Concurrency Control

CENG315 INFORMATION MANAGEMENT

Three Concurrency Problems

- In a multi-processing environment transactions can interfere with each other.
- Three concurrency problems can arise, that any DBMS must account for and avoid:
  - Lost Update
  - Uncommitted Dependency
  - Inconsistent Analysis

Transaction A	*Time*	*Transaction B*
–		–
–		–
RETRIEVE t	t1	–
–		–
–	t2	RETRIEVE t
–		–
UPDATE t	t3	–
–		–
–	t4	UPDATE t
–		–

Fig. 16.1 Transaction *A* loses an update at time *t4* .

The Lost Update Problem

A lost update occurs when a second transaction reads the state of the database prior to the first one writing a change, and then stops on the first one's change with its own update.

The Uncommitted Dependency Problem

Transaction A	Time	Transaction B
—		—
—		—
—	t1	UPDATE t
—		—
RETRIEVE t	t2	—
—		—
—	t3	ROLLBACK
—		

Fig. 16.2 Transaction *A* becomes dependent on an uncommitted change at time *t2*

An uncommitted dependency occurs when a second transaction relies on a change which has not yet been committed, which is rolled back after the second transaction has begun.

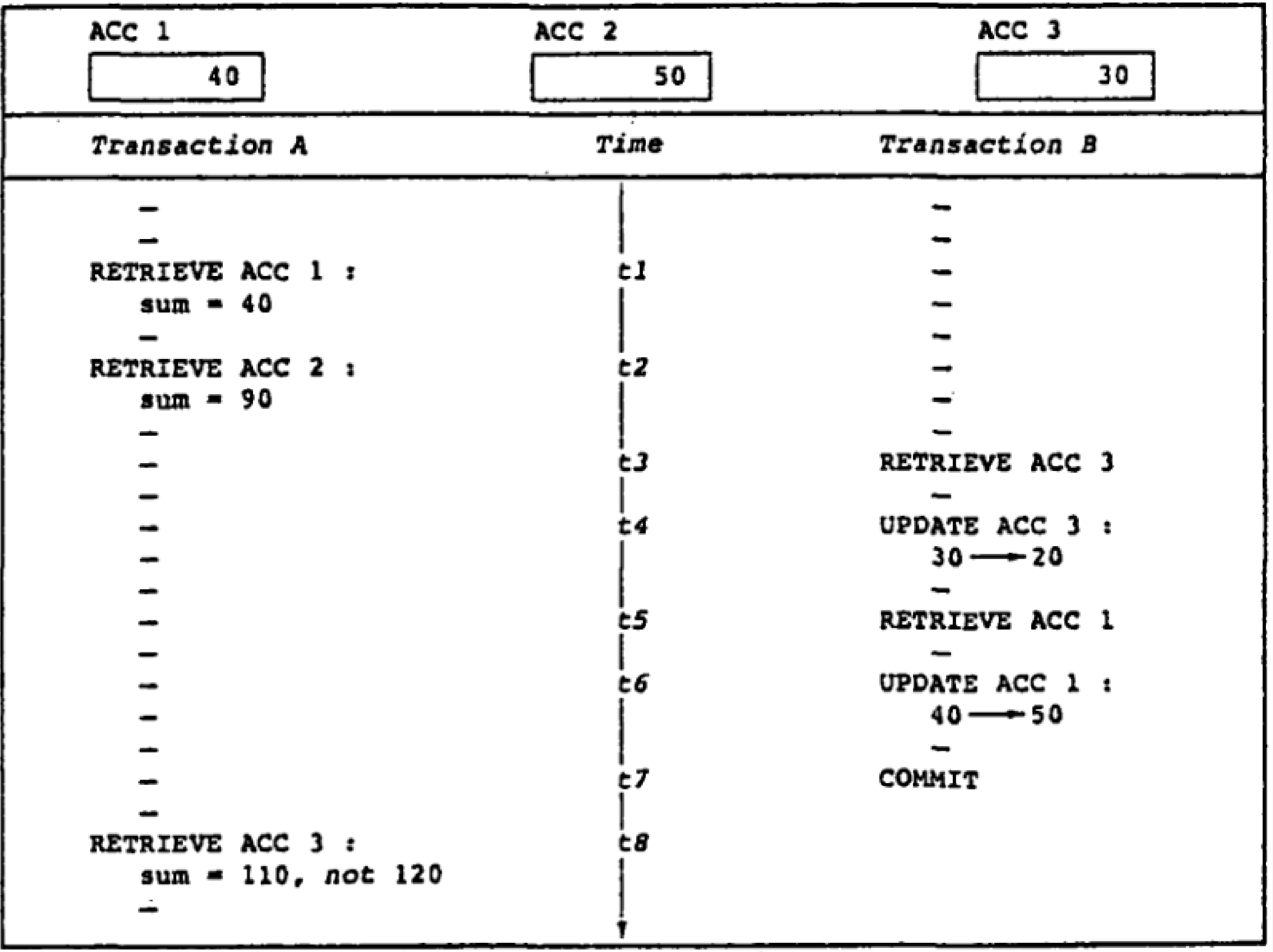


Fig. 16.4 Transaction *A* performs an inconsistent analysis

The Inconsistent Analysis Problem

An inconsistent analysis occurs when totals are calculated during interleaved updates.

Conflicts

- If *A* and *B* are concurrent transactions, problems can occur if *A* and *B* want to read or write the same database object, say tuple *t*.
- There are four possibilities:
  - RR (Read Read): Reads cannot interfere with each other, so there is no problem in this case.
  - RW (Read Write)
  - WR (Write Read)
  - WW (Write Write)

RW Conflict

- *A* reads *t* and then *B* wants to write *t*.
- If *B* is allowed to perform its write, then the inconsistent analysis problem can arise.
  - As we saw in Fig. 16.4.
- Thus, we can say that inconsistent analysis is caused by a RW conflict.

