RDBMS and SQL

Madhavan Mukund

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

Lecture 12, 7 November 2024

Transactions

Transfer money

Debit A

Credit B

Ticket booking

Availability

Payment details

Passenger details

OTP fails

All or nothing

What does this mean?

Desirable properties

- Atomicity — All or nothing wrt possible interruptions / failures

OTP fails

Not enough balance

Intermediate state never visible

Desirable properties

- Atomicity
- Consistency

Internal

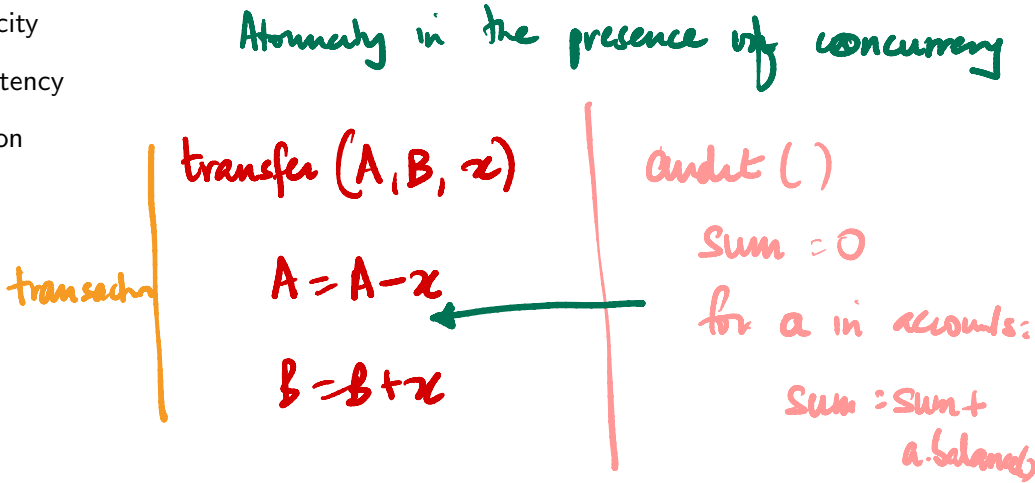
Keys – uniqueness, foreign keys, null values

Invariants

Total money in all accounts

Desirable properties

- Atomicity
- Consistency
- Isolation



Desirable properties

- Atomicity
- Consistency
- Isolation
- Durability – Persistence of update

Desirable properties

- Atomicity
- Consistency
- Isolation
- Durability
- ACID properties

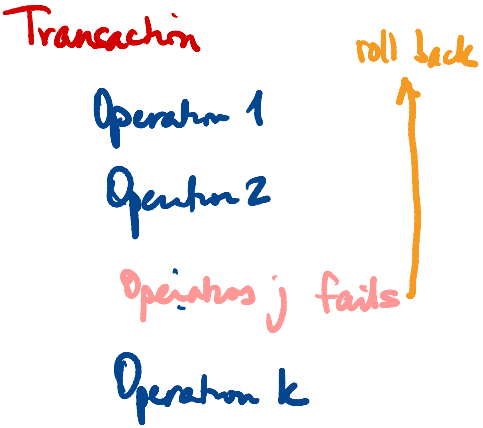
What is the issue?

- Failures
- Efficiency, concurrent transactions

Transaction logs

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

"Optimistic" execution



Concurrent execution and schedules

Scheduler

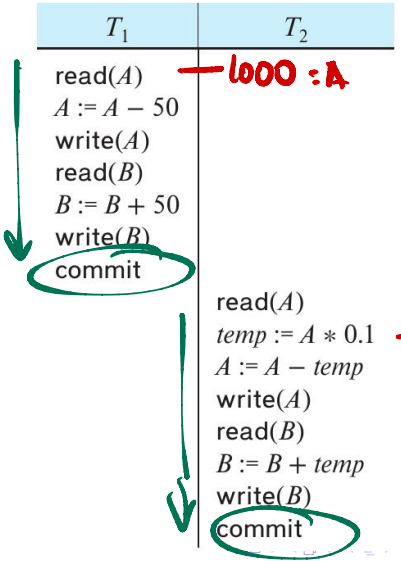
T_1: read(A);
$A := A - 50$;
write(A);
read(B);
$B := B + 50$;
write(B).

