COMP207
Database Development

Lecture 5

Transaction Management:
Which Schedules are “Good”?

Review

- Previous lecture:
  - Exemplified ACID properties
  - Started to talk about concurrency control
    - Schedules
    - Serial & concurrent schedules
- What is a schedule? When is it serial/concurrent?

Schedules

- Schedule:
  - holds operations of one or several transactions for execution
  - operations of individual transactions occur in the order in which they occur in the transaction
- Example:

Transaction T_1
read_item(X)
X := X + N
write_item(X)
read_item(Y)
Y := Y + N
write_item(Y)
commit

Transaction T_2
read_item(X)
X := X + M
write_item(X)
commit

→

Schedule for T_1 and T_2	
read_item(X)	
	read_item(X)
	X := X + M
X := X + N	
write_item(X)	
read_item(Y)	
	write_item(X)
	commit
Y := Y + N	
write_item(Y)	
commit	

Is this schedule serial?

No (but concurrent)

Serial Schedules

- Operations of transaction 1 come first, then operations of transaction 2, then operations of transaction 3, …
- No interleaving of operations

Transaction T_1
read_item(X)
X := X + N
write_item(X)
read_item(Y)
Y := Y + N
write_item(Y)
commit

Transaction T_2
read_item(X)
X := X + M
write_item(X)
commit

→

Serial schedule for T_1 and T_2	
read_item(X)	
X := X + N	
write_item(X)	
read_item(Y)	
Y := Y + N	
write_item(Y)	
commit	
	read_item(X)
	X := X + M
	write_item(X)
	commit

- Always guarantee consistency & isolation

Reminder

(see Lecture 3)

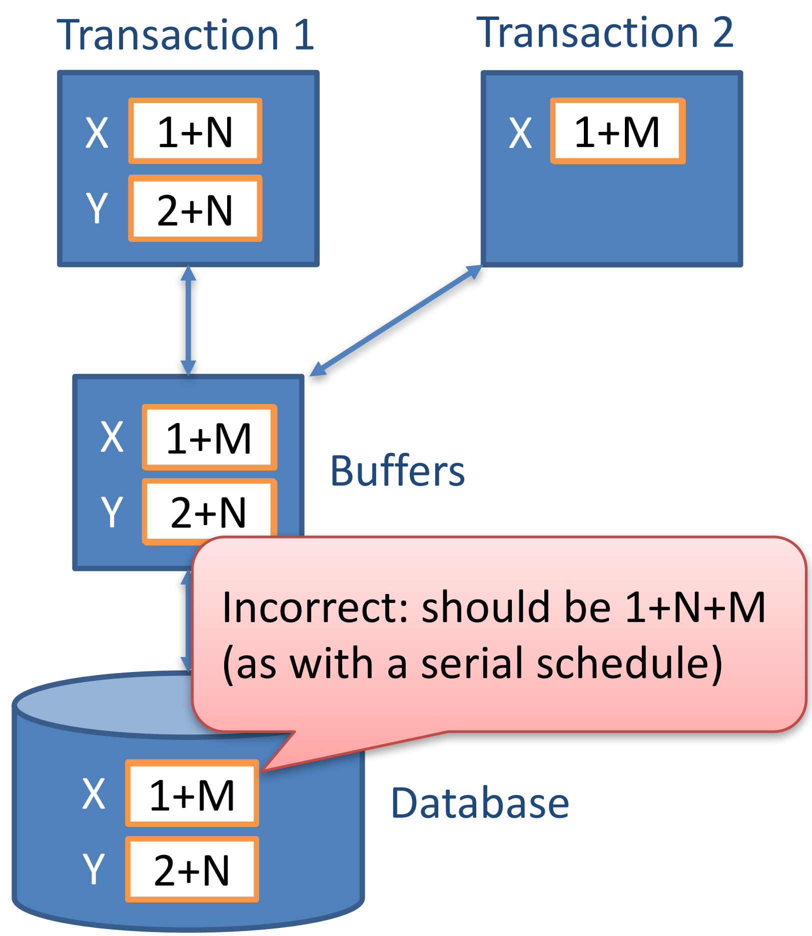
Transactions *always* transform a database from one consistent state to another consistent state.

