



UNIVERSITY OF COPENHAGEN

Concurrency Control: Serializability, Schedules, Advanced Topics

ACS, Marcos Vaz Salles

Do-it-yourself-recap: Locking Solutions for Isolation in ACID Transactions

	S	X
S	Yes	No
X	No	No

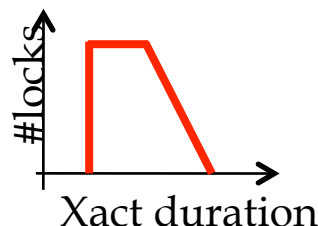
Why two phases? Deadlocks? Cascading aborts?

Solution 4

- 1) Get exclusive locks on data items that are modified and get shared locks on data items that are read
- 2) Execute transaction and release locks on objects no longer needed *during execution*

- Greater concurrency
- Conservative Two Phase Locking (2PL)

- Problems?

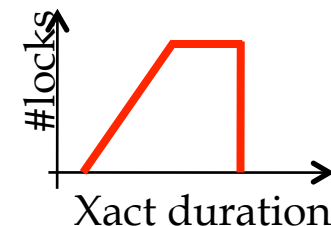


Solution 5

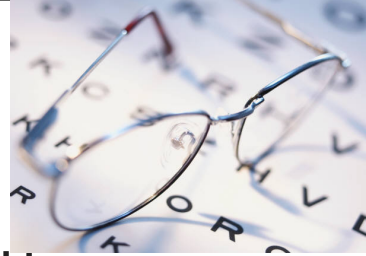
- 1) Get exclusive locks on data items that are modified and get shared locks on data items that are read, but do this *during execution* of transaction (as needed)
- 2) Release all locks

- Greater concurrency
- Strict Two Phase Locking (2PL)

- Problems?



What should we learn today?



- Discuss the definition of serializability and the notion of anomalies
- Apply the conflict-serializability test using a precedence graph to transaction schedules
- Discuss the difference between conflict-serializability and view-serializability
- Explain deadlock prevention and detection techniques
- Apply deadlock detection using a waits-for graph to transaction schedules
- Explain the optimistic concurrency control and multi-version concurrency control models
- Predict validation decisions under optimistic concurrency control

Is Strict 2PL correct? (assuming database is **not** dynamic)

- We will formalize now **serializability** and argue that Strict 2PL is correct
 - Full proof is left as homework 😊
- Strict 2PL can however deadlock
 - We will see how to handle deadlock automatically



Schedules

- Consider a possible interleaving (schedule):

T1:	$A = A + 100,$	$B = B - 100$
T2:	$A = 1.06 * A, \quad B = 1.06 * B$	

- The system's view of the schedule:

T1:	$R(A), W(A),$	$R(B), W(B)$
T2:	$R(A), W(A), R(B), W(B)$	



Scheduling Transactions

- *Serial schedule*: Schedule that does not interleave the actions of different transactions.
- *Equivalent schedules*: For any database state
 - The effect (on the set of objects in the database) of executing the schedules is the same
 - The values read by transactions is the same in the schedules
 - Assume no knowledge of transaction logic
- *Serializable schedule*: A schedule that is equivalent to some serial execution of the transactions.

(Note: If each transaction preserves consistency, every serializable schedule preserves consistency.)



Anomalies with Interleaved Execution

- Reading Uncommitted Data (WR Conflicts, “dirty reads”):

T1:	R(A), W(A),	R(B), W(B), Abort
T2:	R(A), W(A), C	

- Unrepeatable Reads (RW Conflicts):

T1:	R(A),	R(A), W(A), C
T2:	R(A), W(A), C	



Anomalies (contd.)

- Overwriting Uncommitted Data (WW Conflicts):

T1:	W(A),	W(B), C
T2:	W(A), W(B), C	



Conflict Serializable Schedules

- Two schedules are **conflict equivalent** if:
 - Involve the same actions of the same transactions
 - Every pair of conflicting actions is ordered the same way
- Schedule S is **conflict serializable** if S is conflict equivalent to some serial schedule



Example

- A schedule that is not conflict serializable:

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



- The cycle in the graph reveals the problem. The output of T1 depends on T2, and vice-versa.



Precedence Graph

- Precedence graph: One node per Xact; edge from T_i to T_j if operation in T_j conflicts with earlier operation in T_i .
- Theorem: Schedule is conflict serializable if and only if its precedence graph is acyclic
- Strict 2PL only allows conflict serializable schedule
 - Precedence graph is always acyclic



Are the following schedules conflict-serializable?

