Database Management Systems

Madhavan Mukund

https://www.cmi.ac.in/~madhavan

Sai University

Lecture 23, 22 November 2023

Concurrency control

- Ensure that only serializable schedules are generated
- Allow concurrency
- Control access to data to avoid conflicts
- Mechanisms
  - Locking
  - Timestamps
  - Multiple versions — snapshot isolation

Concurrency control using locks

- Each data item [redacted] has an associated lock
  - Transaction locks an item before accessing
  - Transaction unlocks the item when done
  - Ensures non-interference

Concurrency control using locks

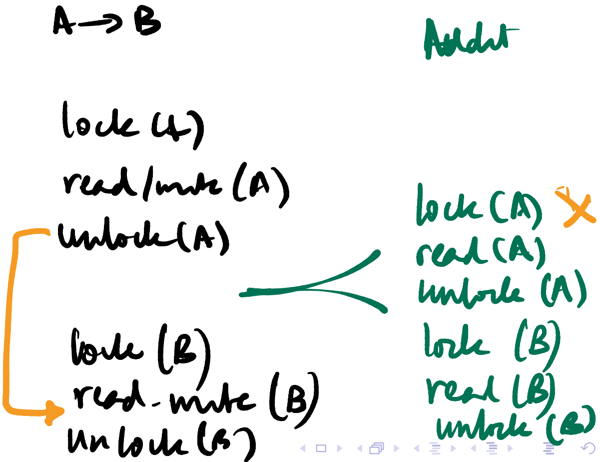
- Each data ~~time~~ item has an associated lock
  - Transaction locks an item before accessing
  - Transaction unlocks the item when done
  - Ensures non-interference
- Shared and exclusive locks
  - To just read a value, use a shared lock — `Lock-S(A)`
  - To write a value, use a exclusive lock — `Lock-X(A)`
  - Multiple transactions can simultaneously hold a shared lock
  - Only one transaction can hold an exclusive lock
  - Upgrade shared lock to exclusive lock, downgrade exclusive lock to shared lock

Concurrency control using locks

- Each data ~~time~~ item has an associated lock
  - Transaction locks an item before accessing
  - Transaction unlocks the item when done
  - Ensures non-interference
- Shared and exclusive locks
  - To just read a value, use a shared lock — `Lock-S(A)`
  - To write a value, use a exclusive lock — `Lock-X(A)`
  - Multiple transactions can simultaneously hold a shared lock
  - Only one transaction can hold an exclusive lock
  - Upgrade shared lock to exclusive lock, downgrade exclusive lock to shared lock
- Lock manager handles lock requests
  - Maintain data structure about items, locks and pending requests
  - Be careful about starvation

T_1 lock-S(A) — unlocks
T_2 lock-X(A) — wait
T_3 lock-S(A) ✓ — unlocks
T_4 lock-S(A) ✓

- Just using locks does not guarantee isolation



Lock protocols

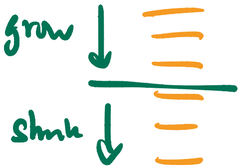
- Just using locks does not guarantee isolation
- Locking protocol — convention for using locks, respected by all transactions

Lock protocols

- Just using locks does not guarantee isolation
- Locking protocol — convention for using locks, respected by all transactions
- Legal schedule — agrees with the locking protocol
  - Goal: Locking protocol that guarantees all legal schedules are conflict serializable

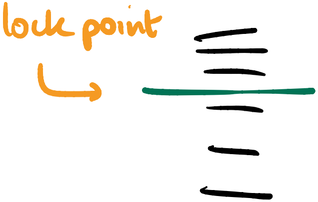
Lock protocols

- Just using locks does not guarantee isolation
- Locking protocol — convention for using locks, respected by all transactions
- Legal schedule — agrees with the locking protocol
  - Goal: Locking protocol that guarantees all legal schedules are conflict serializable
- Two phase locking
  - Growing phase — acquire or upgrade locks
  - Shrinking phase — release or downgrade locks
  - Guarantees conflict serializability