Concurrent Schedules Problems

- **Lost updates**:
  - The problem demonstrated at the end of the last lecture (see next slide).
  - A successful transaction can be overwritten by another transaction.
- **Dirty reads**: a transaction can see the intermediate results of another transaction before it commits
- **Inconsistent analysis**: a transaction reads several values but another transaction updates some of them in the meantime
- Other problems: see tutorial slides

Concurrent Schedules Do Not Guarantee Consistency

Time	Schedule	
t0		
t1	read_item(X)	
t2		read_item(X)
t3		X := X + M
t4	X := X + N	
t5	write_item(X)	
t6	read_item(Y)	
t7		write_item(X)
t8		commit
t9	Y := Y + N	
t10	write_item(Y)	
t11	commit	



Schedules – Two Extremes

- **Serial Schedules**
  - execute correctly
  - maintain consistency of the database
  - *inefficient* in multi-user environments

- **Concurrent Schedules** ('non-serial')
  - may not execute correctly
  - may not guarantee consistency of the database or isolation
  - *efficient* in multi-user environments

What makes a schedule a “good” schedule?

(one that is *efficient* and guarantees *consistency/isolation*)

Serialisable Schedules

- A schedule S is **serialisable** if there is a serial schedule S’ that has the same effect as S on every initial database state.

Schedule S	
read_item(X)	
X := X + N	
write_item(X)	
	read_item(X)
	X := X + M
	write_item(X)
	commit
read_item(Y)	
Y := Y + N	
write_item(Y)	
commit	

Equivalent serial schedule S’	
read_item(X)	
X := X + N	
write_item(X)	
read_item(Y)	
Y := Y + N	
write_item(Y)	
commit	
	read_item(X)
	X := X + M
	write_item(X)
	commit

So, S is serialisable

Serialisable Schedules

- A schedule S is **serialisable** if there is a serial schedule S' that has the same effect as S on every initial database state.

Schedule S	
read_item(X)	
	read_item(X)
	X := X + M
X := X + N	
write_item(X)	
read_item(Y)	
	write_item(X)
	commit
Y := Y + N	
write_item(Y)	
commit	

In general not serialisable

However, serialisable if N = 0 (do you see why?)

Serialisable Schedules

- A schedule S is **serialisable** if there is a serial schedule S’ that has the same effect as S on every initial database state.

Schedule S	
read_item(X)	
	read_item(X)
	X := X + M
X := X + N	
write_item(X)	
read_item(Y)	
	write_item(X)
	commit
Y := Y + N	
write_item(Y)	
commit	

$\xrightarrow{N = 0}$

Equivalent to S (if N = 0)	
read_item(X)	
X := X + N	
write_item(X)	
read_item(Y)	
Y := Y + N	
write_item(Y)	
commit	
	read_item(X)
	X := X + M
	write_item(X)
	commit

Serialisable Schedules

- Serialisable schedules are essentially those schedules that we are looking for.
- Guarantee:
  - Correctness & consistency (because serial schedules do) ✓
  - No isolation (but could be added later…)
- Problem: serialisability is difficult to test
  - Does not only depend on reads, writes, and commits, but also on the non-database operations
  - Non-database operations can be complex

Assumption from now on: serialisability only depends on read and write operations (still difficult to test)

Conflict-serialisability

Conflict-Serialisability

- Stronger form of serialisability that is used/enforced by most commercial DBMS
- Based on the notion of a **conflict**.

> A **conflict** in a schedule is a pair of operations from different transactions *that cannot be swapped* without changing the behaviour of at least one of the transactions.

Do not write commit/abort (will deal with these later)

- Example: $r_1(X); \underline{w_1(X)}; \underline{r_2(X)}; w_2(X); r_1(Y); w_1(Y)$

conflict

Observation: Operations can only be in conflict if one of them is a write & they access the same item.

Conflict – Characterisation

- A **conflict** in a schedule is a pair of operations from different transactions such that:
  - the operations access the same item
  - at least one of them is a write operation
- Example (from previous slide):

$$S: r_1(X);\ \underline{w_1(X)};\ \underline{r_2(X)};\ w_2(X);\ r_1(Y);\ w_1(Y)$$

conflict in S

- Can you find any other conflicts?

