with write rate higher (more conflicts)

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: Why validation is necessary

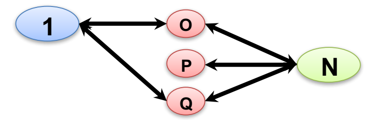
- New txn creates shadow copies of P and Q
- P and Q's copies at inconsistent state



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.

OCC: Validate Phase



A. N completes commit before 1 starts modify
   - Remember that modify includes both read & write. So this just says N finishes before 1 actually starts any read/write → no conflict

B. 1 starts commit after N completes commit, and ReadSet 1 and WriteSet N are disjoint
   - Nothing 1 has recently read depends on what N has written, and 1's writes will all be serialized after N's (even though may overwrite N's values)

C. Both ReadSet 1 and WriteSet 1 are disjoint from WriteSet N, and N completes modify phase.
   - If N has already finished reads (during modify), so it's reads won't depend on WriteSet 1, and similarly, 1's reads don't depend on N's writes.

2PL & OCC = strict serialization

- Provides semantics as if only one transaction was running on DB at time, in serial order

  + Real-time guarantees

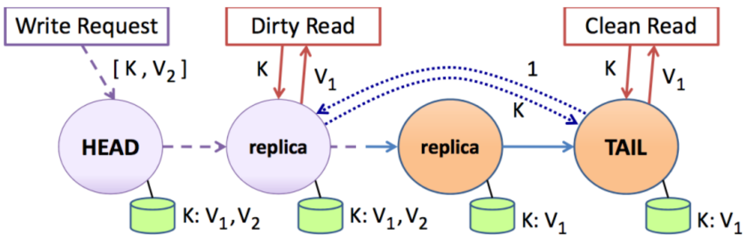
- 2PL: Pessimistically get all the locks first
- OCC: Optimistically create copies, but then recheck all read + written items before commit

Multi-version concurrency control

Generalize use of multiple versions of objects

Multi-version concurrency control

- Maintain multiple versions of objects, each with own timestamp. Allocate correct version to reads.
- Prior example of MVCC:



Multi-version concurrency control

- Maintain multiple versions of objects, each with own timestamp. Allocate correct version to reads.
- Unlike 2PL/OCC, reads never rejected
- Occasionally run garbage collection to clean up

MVCC Intuition

- Split transaction into read set and write set
  - All reads execute as if one "snapshot"
  - All writes execute as if one later "snapshot"
- Yields snapshot isolation < serializability

Serializability vs. Snapshot isolation

- Intuition: Bag of marbles: ½ white, ½ black
- Transactions:
  - T1: Change all white marbles to black marbles
  - T2: Change all black marbles to white marbles
- Serializability (2PL, OCC)
  - T1 → T2 or T2 → T1
  - In either case, bag is either ALL white or ALL black
- Snapshot isolation (MVCC)
  - T1 → T2 or T2 → T1 or T1 || T2
  - Bag is ALL white, ALL black, or ½ white ½ black

Timestamps in MVCC

- Transactions are assigned timestamps, which may get assigned to objects those txns read/write
- Every object version O_V has both read and write TS
  - ReadTS: Largest timestamp of txn that reads O_V
  - WriteTS: Timestamp of txn that wrote O_V

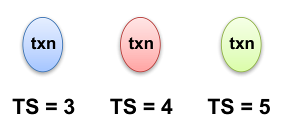
17

Executing transaction T in MVCC

- Find version of object O to read:
  - # Determine the last version written before read snapshot time
  - Find O_V s.t. max { WriteTS(O_V) | WriteTS(O_V) <= TS(T) }
  - ReadTS(O_V) = max(TS(T), ReadTS(O_V))
  - Return O_V to T
- Perform write of object O or abort if conflicting:
  - Find O_V s.t. max { WriteTS(O_V) | WriteTS(O_V) <= TS(T) }
  - # Abort if another T' exists and has read O after T
  - If ReadTS(O_V) > TS(T)
    - Abort and roll-back T
  - Else
    - Create new version O_W
    - Set ReadTS(O_W) = WriteTS(O_W) = TS(T)