Lock protocols

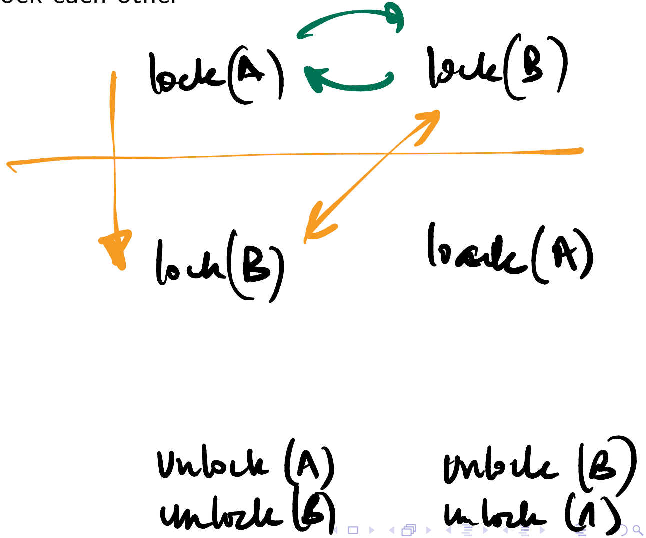
- Just using locks does not guarantee isolation
- Locking protocol — convention for using locks, respected by all transactions
- Legal schedule — agrees with the locking protocol
  - Goal: Locking protocol that guarantees all legal schedules are conflict serializable
- Two phase locking
  - Growing phase — acquire or upgrade locks
  - Shrinking phase — release or downgrade locks
  - Guarantees conflict serializability



- Recovering a serial schedule
  - Lock point for T_i — when T_i completes growing phase
  - Can generate conflict equivalent serial schedule in order of lock points

Deadlocks

- Transactions hold some locks and block each other



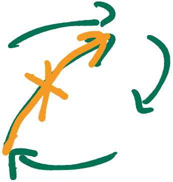
Deadlocks

- Transactions hold some locks and block each other
- Detecting deadlocks — look for cycles in wait-for graph

Deadlocks

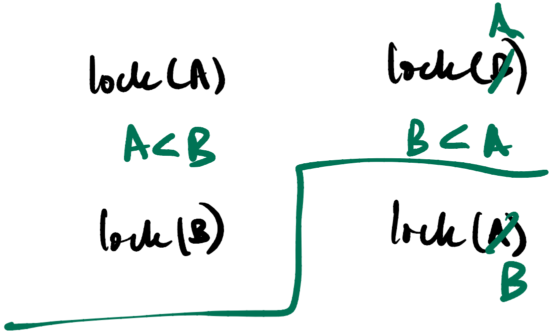
- Transactions hold some locks and block each other
- Detecting deadlocks — look for cycles in wait-for graph
- Resolve deadlocks — kill and rollback some transaction to break the cycle
  - Estimate "cost" of rollback for each transaction
  - Choose the one with minimum cost

"Victim"



Deadlock prevention and pre-emption

- Deadlock prevention
  - Fix an order on all data items, always lock items in that order
  - Example — always lock bank accounts in ascending order of account number



Deadlock prevention and pre-emption

- Deadlock prevention
  - Fix an order on all data items, always lock items in that order
  - Example — always lock bank accounts in ascending order of account number
- Deadlock pre-emption
  - Assign each T_i a timestamp $TS(T_i)$ when it starts
  - If T_i needs a lock held by T_j

sequence number

Deadlock prevention and pre-emption

- Deadlock prevention
  - Fix an order on all data items, always lock items in that order
  - Example — always lock bank accounts in ascending order of account number
- Deadlock pre-emption
  - Assign each T_i a timestamp $TS(T_i)$ when it starts
  - If T_i needs a lock held by T_j
    - Wait-die T_i waits if $TS(T_i) < TS(T_j)$, else T_i rolls back — Retain timestamp

$T_i \rightarrow T_j$

Later T_j asks for T_i lock

Deadlock prevention and pre-emption

- Deadlock prevention
  - Fix an order on all data items, always lock items in that order
  - Example — always lock bank accounts in ascending order of account number
- Deadlock pre-emption
  - Assign each T_i a timestamp $TS(T_i)$ when it starts
  - If T_i needs a lock held by T_j
    - Wait-die T_i waits if $TS(T_i) < TS(T_j)$, else T_i rolls back
    - Wound-wait T_i waits if $TS(T_i) > TS(T_j)$, else T_j rolls back