Conflict-Serialisability

- Two schedules S and S' are **conflict-equivalent** if S' can be obtained from S by swapping any number of *consecutive* non-conflicting operations from different transactions.

!

- Example:

S: $r_1(X); w_1(X); r_2(X); w_2(X); r_1(Y); w_1(Y)$

Conflict-serialisable schedule

$r_1(X); w_1(X); r_2(X); r_1(Y); w_2(X); w_1(Y)$

$r_1(X); w_1(X); r_1(Y); r_2(X); w_2(X); w_1(Y)$

$r_1(X); w_1(X); r_1(Y); r_2(X); w_1(Y); w_2(X)$

S': $r_1(X); w_1(X); r_1(Y); w_1(Y); r_2(X); w_2(X)$

Serial schedule

- A schedule is **conflict-serialisable** if it is conflict-equivalent to a serial schedule.

Example

Is the schedule S below conflict-serialisable?

S: $r_1(X); w_1(X); r_2(X); w_2(X); r_1(Y); w_1(Y); r_2(Y); w_2(Y)$

Example

Is the schedule S below conflict-serialisable?

Yes: Conflict-serialisable

S: $r_1(X); w_1(X); r_2(X); w_2(X); r_1(Y); w_1(Y); r_2(Y); w_2(Y)$

$r_1(X); w_1(X); r_2(X); r_1(Y); w_2(X); w_1(Y); r_2(Y); w_2(Y)$

$r_1(X); w_1(X); r_1(Y); r_2(X); w_2(X); w_1(Y); r_2(Y); w_2(Y)$

$r_1(X); w_1(X); r_1(Y); r_2(X); w_1(Y); w_2(X); r_2(Y); w_2(Y)$

S': $r_1(X); w_1(X); r_1(Y); w_1(Y); r_2(X); w_2(X); r_2(Y); w_2(Y)$

Serial

Example 2

What about this one?

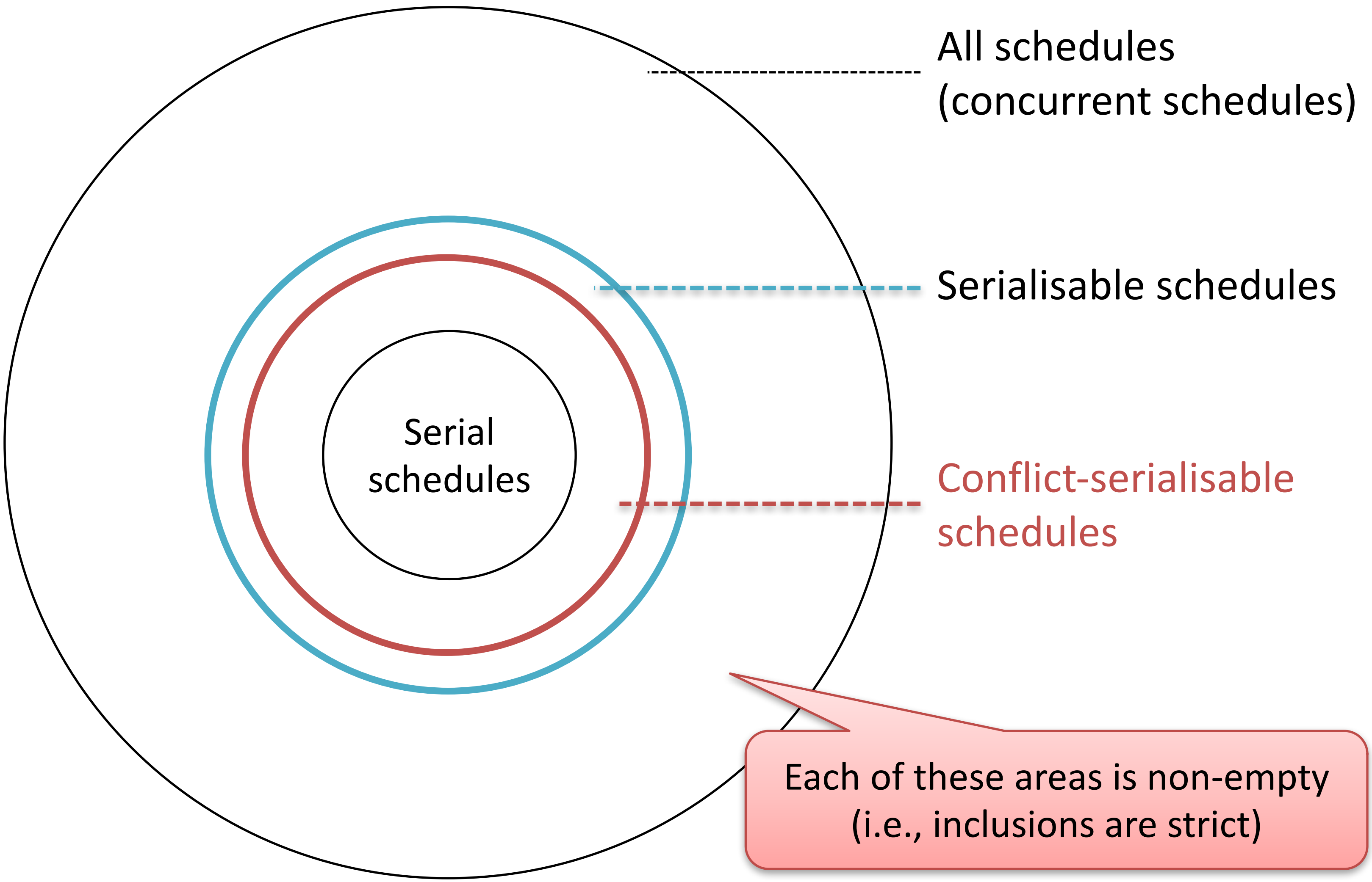
S: $r_2(X); r_1(Y); w_2(X); r_2(Y); r_3(X); w_1(Y); w_3(X); w_2(Y)$