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

Concurrent execution and schedules

Commit

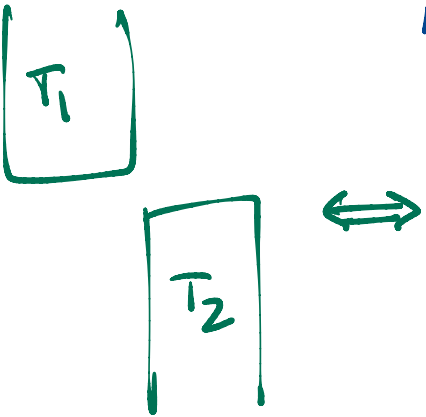
└ signals successful end of transaction



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	

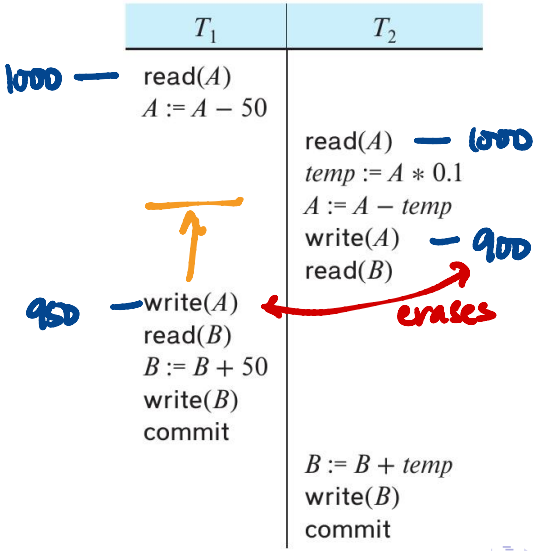
Concurrent execution and schedules



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

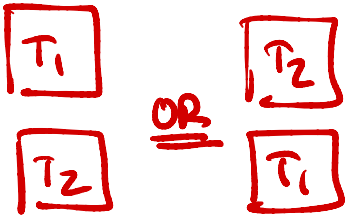
Concurrent execution and schedules

What constitutes a "good" schedule?



Serializability

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

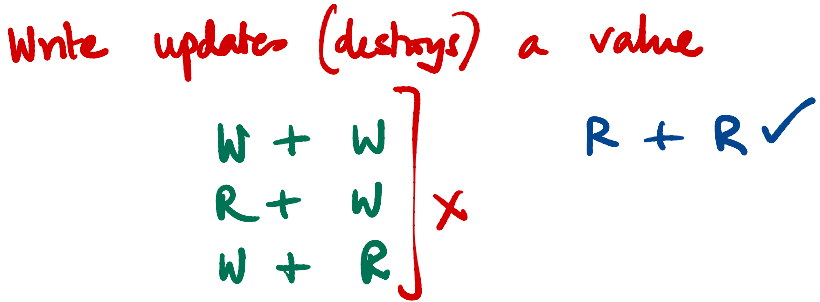


Serializability

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

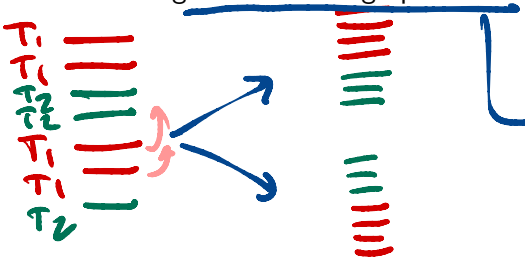
Serializability

- 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*



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



equivalence — reorder

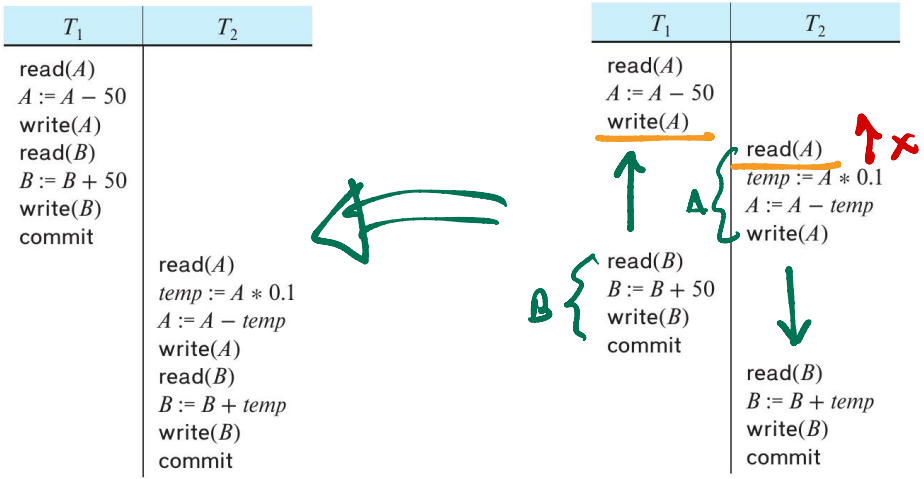
R+R same variable

Any ops on different variable

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



Conflict equivalence

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

Serializable
T_1 then T_2

NOT
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

read(B)

Testing for conflict serializability

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

Testing for conflict serializability

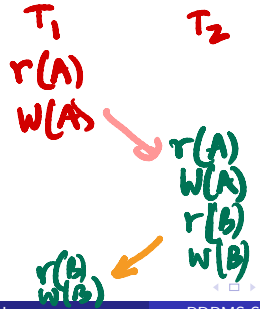
- Start with a schedule — interleaved sequence of operations from multiple transactions
- Build a graph, with transactions as nodes

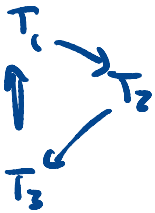
Testing for conflict serializability

- 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



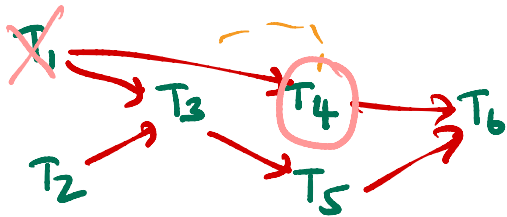


Testing for conflict serializability

- 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

Must be a T_i without an incoming edge



T_2 T_1

T_1 T_2 T_3 T_4

T_1 T_4 T_2 T_3

Transactions in SQL

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

Transactions in SQL

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

Transactions in SQL

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

Transactions in SQL

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

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

With locks

→ Ensure non conflicting access