RDBMS and SQL

Madhavan Mukund

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

Lecture 12, 20 November 2025

Transactions: desirable properties

- Atomicity — All or nothing
- Consistency — Maintain properties
- Isolation
- Durability — Persistence
- ACID properties

Bank, internal transfers

Transfer 1

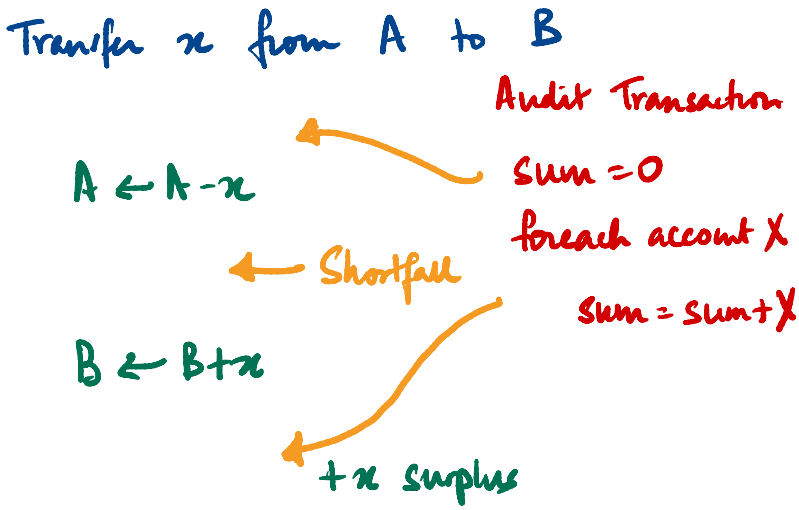
⋮

Transfer n

No change in total amount across accounts

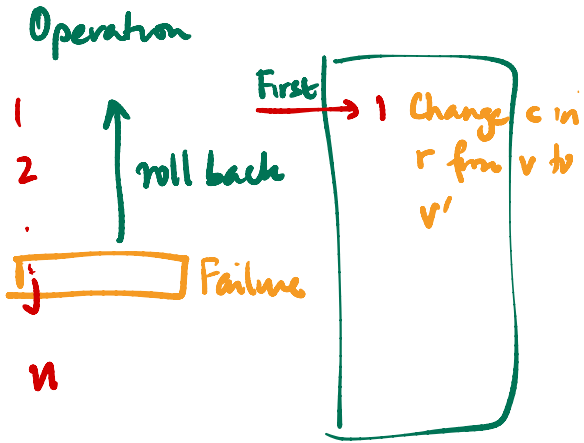
Transactions: desirable properties

- Atomicity
- Consistency
- Isolation
- Durability
- ACID properties



Transaction logs

- Log each update before it happens
- Rollback updates in case of failure



Concurrent execution and schedules

T_1: read(A);
$A := A - 50$;
write(A);
read(B);
$B := B + 50$;
write(B).

T_2: read(A);
$temp := A * 0.1$;
$A := A - temp$;
write(A);
read(B);
$B := B + temp$;
write(B).

Concurrent execution and schedules

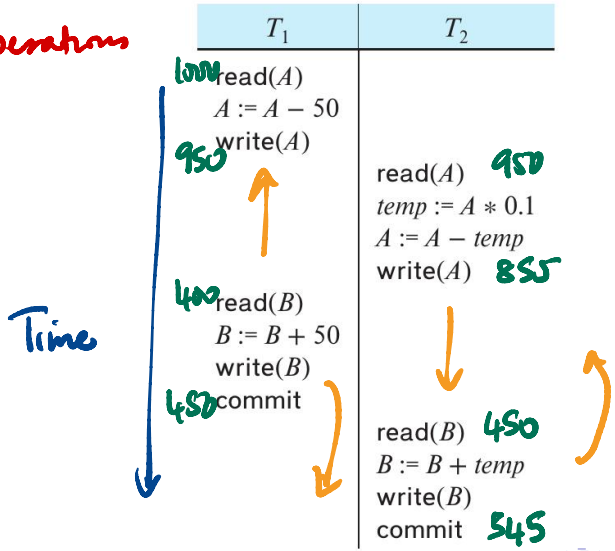
T_1	T_2
read(A)	
$A := A - 50$	
write(A)	
read(B)	
$B := B + 50$	
write(B)	
commit	
	read(A)
	$temp := A * 0.1$
	$A := A - temp$
	write(A)
	read(B)
	$B := B + temp$
	write(B)
	commit