Example 2

What about this one?

S: $r_2(X); r_1(Y); w_2(X); r_2(Y); r_3(X); w_1(Y); w_3(X); w_2(Y)$

Schedules



All schedules
(concurrent schedules)
Serialisable schedules
Conflict-serialisable
schedules
Serial
schedules
Each of these areas is non-empty
(i.e., inclusions are strict)

Example

A schedule that is **not** conflict-serialisable, but serialisable:

S: $w_2(X); w_1(X); w_1(Y); w_2(Y); w_3(X);$

S': $w_1(X); w_1(Y); w_2(X); w_2(Y); w_3(X);$

Why is it serialisable?

- The change in the database is given by $w_2(Y); w_3(X);$ and neither transaction reads anything

Why is it **not** conflict-serialisable?

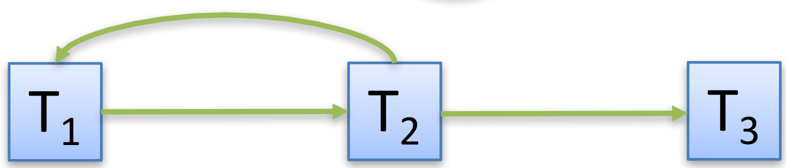
- In a bit

How can we *test* if a schedule is *conflict-serialisable*?

Example

A schedule that is **not** conflict-serialisable:

S: $r_2(X); r_1(Y); \boxed{w_2(X)}; \boxed{r_2(Y)}; \boxed{r_3(X)}; \boxed{w_1(Y)}; w_3(X); \boxed{w_2(Y)}$



The cycle indicates that no serial schedule is conflict-equivalent to S

How do we know?

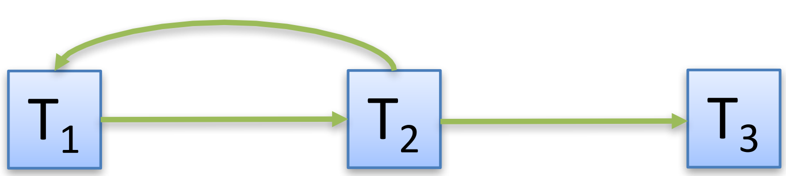
- Identify the transactions in S
- Identify the conflicts in S
- Conflicts impose constraints on the order of the transactions in any conflict-equivalent serial schedule