18

Digging deeper



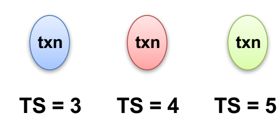
Notation

W(1) = 3: Write creates version 1 with WriteTS = 3

R(1) = 3: Read of version 1 returns timestamp 3

19

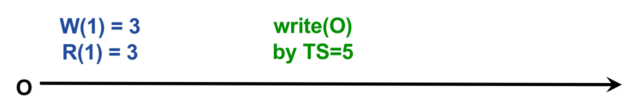
Digging deeper



Notation

W(1) = 3: Write creates version 1 with WriteTS = 3

R(1) = 3: Read of version 1 returns timestamp 3



20

Digging deeper

Notation

W(1) = 3: Write creates version 1 with WriteTS = 3

R(1) = 3: Read of version 1 returns timestamp 3

txn TS = 3 | txn TS = 4 | txn TS = 5

W(1) = 3
R(1) = 3

W(2) = 5
R(2) = 5

O ——————————→

write(O) by TS = 4

Find v such that max WriteTS(v) <= (TS = 4)
⇒ v = 1 has (WriteTS = 3) <= 4
If ReadTS(1) > 4, abort
⇒ 3 > 4: false
Otherwise, write object

21

Digging deeper

Notation

W(1) = 3: Write creates version 1 with WriteTS = 3

R(1) = 3: Read of version 1 returns timestamp 3

txn TS = 3 | txn TS = 4 | txn TS = 5

W(1) = 3
R(1) = 3

W(3) = 4
R(3) = 4

W(2) = 5
R(2) = 5

O ——————————→

Find v such that max WriteTS(v) <= (TS = 4)
⇒ v = 1 has (WriteTS = 3) <= 4
If ReadTS(1) > 4, abort
⇒ 3 > 4: false
Otherwise, write object

22

Digging deeper

Notation

W(1) = 3: Write creates version 1 with WriteTS = 3

R(1) = 3: Read of version 1 returns timestamp 3

txn TS = 3 | txn TS = 4 | txn TS = 5

W(1) = 3
R(1) = 5

O ——————————→

BEGIN Transaction
tmp = READ(O)
WRITE (O, tmp + 1)
END Transaction

Find v such that max WriteTS(v) <= (TS = 5)
⇒ v = 1 has (WriteTS = 3) <= 5
Set R(1) = max(5, R(1)) = 5

23

Digging deeper

Notation

W(1) = 3: Write creates version 1 with WriteTS = 3

R(1) = 3: Read of version 1 returns timestamp 3

txn TS = 3 | txn TS = 4 | txn TS = 5

W(1) = 3
R(1) = 5

W(2) = 5
R(2) = 5

O ——————————→

BEGIN Transaction
tmp = READ(O)
WRITE (O, tmp + 1)
END Transaction

Find v such that max WriteTS(v) <= (TS = 5)
⇒ v = 1 has (WriteTS = 3) <= 5
If ReadTS(1) > 5, abort
⇒ 5 > 5: false
Otherwise, write object

24

Digging deeper

Notation

W(1) = 3: Write creates version 1 with WriteTS = 3

R(1) = 3: Read of version 1 returns timestamp 3

txn TS = 3 | txn TS = 4 | txn TS = 5

W(1) = 3
R(1) = 5

W(2) = 5
R(2) = 5

O

write(O)
by TS = 4

Find v such that max WriteTS(v) <= (TS = 4)
⇒ v = 1 has (WriteTS = 3) <= 4
If ReadTS(1) > 4, abort
⇒ 5 > 4: **true**

25

Digging deeper

Notation

W(1) = 3: Write creates version 1 with WriteTS = 3

R(1) = 3: Read of version 1 returns timestamp 3

txn TS = 3 | txn TS = 4 | txn TS = 5

W(1) = 3
R(1) = 5

W(2) = 5
R(2) = 5

O

BEGIN Transaction
tmp = READ(O)
WRITE (P, tmp + 1)
END Transaction

Find v such that max WriteTS(v) <= (TS = 4)
⇒ v = 1 has (WriteTS = 3) <= 4
Set R(1) = max(4, R(1)) = 5

Then write on P succeeds as well

26

No class Wednesday! 🦃

Monday lecture

Distributed Transactions

+ Google Spanner

27