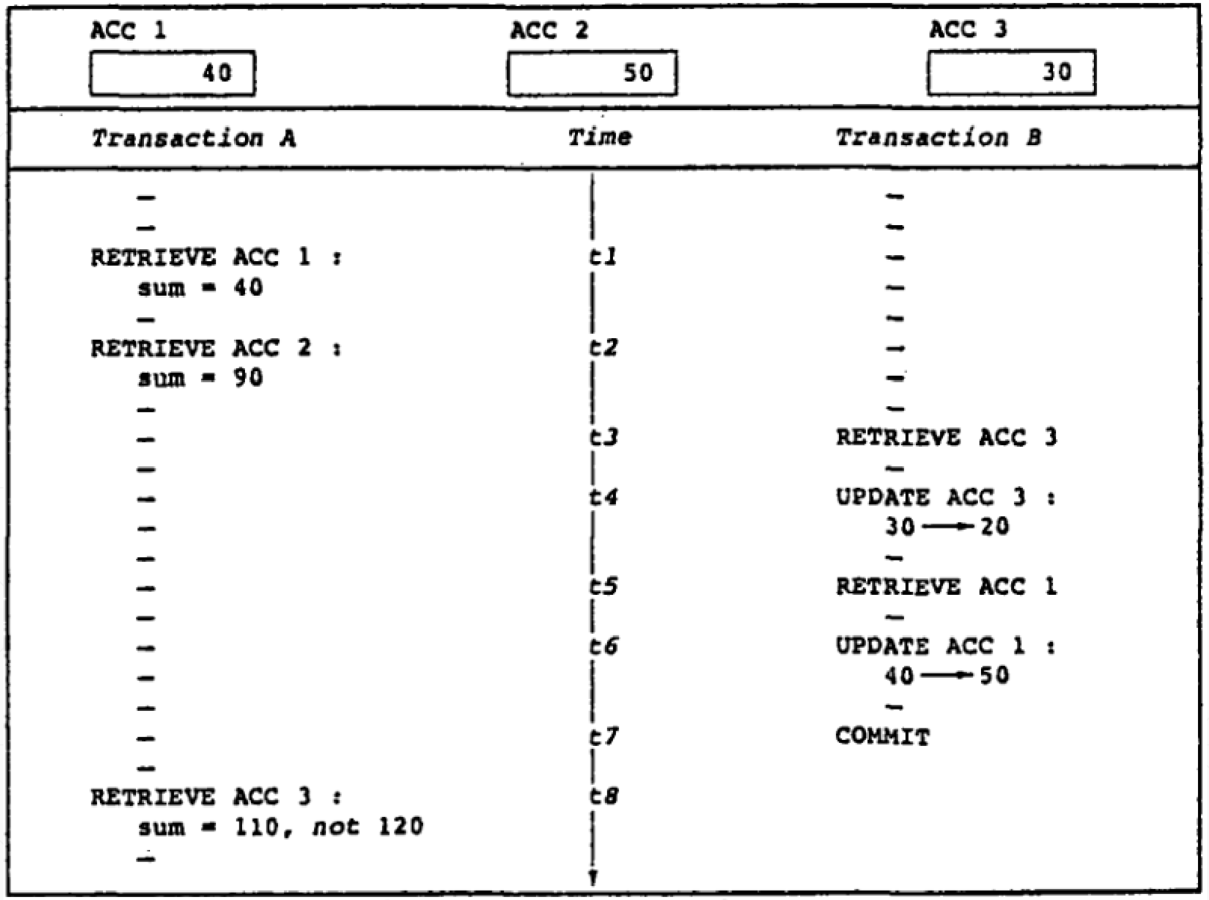


Fig. 16.4 Transaction *A* performs an inconsistent analysis

WR Conflict

- *B* writes *t* and then *A* wants to read *t*.
- If *A* is allowed to perform its read, then the uncommitted dependency problem can arise.
  - As we saw in Fig. 16.2.

Transaction A	Time	Transaction B
–		–
–		–
–	t1	UPDATE t
–		–
RETRIEVE t	t2	–
–		–
–	t3	ROLLBACK
–		

Fig. 16.2 Transaction *A* becomes dependent on an uncommitted change at time *t2*

WR Conflict

- *B* writes *t* and then *A* wants to read *t*.
- If *A* is allowed to perform its read, then the uncommitted dependency problem can arise.
  - As we saw in Fig. 16.2.
- Thus, we can say that uncommitted dependencies are caused by WR conflicts.
- Note: *A*'s read, if it is allowed, is said to be a dirty read.

WW Conflict

- *A* writes *t* and then *B* wants to write *t*.
- If *B* is allowed to perform its write, then the lost update problem can arise.
  - As we saw in Fig. 16.1.

Transaction A	Time	Transaction B
–		–
–		–
RETRIEVE t	t1	–
–		–
–	t2	RETRIEVE t
–		–
UPDATE t	t3	–
–		–
–	t4	UPDATE t
–		–

Fig. 16.1 Transaction *A* loses an update at time *t4* .

WW Conflict

- *A* writes *t* and then *B* wants to write *t*.
- If *B* is allowed to perform its write, then the lost update problem can arise.
  - As we saw in Fig. 16.1.
- Thus, we can say that lost updates are caused by WW conflicts.

Locking

- The mentioned problems can all be solved by means of a concurrency control mechanism called locking.
- A transaction locks a portion of the database to prevent concurrency problems.
- Exclusive lock (X) – write lock, will lock out all other transactions
- Shared lock (S) – read lock, will lock out writes, but allow other reads

Lock-Based Protocols

- Lock Compatibility Matrix:

