OVERVIEW OF TRANSACTION MANAGEMENT

CSC443, Winter 2018
Sayyed Nezhadi

Chapter 16

Important Properties of Transactions

ACID

- Atomic: either all actions are carried out or none are (e.g. when a system crash occurs)
- Consistency: each transaction preserves the consistency of the database (database constraints are preserved)
- Isolation: it appears to the user as if only one transaction executes at a time (even if the DBMS interleaves the actions of several transactions for performance reasons)
- Durability: the effects of transaction persist even if the system crashes before all its changes are reflected on disk

Transactions and Schedules

- A transaction is seen by the DBMS as a series, or list, of actions.
- The actions that can be executed by a transaction include reads ($R_T(O)$)and writes ($W_T(O)$) of database objects (O).
- Each transaction must specify as its final action either commit ($Commit_T$) (i.e., complete successfully) or abort ($Abort_T$) (i.e., terminate and undo all the actions carried out thus far).
- A schedule is a list of actions (reading, writing, aborting, or committing) over a set of transactions in a sequence that they need to be executed.

Assumptions

- Transactions interact with each other only via database read and write operations; for example, they are not allowed to exchange messages.

- A database is a collection of independent objects. When objects are added to or deleted from a database or there are relationships between database objects that we want to exploit for performance, some additional issues arise.

Example: A Schedule Involving Two Transactions

T1	T2
R(A)	
W(A)	
	R(B)
	W(B)
R(C)	
W(C)	

☛ *For simplicity, we omit subscripts T1 & T2. We also assume there is a Commit at the end of each transaction*

Concurrent Execution

- The DBMS interleaves the actions of different transactions to improve performance, but not all interleaving should be allowed.
- Ensuring transaction isolation while permitting such concurrent execution is difficult but necessary for performance reasons:
  - While one transaction is waiting for a page to be read in from disk, the CPU can process another transaction. Overlapping I/O and CPU activity increases system throughput.
  - Interleaved execution of a short transaction with a long transaction usually allows the short transaction to complete quickly.

Serializability

- A serializable schedule over a set **S** of committed transactions is a schedule whose effect on any consistent database instance is guaranteed to be identical to that of some complete serial schedule over **S**.

T1	T2
R(A)	
W(A)	
	R(A)
	W(A)
R(B)	
W(B)	
	R(B)
	W(B)

Two examples of serializable schedules

T1	T2
	R(A)
	W(A)
R(A)	
	R(B)
	W(B)
W(A)	
R(B)	
W(B)	

Anomalies Due to Interleaved Execution

Three main ways:

- Reading Uncommitted Data (WR Conflicts)
- Unrepeatable Reads (RW Conflicts)
- Overwriting Uncommitted Data (WW Conflicts)

Reading Uncommitted Data (WR Conflicts)

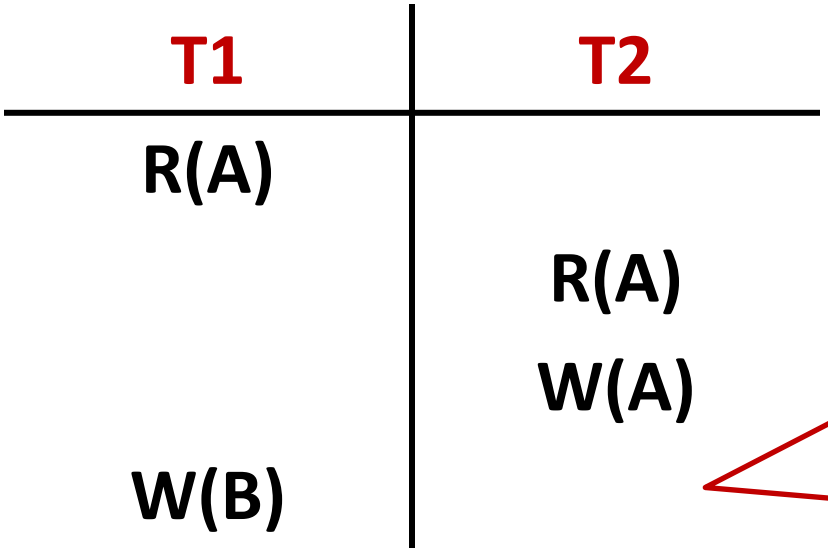
T1	T2
R(A)	
W(A)	
	R(A)
	W(A)
	R(B)
	W(B)
R(B)	
W(B)	

Reading object **A** that is modified by **T1** but object **B** is not modified yet.

Example:

- T1 transfers $100 from A to B
- T2 increments both A & B by 6%

Unrepeatable Reads (RW Conflicts)