Precedence Graph

- The precedence graph for a schedule S is defined as follows:
  - It is a **directed graph.**
  - Its **nodes** are the transactions that occur in S.
  - It has an **edge** from transaction T_i to transaction T_j if there is a conflicting pair of operations op_1 and op_2 in S such that
    - op_1 appears before op_2 in S
    - op_1 belongs to transaction T_i
    - op_2 belongs to transaction T_j.
- Example:

S: $r_2(X); r_1(Y); w_2(X); r_2(Y); r_3(X); w_1(Y); w_3(X); w_2(Y)$

Precedence graph for S:



Testing Conflict-Serialisability

- To test if a schedule S is **conflict-serialisable**:
  - Construct the precedence graph for S.
  - If the precedence graph is **acyclic**, then S is conflict-serialisable. Otherwise not.

Acyclic graph: graph without a directed cycle

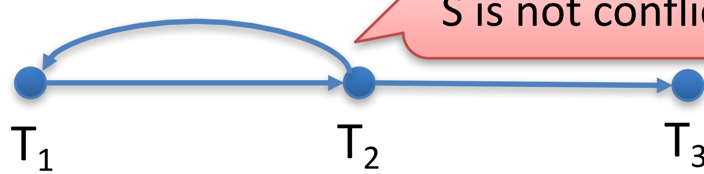
- Example 1: S: $r_1(X); w_1(X); r_2(X); w_2(X); r_1(Y); w_1(Y); r_2(Y); w_2(Y)$

  Precedence graph for S:

has no cycle → S is conflict-serialisable

- Example 2: S: $r_2(X); r_1(Y); w_2(X); r_2(Y); r_3(X); w_1(Y); w_3(X); w_2(Y)$

  Precedence graph for S:



contains a cycle → S is not conflict-serialisable

Why does this work?

$$T_1 \longrightarrow T_2$$

This says:

- There is a conflict between an operation in T_1 (that appears first) and an operation in T_2

All conflict-equivalent schedulers: operation in T_1 is before operation in T_2

Why does this work?

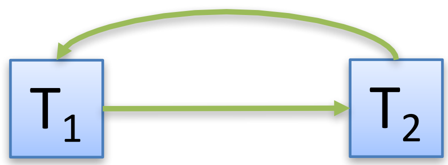
$T_1 \rightarrow T_2$

All conflict-equivalent schedulers: operation x in T_1 is before operation y in T_2

- Proof by contradiction: Assume conflict-equivalent scheduler S' where this is not so
- Consider first *consecutive* swap between S and S' where x goes from being before y to being after y
- In that swap, either:
  - We swap x and y (not legal since they conflict)
  - Or we swap at most 1 of them (contradicts choice of swap)

Implication of a cycle

- A cycle in the precedence graph



- In serial scheduler: No transaction can be first among those in the cycle (since another must be before)

Exercise (3 min)

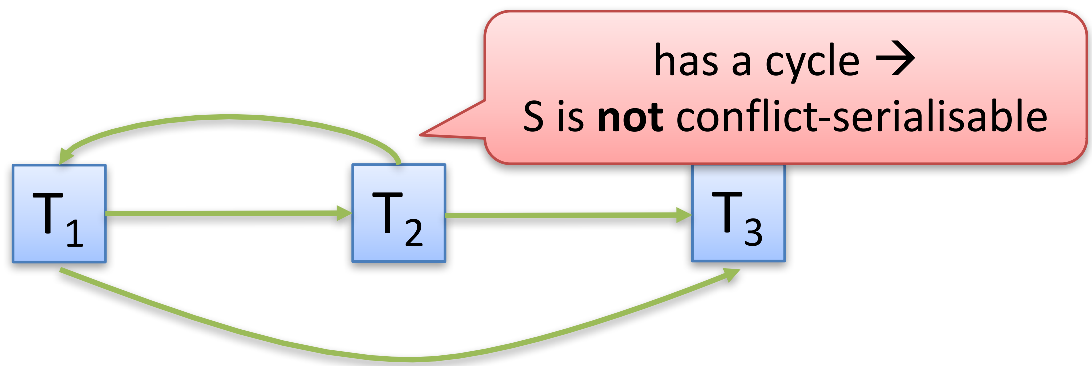
- Verify that the schedule below is **not** conflict-serialisable (using a precedence graph).