Concurrent execution and schedules

T_1	T_2
	read(A)
	$temp := A * 0.1$
	$A := A - temp$
	write(A)
	read(B)
	$B := B + temp$
	write(B)
	commit
read(A)	
$A := A - 50$	
write(A)	
read(B)	
$B := B + 50$	
write(B)	
commit	

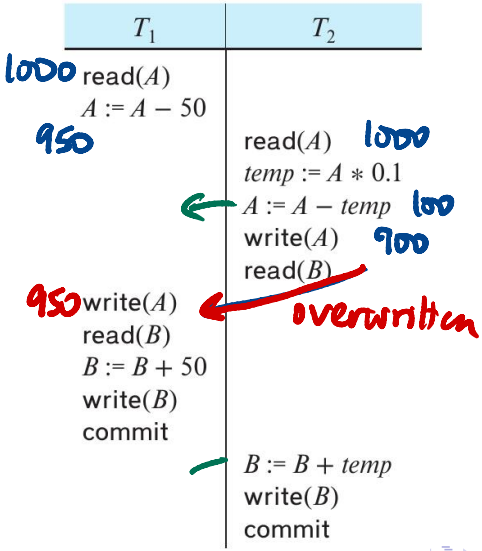
Concurrent execution and schedules

Interleaved order of operations across transaction



Concurrent execution and schedules

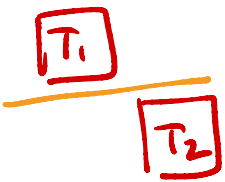
Some schedules are not "valid"

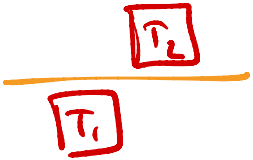


Serializability

- Serial schedule — each transaction executes as a block, no interleaving

Guaranteed to be "ok"



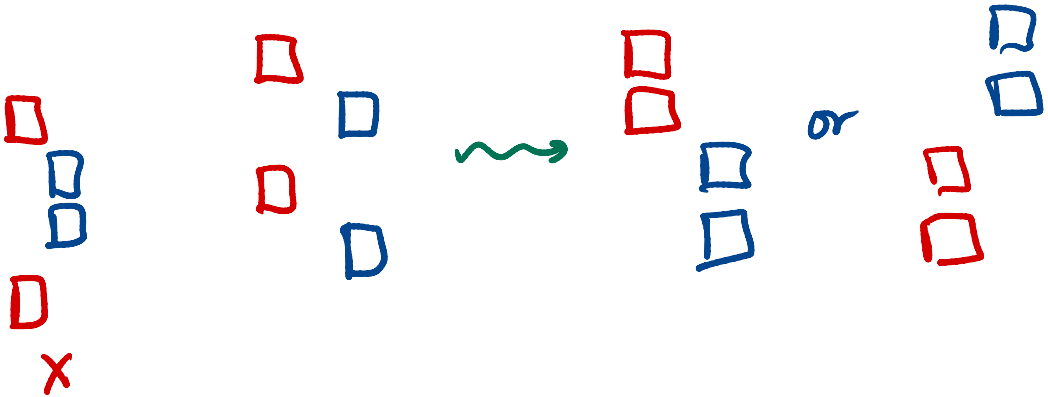


T_1
T_2

T_2
T_1

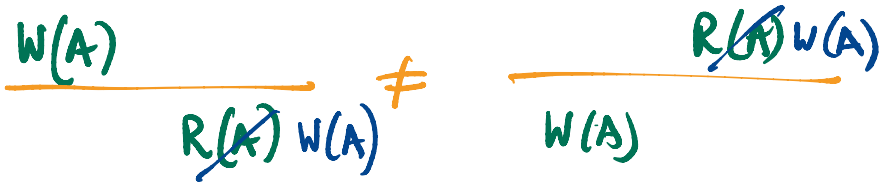
Serializability

- Serial schedule — each transaction executes as a block, no interleaving
- Serializable schedule — equivalent to *some* serial schedule