	S	X
S	true	false
X	false	false

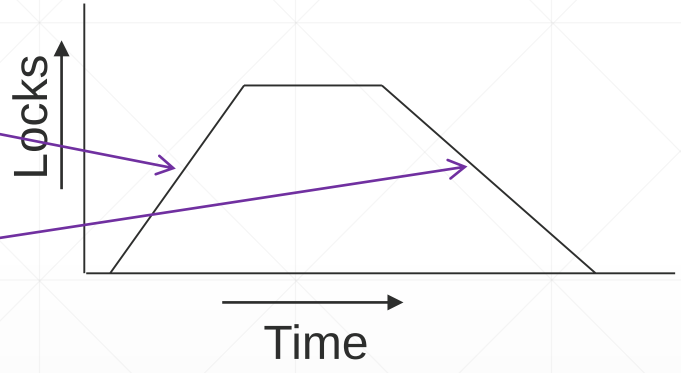
- A transaction may be granted a lock on an item if the requested lock is compatible with locks already held on the item by other transactions.
- Any number of transactions can hold shared locks on an item.
- But if any transaction holds an exclusive on the item no other transaction may hold any lock on the item.

Locking Protocol

- A locking protocol is a set of rules that state when a transaction may lock and unlock each of the data items in the database.
- **The Two-Phase Locking Protocol**

Two-Phase Locking Protocol

- A protocol which ensures conflict-serializable schedules.
- Phase 1: **Growing Phase**
  - Transaction may obtain locks
  - Transaction may not release locks
- Phase 2: **Shrinking Phase**
  - Transaction may release locks
  - Transaction may not obtain locks



Example: Two-Phase Locking Protocol

- Consider the transactions T_{34} and T_{35}:
  - Add lock and unlock instructions to transactions T_{34} and T_{35}, so that they observe the two-phase locking protocol.
    - A transaction requests a shared lock on data item Q by executing the lock-S(Q) instruction.
    - A transaction requests an exclusive lock on data item Q by executing the lock-X(Q) instruction.
    - A transaction can unlock a data item Q by the unlock(Q) instruction.

T_{34}: read(A);
read(B);
if $A = 0$ then $B := B + 1$;
write(B).

T_{35}: read(B);
read(A);
if $B = 0$ then $A := A + 1$;
write(A).

Example: Two-Phase Protocol (Cont.)

- Lock and unlock instructions:

T_{34}: read(A);
read(B);
if $A = 0$ then $B := B + 1$;
write(B).

T_{35}: read(B);
read(A);
if $B = 0$ then $A := A + 1$;
write(A).

T_{34}:		T_{35}:
lock-S(A)		lock-S(B)
read(A)		read(B)
lock-X(B)		lock-X(A)
read(B)		read(A)
if $A = 0$		if $B = 0$
then $B := B + 1$		then $A := A + 1$
write(B)		write(A)
unlock(A)		unlock(B)
unlock(B)		unlock(A)

Deadlock

- Consider the partial schedule:
- Neither T_3 nor T_4 can make progress — executing **lock-S**(*B*) causes T_4 to wait for T_3 to release its lock on *B*, while executing **lock-X**(*A*) causes T_3 to wait for T_4 to release its lock on *A*.
- Such a situation is called a **deadlock**.
  - To handle a deadlock one of T_3 or T_4 must be rolled back and its locks released.
- The potential for deadlock exists in most locking protocols.

T_3	T_4
lock-X(B)	
read(B)	
$B := B - 50$	
write(B)	
	lock-S(A)
	read(A)
	$\boxed{\text{lock-S}(B)}$
$\boxed{\text{lock-X}(A)}$	

Starvation

- **Starvation** is also possible if concurrency control manager is badly designed. For example:
  - A transaction may be waiting for an X-lock on an item, while a sequence of other transactions request and are granted an S-lock on the same item.
  - The same transaction is repeatedly rolled back due to deadlocks.
- Concurrency control manager can be designed to prevent starvation.

Example: Two-Phase Protocol and Deadlock

- Can the execution of these transactions result in a deadlock?

T_{34}:
lock-S(A)
read(A)
lock-X(B)
read(B)
if $A = 0$
then $B := B + 1$
write(B)
unlock(A)
unlock(B)

T_{35}:
lock-S(B)
read(B)
lock-X(A)
read(A)
if $B = 0$
then $A := A + 1$
write(A)
unlock(B)
unlock(A)

Example: Two-Phase Protocol and Deadlock (Cont.)