"wound" T_j

Deadlock prevention and pre-emption

- Deadlock prevention
  - Fix an order on all data items, always lock items in that order
  - Example — always lock bank accounts in ascending order of account number
- Deadlock pre-emption
  - Assign each T_i a timestamp $TS(T_i)$ when it starts
  - If T_i needs a lock held by T_j
    - Wait-die T_i waits if $TS(T_i) < TS(T_j)$, else T_i rolls back
    - Wound-wait T_i waits if $TS(T_i) > TS(T_j)$, else T_j rolls back
- Lock timeout
  - Transaction rolls itself back if request for lock times out
  - How to fix time out period to prevent unnecessary rollbacks?

Concurrency control using timestamps

- Fix a serializability order in advance
  - Assign a starting timestamp $TS(T_i)$ to each T_i that respects this order

No locks!

Concurrency control using timestamps

- Fix a serializability order in advance
  - Assign a starting timestamp $TS(T_i$ to each T_i that respects this order
- Record read and write timestamps for each item
  - $R{-}timestamp(A)$ — largest timestamp of transactions to successfully read A
  - $W{-}timestamp(A)$ — largest timestamp of transactions to successfully write A

Concurrency control using timestamps

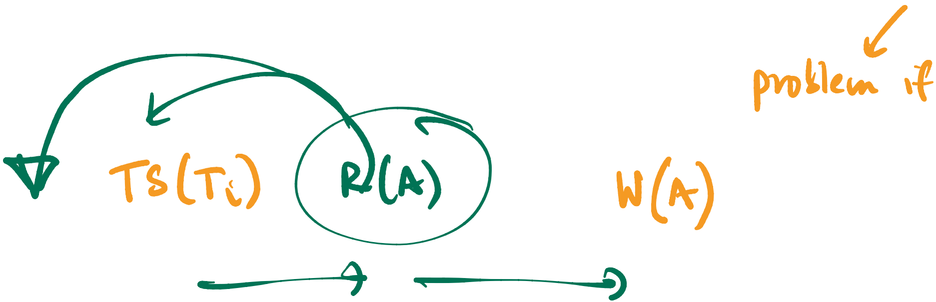
- Fix a serializability order in advance
  - Assign a starting timestamp $TS(T_i$ to each T_i that respects this order
- Record read and write timestamps for each item
  - $R{-}timestamp(A)$ — largest timestamp of transactions to successfully read A
  - $W{-}timestamp(A)$ — largest timestamp of transactions to successfully write A
- Reading: T_i attempts to read A
  - $TS(T_i) < W{-}timestamp(A)$ — need an older value, reject and rollback TS_i New Timest
  - $TS(T_i) \geq W{-}timestamp(A)$ — read succeeds, update $R{-}timestamp(A)$ if needed
    max(current, T_i)

Concurrency control using timestamps

- Fix a serializability order in advance
  - Assign a starting timestamp $TS(T_i$ to each T_i that respects this order
- Record read and write timestamps for each item
  - $R-timestamp(A)$ — largest timestamp of transactions to successfully read A
  - $W-timestamp(A)$ — largest timestamp of transactions to successfully write A
- Reading: T_i attempts to read A
  - $TS(T_i < W-timestamp(A)$ — need an older value, reject and rollback TS_i
  - $TS(T_i \geq W-timestamp(A)$ — read succeeds, update $R-timestamp(A)$ if needed
- Writing: T_i attempts to write A
  - $TS(T_i < R-timestamp(A)$ — current value was needed earlier value, reject and rollback TS_i
  - $TS(T_i < W-timestamp(A)$ — writing an obsolete value, reject and rollback TS_i
  - Otherwise — read succeeds, update $W-timestamp(A)$ to $TS(T_i)$

Concurrency control using timestamps

- Writing: T_i attempts to write A
  - $TS(T_i) < R\text{–}timestamp(A)$ — current value was needed earlier value, reject and rollback TS_i
  - $TS(T_i) < W\text{–}timestamp(A)$ — writing an obsolete value, reject and rollback TS_i
  - Otherwise — read succeeds, update $W\text{–}timestamp(A)$ to $TS(T_i)$