Serializability

- Serial schedule — each transaction executes as a block, no interleaving
- Serializable schedule — equivalent to *some* serial schedule
- Conflicting operations — two operations on the *same* value where *at least one is a write* — CANNOT swap



Serializability

- Serial schedule — each transaction executes as a block, no interleaving
- Serializable schedule — equivalent to *some* serial schedule
- Conflicting operations — two operations on the *same* value where *at least one is a write*
- Conflict equivalence — one schedule can be transformed into the other by reordering non-conflicting operations (adjacent)

Serializability

- Serial schedule — each transaction executes as a block, no interleaving
- Serializable schedule — equivalent to *some* serial schedule
- Conflicting operations — two operations on the *same* value where *at least one is a write*
- Conflict equivalence — one schedule can be transformed into the other by reordering non-conflicting operations
- Conflict serializable — can be reordered to a conflict-equivalent serial schedule

Conflict equivalence

Serial

T_1	T_2
read(A)	
$A := A - 50$	
write(A)	
read(B)	
$B := B + 50$	
write(B)	
commit	
	read(A)
	$temp := A * 0.1$
	$A := A - temp$
	write(A)
	read(B)
	$B := B + temp$
	write(B)
	commit

Signal successful update

Conflict serializable

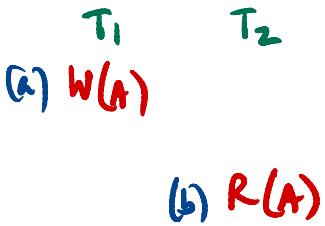
T_1	T_2
read(A)	
$A := A - 50$	
write(A)	
	read(A)
	$temp := A * 0.1$
	$A := A - temp$
	write(A)
read(B)	
$B := B + 50$	
write(B)	
commit	
	read(B)
	$B := B + temp$
	write(B)
	commit

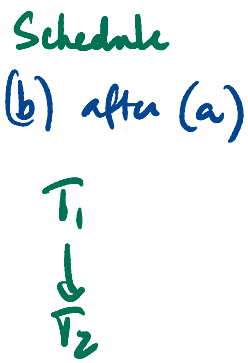
Testing for conflict serializability

- Start with a schedule — interleaved sequence of operations from multiple transactions

Testing for conflict serializability

- Start with a schedule — interleaved sequence of operations from multiple transactions
- Build a graph, with transactions as nodes



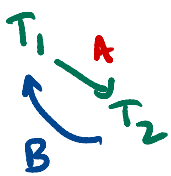


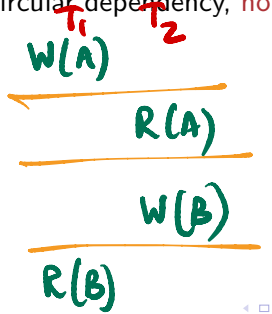
Testing for conflict serializability

- Start with a schedule — interleaved sequence of operations from multiple transactions
- Build a graph, with transactions as nodes
- Edge $T_i \rightarrow T_j$ if an earlier operation in T_i conflicts with a later operation in T_j

Testing for conflict serializability

- Start with a schedule — interleaved sequence of operations from multiple transactions
- Build a graph, with transactions as nodes
- Edge $T_i \rightarrow T_j$ if an earlier operation in T_i conflicts with a later operation in T_j
- If this conflict graph has cycles, there is a circular dependency, not conflict serializable





