Database Management Systems

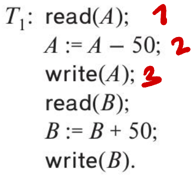
Madhavan Mukund

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

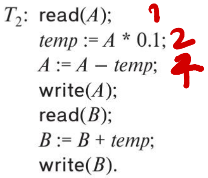
Sai University

Lecture 21, 10 November 2023

Concurrent execution and schedules



```
T1: read(A);          1
    A := A – 50;      2
    write(A);         3
    read(B);
    B := B + 50;
    write(B).
```



```
T2: read(A);              1
    temp := A * 0.1;      2
    A := A – temp;        4
    write(A);
    read(B);
    B := B + temp;
    write(B).
```

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).
```

1000
2000

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

950
2050

95

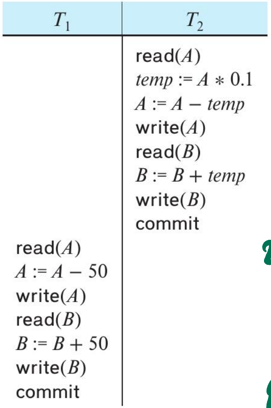
855
2145

Serial schedule 1

Concurrent execution and schedules

```
T1: read(A);
    A := A − 50;
    write(A);
    read(B);
    B := B + 50;
    write(B).
```

```
T2: read(A);
    temp := A * 0.1;
    A := A − temp;
    write(A);
    read(B);
    B := B + temp;
    write(B).
```



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	

1000
2000

900
2100

850
2150

Serial schedule 2

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).

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

Serial schedule 1

reorder
oper

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

B

A

Consistent concurrent schedule

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).

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

read
1000

Serial schedule 1

T_1	T_2
read(A)	
$A := A - 50$	
	read(A)
	$temp := A * 0.1$
	$A := A - temp$
	write(A) - 900
	read(B)
950 write(A)	
read(B)	
$B := B + 50$	
write(B)	
commit	
	$B := B + temp$
	write(B)
	commit

Inconsistent concurrent schedule

Serializability

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

Isolation

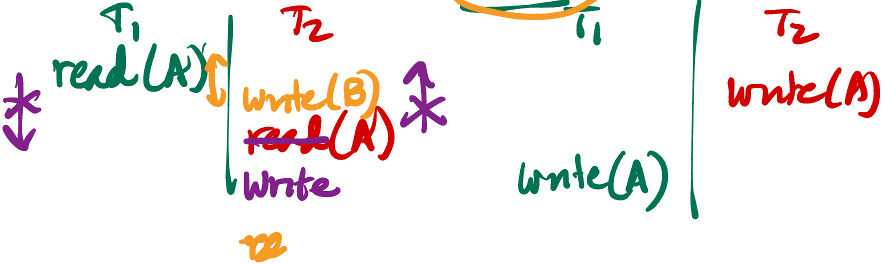
This happened because T_2 was in parallel

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

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

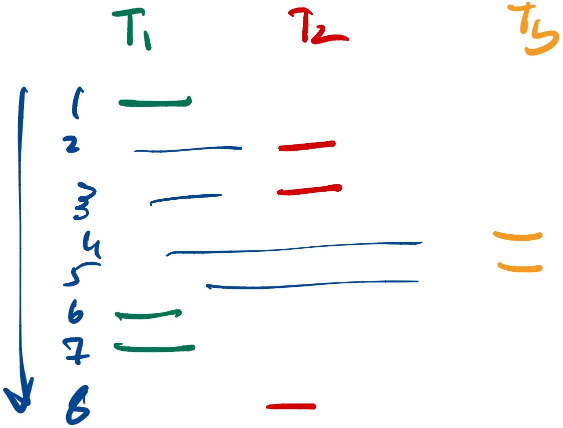
Conflict serializable → serializable

sufficient ↚

but not necessary

Testing for conflict serializability

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

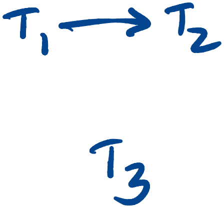


Is this conflict serializable?

Testing for conflict serializability

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

- Build a graph, with transactions as nodes



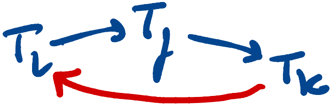
Testing for conflict serializability

- 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

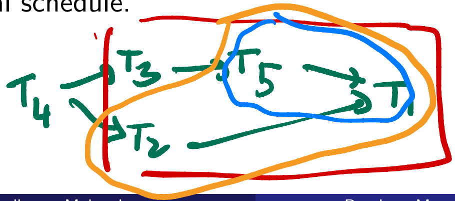
Detecting cycles in a directed graph



— If serializable ⇒ witness

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.



T_4 T_3 T_5 T_2 T_1

T_4 T_3 T_2 T_5 T_1

T_4 T_2 T_3 T_5 T_1

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.

Given a schedule — detect (conflict) serializability
and generate a witness

If not? Avoid non serializable schedule?

Recoverable schedules

Aborting a transaction cannot be ruled out

T_6	T_7
read(A)	
write(A)	
	read(A)
	commit
read(B)	

Ignoring in between computations

aborts

over Durable

Non recoverable

Cascading rollbacks

T_6 aborts → T_9 aborts

↓

T_{10} aborts

T_8	T_9	T_{10}
read(A)		
read(B)		
write(A)		
	read(A)	
	write(A)	
		read(A)
abort		

Not committed

Cascadeless schedules

- If T_j reads data written by T_i, T_i commits before the read of T_j

Uncommitted writes

"Dirty writes"

Transactions in SQL

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

starts

commit

undo current transactn

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
  - Read committed — No dirty writes are read

Transactions in SQL

- `START TRANSACTION`, `COMMIT`, `ROLLBACK`
- Isolation levels
  - Serializable
  - Read committed
  - Read uncommitted — Very flexible

E commerce

Browsing data — "weakly consistent"

Purchasing — strongly consistent

Transactions in SQL

- `START TRANSACTION`, `COMMIT`, `ROLLBACK`
- Isolation levels
  - Serializable
  - Read committed
  - Read uncommitted
  - Repeatable read ~ Read same variable — same value

Transactions in SQL

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

Update / change isolation

Concurrency control

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

How does T_2 know that read(A) is reading a dirty write?

Trivial solution

Queue up all pending transactions

Execute serially

Concurrency control

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

Concurrency control

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

Before accessing an item, "lock" it

Finish → unlock it

"Locking protocols"

Concurrency control

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

T_2 then T_3 then T_1

Decide the serial order in advance

- Assign each transaction a time-stamp

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