- Build the precedence graph for each of the following transaction schedules

T1:	R(A)		W(B)			C
T2:		R(B)		R(A)	R(C)	C
T3:			R(B)		W(C)	C

Note: C alone stands for commit

T1:	R(A)		W(B)			C
T2:		R(B)		R(A)	R(C)	C
T3:			R(B)		W(C)	C

Returning to Definition of Serializability

- A schedule S is serializable if there exists a serial order SO such that:
 - The **state of the database** after S is the **same** as the state of the database after SO
 - The **values read** by each transaction in S are the **same** as that returned by each transaction in SO
 - Database does not know anything about the internal structure of the transaction programs
- **Under this definition, certain serializable executions are not conflict serializable!**



Is this schedule serializable?

T1:	R(A)		W(A)
T2:		W(A)	
T3:			W(A)

T1:	R(A), W(A)		
T2:		W(A)	
T3:			W(A)



View Serializability

- Schedules S1 and S2 are **view equivalent** if:
 - If T_i reads initial value of A in S1, then T_i also reads initial value of A in S2
 - If T_i reads value of A written by T_j in S1, then T_i also reads value of A written by T_j in S2
 - If T_i writes final value of A in S1, then T_i also writes final value of A in S2

T1:	R(A)		W(A)
T2:		W(A)	
T3:			W(A)

T1:	R(A), W(A)		
T2:		W(A)	
T3:			W(A)



Deadlocks

- Deadlock: Cycle of transactions waiting for locks to be released by each other.
- Two ways of dealing with deadlocks:
 - Deadlock prevention
 - Deadlock detection



Deadlock Prevention

- Assign priorities based on timestamps. Assume T_i wants a lock that T_j holds. Two policies are possible:
 - Wait-Die: If T_i has higher priority, T_i waits for T_j ; otherwise T_i aborts
 - Wound-wait: If T_i has higher priority, T_j aborts; otherwise T_i waits
- If a transaction re-starts, make sure it has its original timestamp



Deadlock Detection

- Create a **waits-for graph**:
 - Nodes are transactions
 - There is an edge from T_i to T_j if T_i is waiting for T_j to release a lock
- Periodically check for cycles in the waits-for graph



Deadlock Detection

• Example

T1: S(A), R(A), S(B)

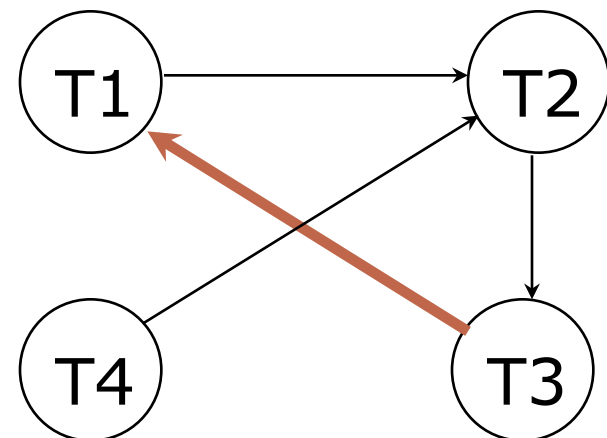
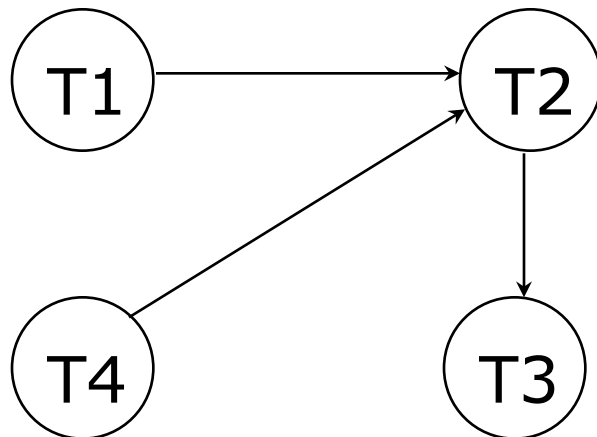
T2: X(B), W(B)

T3: S(C), R(C)

T4: X(B)

X(C)

X(A)



Do the following schedules lead to deadlock?