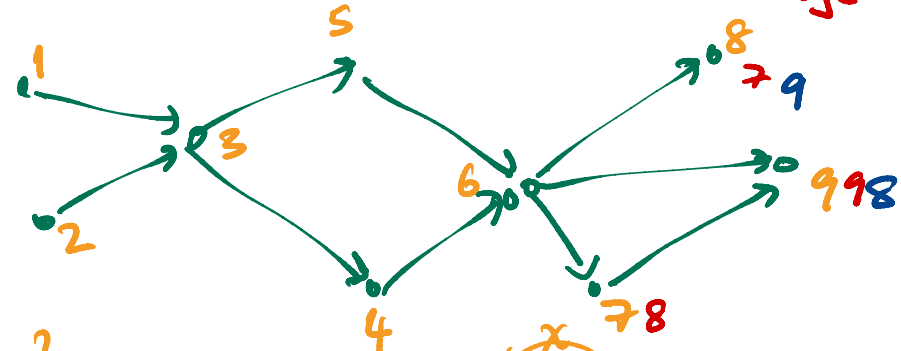
Testing for conflict serializability

- Start with a schedule — interleaved sequence of operations from multiple transactions
- Build a graph, with transactions as nodes
- Edge $T_i \rightarrow T_j$ if an earlier operation in T_i conflicts with a later operation in T_j
- If this conflict graph has cycles, there is a circular dependency, not conflict serializable
- If the conflict graph is acyclic, use topological sort to order the transactions into a serial schedule.

Directed Acyclic Graph

n nodes - must be a node with no incoming edge

Topological Sort



Transactions in SQL

- `START TRANSACTION`, `COMMIT`, `ROLLBACK`

Transactions in SQL

- `START TRANSACTION`, `COMMIT`, `ROLLBACK`
- Isolation levels

Strict non interference

Transactions in SQL

- `START TRANSACTION`, `COMMIT`, `ROLLBACK`
- Isolation levels
  - Serializable

No interference

Transactions in SQL

- `START TRANSACTION`, `COMMIT`, `ROLLBACK`
- Isolation levels
  - Serializable
  - Repeatable read

Read from another transaction

But no further change

Transactions in SQL

- `START TRANSACTION`, `COMMIT`, `ROLLBACK`
- Isolation levels
  - Serializable
  - Repeatable read
  - Read committed

Transactions in SQL

- `START TRANSACTION`, `COMMIT`, `ROLLBACK`
- Isolation levels
  - Serializable
  - Repeatable read
  - Read committed
  - Read uncommitted

Transactions in SQL

- `START TRANSACTION`, `COMMIT`, `ROLLBACK`
- Isolation levels
  - Serializable
  - Repeatable read
  - Read committed
  - Read uncommitted
  - `SET TRANSACTION ISOLATION LEVEL READ COMMITTED`

Concurrency control

- Ensure that only serializable schedules are generated
- Allow concurrency
- Control access to data to avoid conflicts

Concurrency control using locks

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

Concurrency control using locks

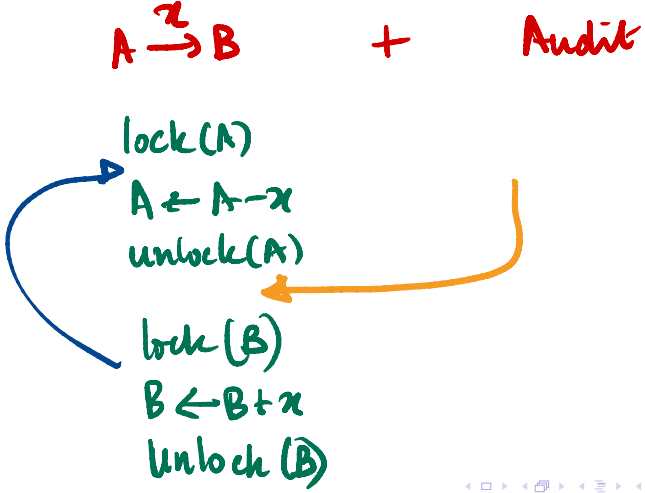
- Each data item has an associated lock
  - Transaction locks an item before accessing
  - Transaction unlocks the item when done
  - Ensures non-interference
- Shared and exclusive locks
  - To just read a value, use a shared lock — `Lock-S(A)`
    - Multiple transactions can simultaneously hold a shared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