S: $w_2(X)$; $w_1(X)$; $w_1(Y)$; $w_2(Y)$; $w_3(X)$;

Exercise (3 min)

- Verify that the schedule below is **not** conflict-serialisable (using a precedence graph).

S: $w_2(X)$; $w_1(X)$; $w_1(Y)$; $w_2(Y)$; $w_3(X)$;



Exercise 2 (5 min)

- Verify that the schedule below is conflict-serialisable (using a precedence graph).

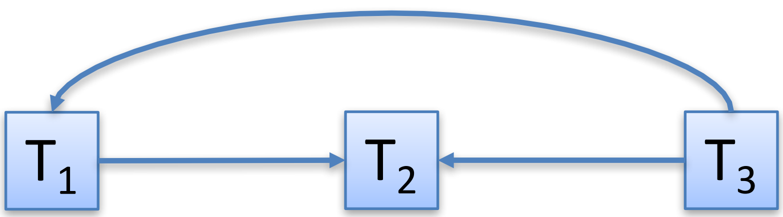
S: $r_1(Y)$, $r_3(Y)$, $r_1(X)$, $r_2(X)$, $w_2(X)$, $r_3(Z)$, $w_3(Z)$, $r_1(Z)$, $w_1(Y)$, $r_2(Z)$

- Can you find a conflict-equivalent serial schedule (using your precedence graph)?

Exercise 2 (5 min)

- Verify that the schedule below is conflict-serialisable (using a precedence graph).

S: $r_1(Y)$, $r_3(Y)$, $r_1(X)$, $r_2(X)$, $w_2(X)$, $r_3(Z)$, $w_3(Z)$, $r_1(Z)$, $w_1(Y)$, $r_2(Z)$



has no cycle → S is conflict-serialisable

Any topological ordering yields a conflict-equivalent serial schedule

- Can you find a conflict-equivalent serial schedule (using your precedence graph)?

$r_3(Y)$, $r_3(Z)$, $w_3(Z)$

Exercise 2 (5 min)

- Verify that the schedule below is conflict-serialisable (using a precedence graph).

S: $r_1(Y), r_3(Y), r_1(X), r_2(X), w_2(X), r_3(Z), w_3(Z), r_1(Z), w_1(Y), r_2(Z)$

has no cycle → S is conflict-serialisable

Any topological ordering yields a conflict-equivalent serial schedule

- Can you find a conflict-equivalent serial schedule (using your precedence graph)?

$r_3(Y), r_3(Z), w_3(Z)$

Exercise 2 (5 min)

- Verify that the schedule below is conflict-serialisable (using a precedence graph).

S: $r_1(Y)$, $r_3(Y)$, $r_1(X)$, $r_2(X)$, $w_2(X)$, $r_3(Z)$, $w_3(Z)$, $r_1(Z)$, $w_1(Y)$, $r_2(Z)$

T_2

- Can you find a conflict-equivalent serial schedule (using your precedence graph)?

$r_3(Y)$, $r_3(Z)$, $w_3(Z)$, $r_1(Y)$, $r_1(X)$, $r_1(Z)$, $w_1(Y)$,

Exercise 2 (5 min)

- Verify that the schedule below is conflict-serialisable (using a precedence graph).