- Execution of these transactions can result in deadlock. For example, consider the following partial schedule:

T_{34}: lock-S(A)
read(A)
lock-X(B)
read(B)
if $A = 0$
then $B := B + 1$
write(B)
unlock(A)
unlock(B)

T_{35}: lock-S(B)
read(B)
lock-X(A)
read(A)
if $B = 0$
then $A := A + 1$
write(A)
unlock(B)
unlock(A)

T_{34}	T_{35}
lock-S (A)	
	lock-S (B)
	read(B)
read(A)	
lock-X (B)	
	lock-X (A)

Transaction A	*Time*	*Transaction B*
–		–
–		–
RETRIEVE t	t1	–
–		–
–	t2	RETRIEVE t
–		–
UPDATE t	t3	–
–		–
–	t4	UPDATE t
–		–

Fig. 16.1 Transaction *A* loses an update at time *t4* .

The Lost Update Problem

A lost update occurs when a second transaction reads the state of the database prior to the first one writing a change, and then stomps on the first one's change with its own update.

The Lost Update Problem - Revisited

Transaction A	Time	Transaction B
–		–
–		–
RETRIEVE t	t1	–
(acquire S lock on t)		–
–		–
–	t2	RETRIEVE t
–		(acquire S lock on t)
–		–
UPDATE t	t3	–
(request X lock on t)		–
wait		–
wait	t4	UPDATE t
wait		(request X lock on t)
wait		wait
wait		wait
wait		wait

Fig. 16.6 No update is lost, but deadlock occurs at time t4

Transaction A	*Time*	*Transaction B*
–		–
–		–
–	*t1*	UPDATE t
–		–
RETRIEVE t	*t2*	–
–		–
–	*t3*	ROLLBACK
–		

Fig. 16.2 Transaction *A* becomes dependent on an uncommitted change at time *t2*

The Uncommitted Dependency Problem

An uncommitted dependency occurs when a second transaction relies on a change which has not yet been committed, which is rolled back after the second transaction has begun.

The Uncommitted Dependency Problem - Revisited

Transaction A	Time	Transaction B
–		–
–		–
–	t1	UPDATE t
–		(acquire X lock on t)
–		–
RETRIEVE t	t2	–
(request S lock on t)		–
wait		–
wait	t3	COMMIT / ROLLBACK
wait		(release X lock on t)
resume : RETRIEVE t	t4	
(acquire S lock on t)		
–		

Fig. 16.7 Transaction *A* is prevented from seeing an uncommitted change at time *t2*