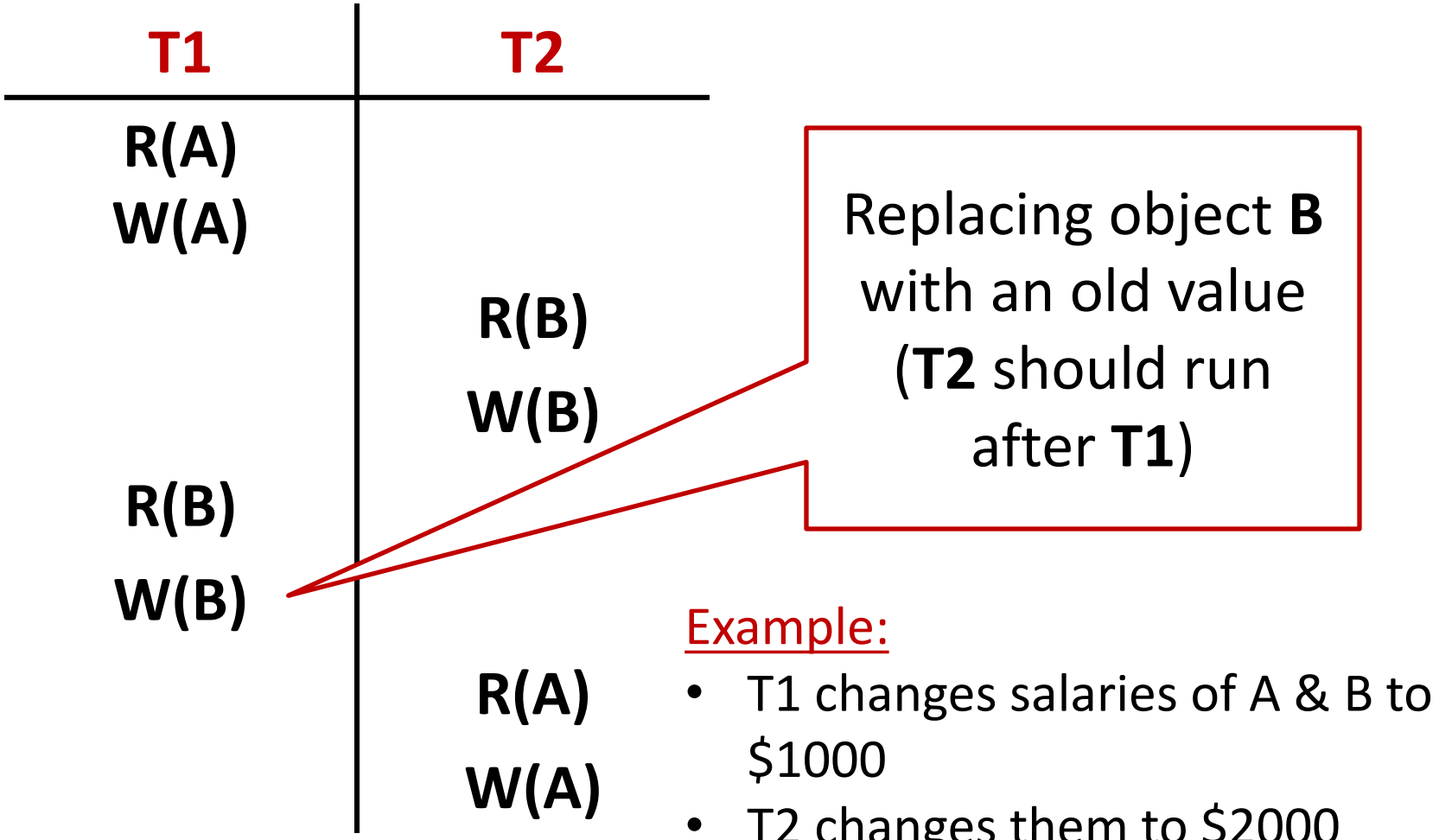


T2 changes the value of object **A** that has been read by the transaction **T1**, while **T1** is still in progress

Example:

- Both T1 & T2 try to order a book with one copy in stock

Overwriting Uncommitted Data (WW Conflicts)



Example:

- T1 changes salaries of A & B to $1000
- T2 changes them to $2000

Unrecoverable Schedules

Note: all actions of aborted transactions are to be undone

T1	T2
R(A)	
W(A)	
	R(A)
	W(A)
	R(B)
	W(B)
	Commit
Abort	

T2 is committed and **T1** cannot recover value of object **A**

Example:

- T1 changes salaries of A & B to $1000
- T2 changes them to $2000

Lock-based Concurrency Control

- A DBMS typically uses a locking protocol to ensure that only serializable, recoverable schedules are allowed and that no actions of committed transactions are lost while undoing aborted transactions.
- A lock is a small bookkeeping object associated with a database object.
- A locking protocol is a set of rules to be followed by each transaction to ensure the net effect is identical to executing all transactions in same serial order.
- Different locking protocols use different types of locks, such as shared locks or exclusive locks.

Strict Two-Phase Locking (Strict 2PL)

- The most widely used locking protocol.
- Has two rules:
  - If a transaction T wants to read (respectively, modify) an object, it first requests a shared (respectively, exclusive) lock on the object.
  - All locks held by a transaction are released when the transaction is completed.
- Shared lock ($S_T(O)$): other transactions can read but not write.
- Exclusive lock ($X_T(O)$): no other transactions can read or write

Notes about Strict 2PL

- A transaction that has an exclusive lock can also read the object; an additional shared lock is not required.

- A transaction that has an exclusive lock can also read the object; an additional shared lock is not required.

- The DBMS keeps track of the locks it has granted and ensures that if a transaction holds an exclusive lock on an object, no other transaction holds a shared or exclusive lock on the same object.

Example: Uncommitted Data

T1	T2
R(A)	
W(A)	
	R(A)
	W(A)
	R(B)
	W(B)
	Commit
R(B)	
W(B)	
Commit	

Strict 2PL

T1	T2
X(A)	
R(A)	
W(A)	
X(B)	
R(B)	
W(B)	
Commit	
	X(A)
	R(A)
	W(A)
	X(B)
	R(B)
	W(B)
	Commit

Deadlocks

- T1 is waiting for T2 to release its lock and T2 is waiting for T1 to release its lock. Such a cycle of transactions waiting for locks to be released is called a deadlock.

- The DBMS must either prevent or detect (and resolve) such deadlock situations; the common approach is to detect and resolve deadlocks. A simple way to identify deadlocks is to use a timeout mechanism.

T1	T2
X(A)	
	X(B)
Request X(B)	
	Request X(A)