- Build the waits-for graph for each of the following transaction schedules

T1:	S(A)		X(D)	X(C)	C
T2:		X(A)			X(B)
T3:	S(B)				S(C)

Note: we only show locking operations for brevity!
C alone denotes commit, all locks released

T1:		S(C)	S(A)		X(D)
T2:	S(B)			X(C)	
T3:	S(D)	X(B)			



Questions so far?



The Problems with Locking

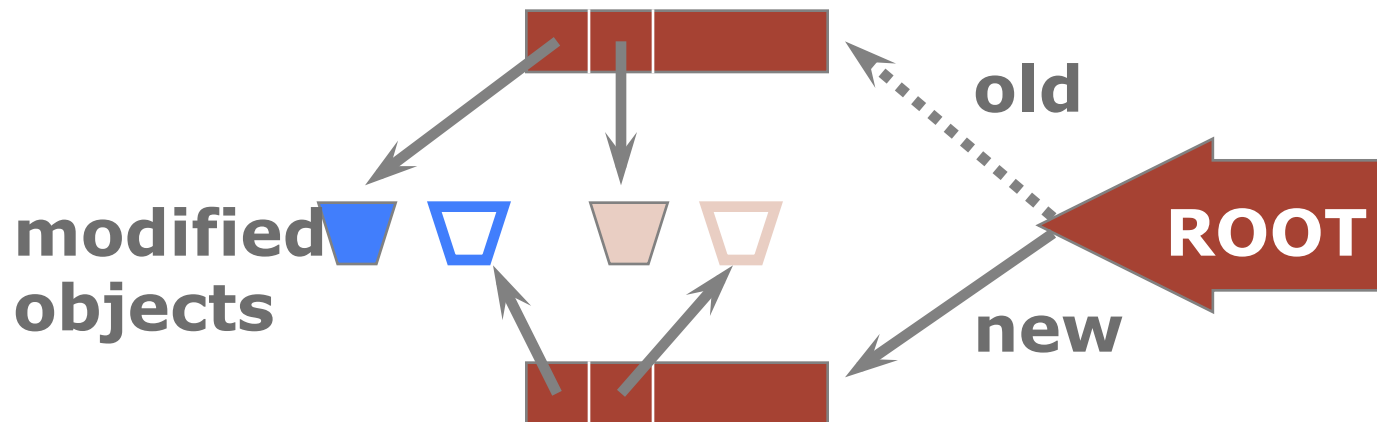
- Locking is a pessimistic approach in which conflicts are prevented. Disadvantages:
 - Lock management overhead.
 - Deadlock detection/resolution.
 - Lock contention for heavily used objects.
- Remember: We must devise a way to **enforce serializability**, without destroying concurrency
- Two approaches:
 - Prevent violations → **locking**
 - Fix violations → **aborts**

How can we design a protocol based on
aborts instead of locks?



Optimistic CC: Kung-Robinson Model

- Xacts have three phases
- **READ:** Xacts read from the database, but make changes to private copies of objects.
- **VALIDATE:** Check for conflicts.
- **WRITE:** Make local copies of changes public.



Validation

- Test conditions that are **sufficient** to ensure that no conflict occurred.
- Each Xact is assigned a numeric id.
 - Just use a **timestamp**.
- Xact ids assigned at end of READ phase, just before validation begins. (**Why then?**)
- **ReadSet(Ti)**: Set of objects read by Xact Ti.
- **WriteSet(Ti)**: Set of objects modified by Ti.



Test 1

- For all i and j such that $T_i < T_j$, check that T_i completes before T_j begins.



Test 2

- For all i and j such that $T_i < T_j$, check that:
 - T_i completes before T_j begins its Write phase +
 - $\text{WriteSet}(T_i) \cap \text{ReadSet}(T_j)$ is empty.