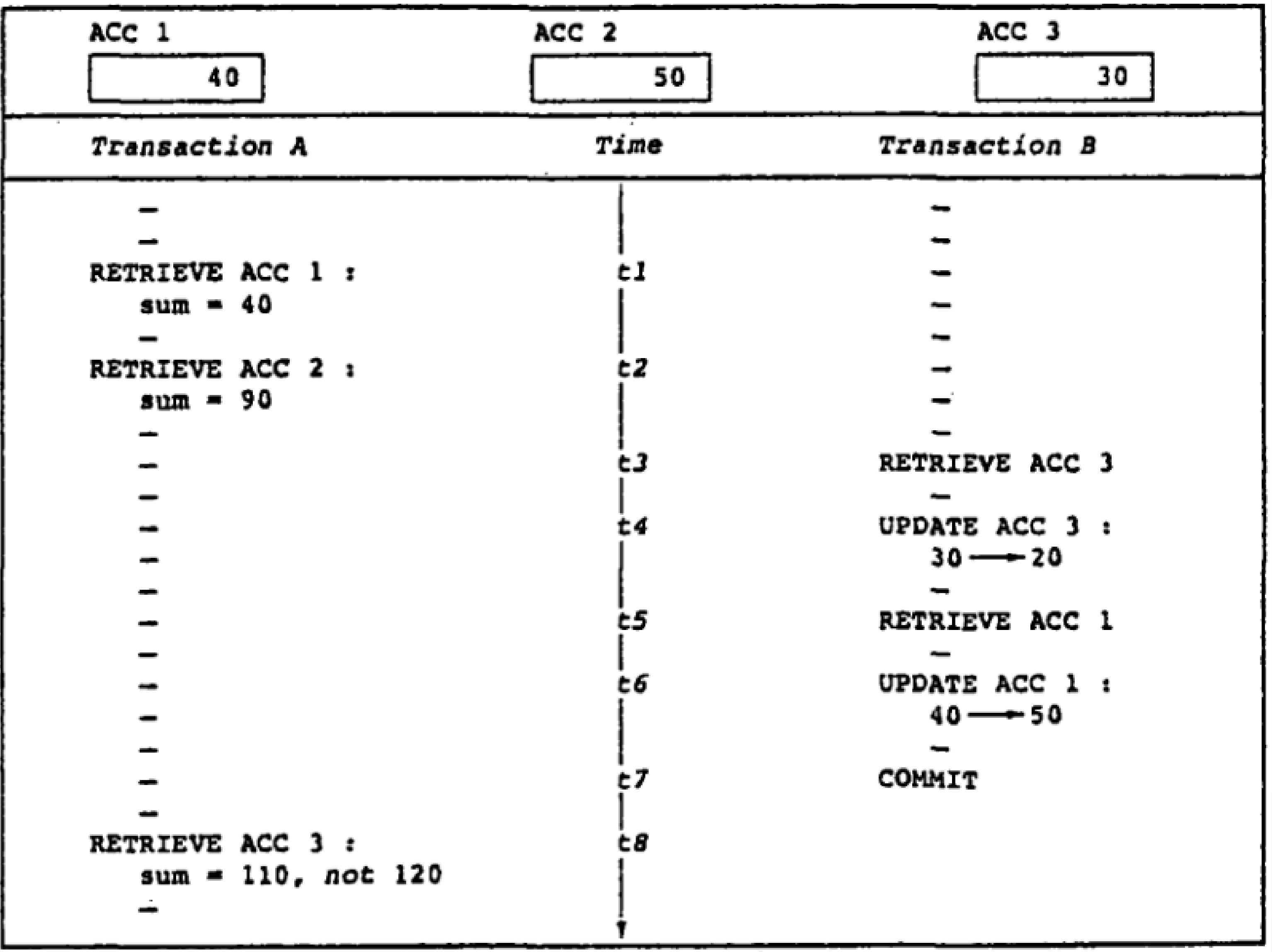


Fig. 16.4 Transaction *A* performs an inconsistent analysis

The Inconsistent Analysis Problem

An inconsistent analysis occurs when totals are calculated during interleaved updates.

The Inconsistent Analysis Problem - Revisited

ACC 1	ACC 2	ACC 3
40	50	30

Transaction A	*Time*	*Transaction B*
—		—
—		—
RETRIEVE ACC 1 :	t1	—
(acquire S lock on ACC 1)		—
sum = 40		—
—		—
RETRIEVE ACC 2 :	t2	—
(acquire S lock on ACC 2)		—
sum = 90		—
—		—
—	t3	RETRIEVE ACC 3
—		(acquire S lock on ACC 3)
—		—
—	t4	UPDATE ACC 3
—		(acquire X lock on ACC 3)
—		30 → 20
—		—
—	t5	RETRIEVE ACC 1
—		(acquire S lock on ACC 1)
—		—
—	t6	UPDATE ACC 1
—		(request X lock on ACC 1)
—		wait
RETRIEVE ACC 3 :	t7	wait
(request S lock on ACC 3)		wait
wait		wait
wait		wait

Fig. 16.9 Inconsistent analysis is prevented, but deadlock occurs at time *t7*

Extensions to Basic Two-Phase Locking Protocol

- **Two-phase locking:** Allows a transaction to lock a new data item only if that transaction has not yet unlocked any data item.
  - Not free from cascading rollbacks
- **Strict two-phase locking:** A transaction must hold all its exclusive locks till it commits/aborts.
  - Ensures recoverability of freedom from cascading rollback.
- **Rigorous two-phase locking**: A transaction must hold *all* locks till commit/abort.
  - Transactions can be serialized in the order in which they commit.
  - Ensures recoverability of freedom from cascading rollback.