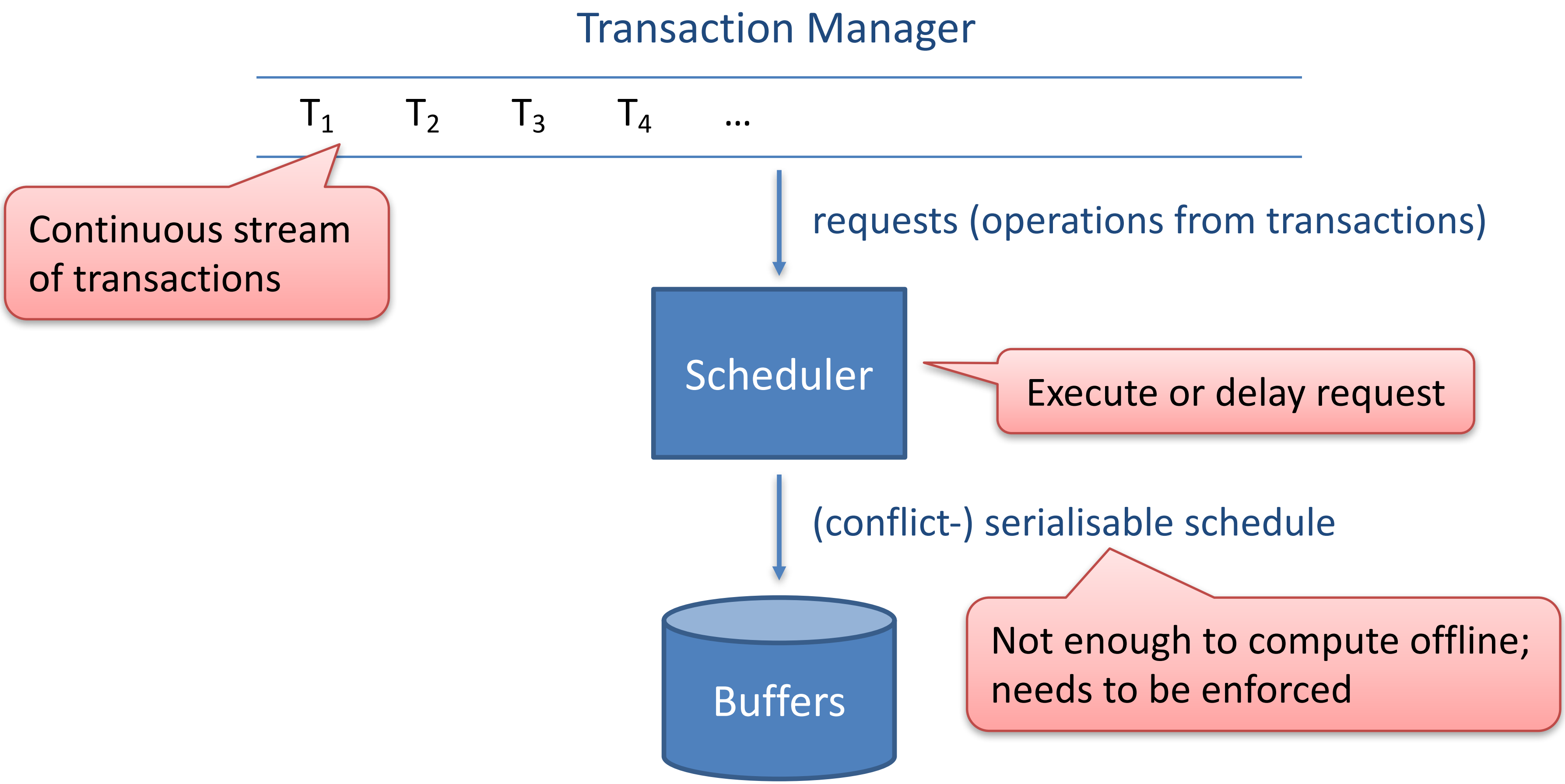
S: $r_1(Y)$, $r_3(Y)$, $r_1(X)$, $r_2(X)$, $w_2(X)$, $r_3(Z)$, $w_3(Z)$, $r_1(Z)$, $w_1(Y)$, $r_2(Z)$

- Can you find a conflict-equivalent serial schedule (using your precedence graph)?

$r_3(Y)$, $r_3(Z)$, $w_3(Z)$, $r_1(Y)$, $r_1(X)$, $r_1(Z)$, $w_1(Y)$, $r_2(X)$, $w_2(X)$, $r_2(Z)$

Are we done?

Transaction Scheduling in a DBMS



Enforcing Conflict-Serialisability Using Locks

Summary

- What schedules are “good”?
- First approximation: serialisable schedules
  - Equivalent to serial schedules
  - Guarantee correctness & consistency
  - But: difficult to check
- A good compromise: conflict-serialisable schedules
  - Conflict-equivalent to serial schedules
  - Imply serialisability (so inherit all the nice properties)