Concurrency control using timestamps

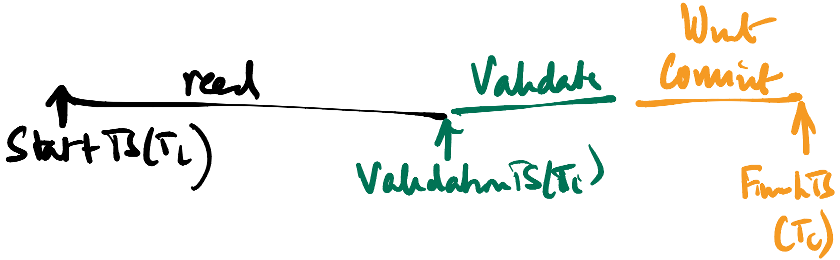
- Writing: T_i attempts to write A
  - $TS(T_i < R{-}timestamp(A)$ — current value was needed earlier value, reject and rollback TS_i
  - $TS(T_i < W{-}timestamp(A)$ — writing an obsolete value, reject and rollback TS_i
  - Otherwise — read succeeds, update $W{-}timestamp(A)$ to $TS(T_i)$
- Thomas's Write Rule
  - Allow harmless obsolete writes
  - $TS(T_i < R{-}timestamp(A)$ — current value was needed earlier value, reject and rollback TS_i
  - $TS(T_i < W{-}timestamp(A)$ — writing an obsolete value, ignore this write and proceed
  - Otherwise — read succeeds, update $W{-}timestamp(A)$ to $TS(T_i)$

"View Serializability"

└ Produces an equivalent schedule but not conflict serializable

Validation based protocols

- Predict serial order as before — each transaction T_i is assigned a timestamp $TS(T_i)$
- Transactions execute in three phases, maintain three time stamps
  - Read phase — $StartTS(T_i$ is start of read phase
  - Validation phase — $ValidationTS(T_i$ is start of validation phase
  - Write phase — $FinishTS(T_i$ is end of write phase



Validation based protocols

- Predict serial order as before — each transaction T_i is assigned a timestamp $TS(T_i)$
- Transactions execute in three phases, maintain three time stamps
  - Read phase — $StartTS(T_i$ is start of read phase
  - Validation phase — $ValidationTS(T_i$ is start of validation phase
  - Write phase — $FinishTS(T_i$ is end of write phase
- TS_i Write values back to database only if validation phase succeeds
  For each transaction T_k with $TS(T_k) < TS(T_i)$,
  - $FinishTS(T_k) < StartTS(T_i)$, or
  - Data written by T_k disjoint from data read by T_i and $StartTS(T_i) < FinishTS(T_k) < ValidationTS(T_i)$

— no inconsistent reads
— no overwriting

Validation based protocols

- Predict serial order as before — each transaction T_i is assigned a timestamp $TS(T_i)$
- Transactions execute in three phases, maintain three time stamps
  - Read phase — $StartTS(T_i$ is start of read phase
  - Validation phase — $ValidationTS(T_i$ is start of validation phase
  - Write phase — $FinishTS(T_i$ is end of write phase
- TS_i Write values back to database only if validation phase succeeds
  For each transaction T_k with $TS(T_k) < TS(T_i)$,
  - $FinishTS(T_k) < StartTS(T_i)$, or
  - Data written by T_k disjoint from data read by T_i and $StartTS(T_i) < FinishTS(T_k) < ValidationTS(T_i)$
- Optimistic concurrency control

Multi-version concurrency control

Keep multiple timestamped versions of each item

$A \rightarrow \langle A_1, A_2, \ldots, A_n \rangle$

$TS(T_i)$

$T_i \rightarrow read(A)$

Beyond RDBMS

Tables = Structured Data

- Semi-structured data
  - Notation

Tag - Value

XML

NoSQL databases

<book>
<title> -- </title>
<author> -- </author>
<author> -- </author>
</book>

Beyond RDBMS

- Semi-structured data
- CAP theorem

Distributed Data

Replicated across servers

Consistency
Availability
Partition Tolerance

Not simultaneously possible, in general

P unavoidable

C? A? ✓

traditional DBMS keeps C

Business on internet

- Semi-structured data
- CAP theorem
- Weak consistency

"Eventual" consistency

- Servers will reconcile all changes consistently

Weak consistency example