Concurrency control using locks

- Each data 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)`
    - Multiple transactions can simultaneously hold a shared lock
  - To write a value, use a exclusive lock — `Lock-X(A)`
    - Only one transaction can hold an exclusive lock
  - Can 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 — fairness, starvation

Lock protocols

- 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

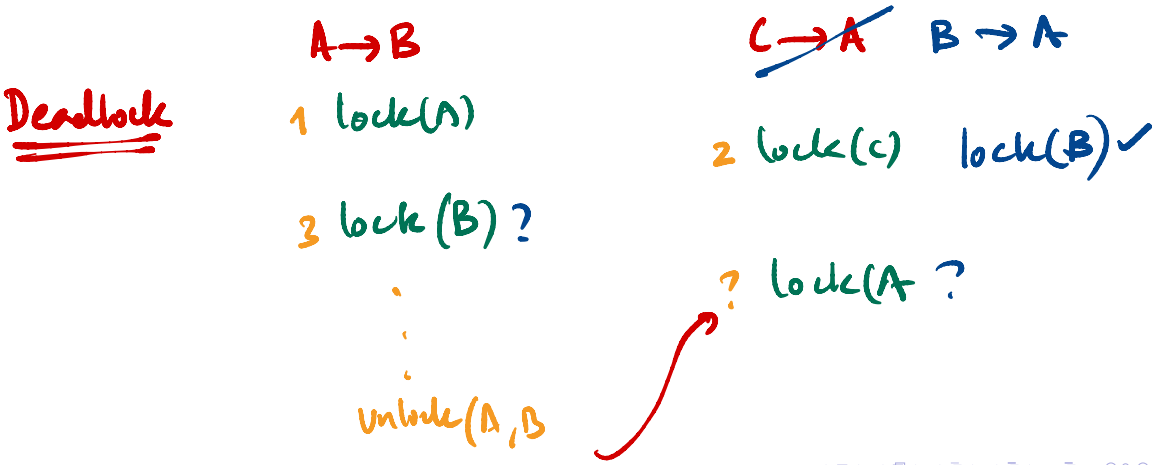
- Just using locks does not guarantee isolation
- Locking protocol — convention for using locks, respected by all transactions
- Legal schedule — agrees with the locking protocol
  - Goal: Locking protocol that guarantees all legal schedules are conflict serializable

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

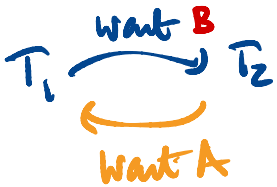
Deadlocks

- 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



Kill a transaction

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

Deadlock prevention and pre-emption

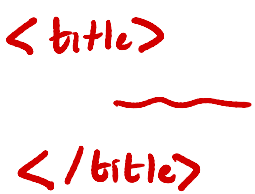
- Deadlock prevention
  - Fix an order on all data items, always lock items in that order
  - Example — always lock bank accounts in ascending order of account number

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, decide whether to wait or roll back one of them, based on $TS(T_i)$ and $TS(T_j)$

Beyond RDBMS

- Semi-structured data



XML

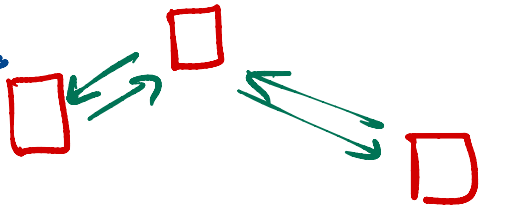
Beyond RDBMS

- Semi-structured data
- CAP theorem

Partition Tolerance ?

Available

Consistency



Shared distributed data

Beyond RDBMS

- Semi-structured data
- CAP theorem
- Weak consistency

Weak consistency example