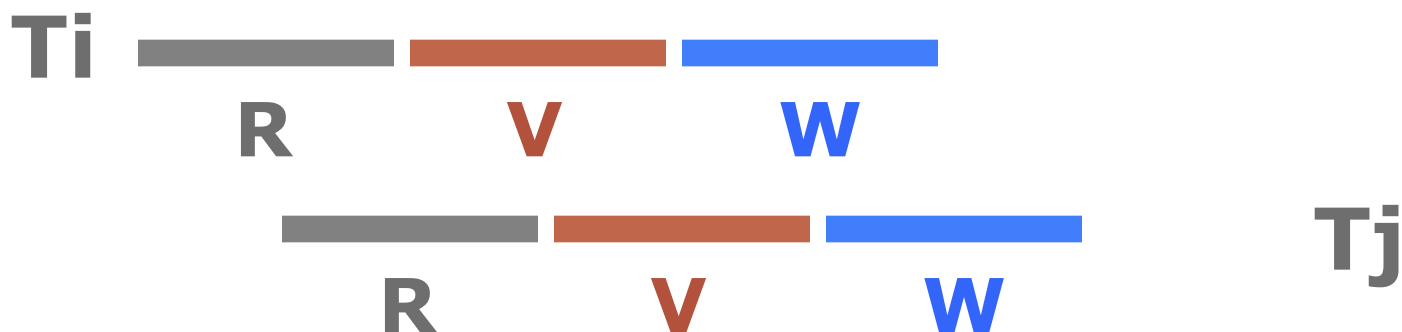


Does T_j read dirty data?



Test 3

- For all i and j such that $T_i < T_j$, check that:
 - T_i completes Read phase before T_j does +
 - $\text{WriteSet}(T_i) \cap \text{ReadSet}(T_j)$ is empty +
 - $\text{WriteSet}(T_i) \cap \text{WriteSet}(T_j)$ is empty.



Does T_j read dirty data? Does T_i overwrite T_j 's writes?



Validation Example

- Predict whether T3 will be allowed to commit, given the transactions below

T1: $RS(T1) = \{1, 2, 3\}$, $WS(T1) = \{4\}$,
T1 completes before T3 begins with its write phase.

T2: $RS(T2) = \{6, 7, 8\}$, $WS(T2) = \{8\}$,
T2 completes read phase before T3 does.

T3: $RS(T3) = \{3, 5, 6, 7\}$, $WS(T3) = \{4\}$,
allow commit or roll back?

Overheads in Optimistic CC

- Must record read/write activity in ReadSet and WriteSet per Xact.
 - Must create and destroy these sets as needed.
- Must check for conflicts during validation, and must make validated writes “global”.
 - Critical section can reduce concurrency.
 - Scheme for making writes global can reduce clustering of objects.
- Optimistic CC restarts Xacts that fail validation.
 - Work done so far is wasted; requires clean-up.
- Still, optimistic techniques widely used in software transactional memory (STM), main-memory databases



Multiversion Concurrency Control (MVCC)

- This approach maintains a number of **versions** of a data item and allocates the **right version to a read operation** of a transaction. Thus unlike other mechanisms a **read operation in this mechanism is never rejected**.
- Side effect:
 - Significantly more storage (RAM and disk) is required to maintain multiple versions. To check unlimited growth of versions, a **garbage collection** is run when some criteria is satisfied
- Many commercial database systems implement a combination of MVCC and S2PL
- See compendium for more details



Snapshot Isolation

- Often databases implement properties that are **weaker** than serializability
- **Snapshot isolation**
 - **Snapshots:** Transactions see snapshot as of beginning of their execution
 - **First Committer Wins:** Conflicting writes to same item lead to aborts
- May lead to **write skew**
 - Database must have at least one doctor on call
 - Two doctors on call concurrently examine snapshot and see exactly each other on call
 - Doctors update their own records to being on leave
 - No write-write conflicts: different records!
 - After commits, database has no doctors on call



Transaction Support in SQL-92

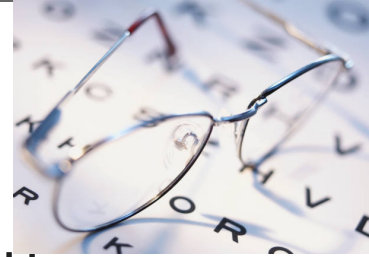
- Each transaction has an access mode, a diagnostics size, and an isolation level.

Problem with SQL standard: snapshot isolation satisfies all requirements!

Isolation Level	Dirty Read	Unrepeatable Read	Phantom Problem
Read Uncommitted	Maybe	Maybe	Maybe
Read Committed	No	Maybe	Maybe
Repeatable Reads	No	No	Maybe
Serializable	No	No	No



What should we learn today?



- Discuss the definition of serializability and the notion of anomalies
- Apply the conflict-serializability test using a precedence graph to transaction schedules
- Discuss the difference between conflict-serializability and view-serializability
- Explain deadlock prevention and detection techniques
- Apply deadlock detection using a waits-for graph to transaction schedules
- Explain the optimistic concurrency control and multi-version concurrency control models
- Predict validation decisions under optimistic concurrency control