Deadlock Handling

- Various locking protocols do not guard against deadlocks.
- System is **deadlocked** if there is a set of transactions such that every transaction in the set is waiting for another transaction in the set.

T_3	T_4
lock-X(B)	
read(B)	
$B := B - 50$	
write(B)	
	lock-S(A)
	read(A)
	lock-S(B)
lock-X(A)	

Deadlock Prevention Strategies

- The locking protocol may be modified to avoid deadlock by using Wait-Die Scheme and Wound-Wait Scheme
- **Wait-Die**: Transaction 2 waits if it is older than 1; otherwise it dies (rolls back)
- **Wound-Wait:** Transaction 2 wounds 1 if it is older; that is, it rolls it back
- Net effect: oldest transaction wins
- In both schemes, a rolled back transactions is restarted with its original timestamp.
  - Ensures that older transactions have precedence over newer ones, and starvation is thus avoided.

Example: Wait-Die

T1	T2
lock-S(A)	
read(A)	
	lock-S(B)
	read(B)
lock-X(B)	
	lock-X(A)

- T1 is older so it is allowed to wait.
- T2 is younger so it is roll backed.
  - Its locks are released.
  - Allows T1 to proceed.

Example: Wound-Wait

T1	T2
lock-S(A)	
read(A)	
	lock-S(B)
	read(B)
lock-X(B)	

- T1 is older so T2 is roll backed and that allows to T1 proceed.

Deadlock Prevention Strategies (Cont.)

Wait-Die Scheme — non-preemptive

- Older transaction may wait for younger one to release data item.
- Younger transactions never wait for older ones; they are rolled back instead.
- A transaction may die several times before acquiring a lock.

Wound-Wait Scheme — preemptive

- Older transaction *wounds* (forces rollback) of younger transaction instead of waiting for it.
- Younger transactions may wait for older ones.
- Fewer rollbacks than *wait-die* scheme.

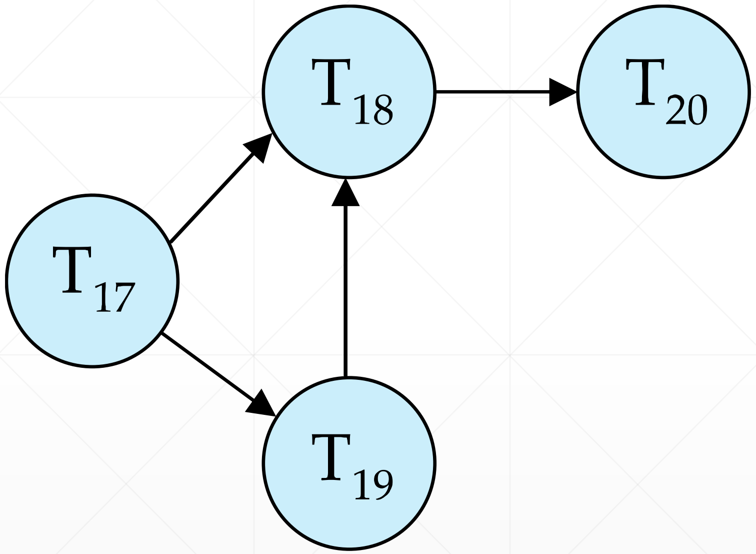
Deadlock Prevention Strategies (Cont.)

- **Timeout-Based Schemes**:
  - A transaction waits for a lock only for a specified amount of time. After that, the wait times out and the transaction is rolled back.
  - Ensures that deadlocks get resolved by timeout if they occur.
  - Simple to implement.
  - But may roll back transaction unnecessarily in absence of deadlock
    - Difficult to determine good value of the timeout interval.
  - Starvation is also possible.

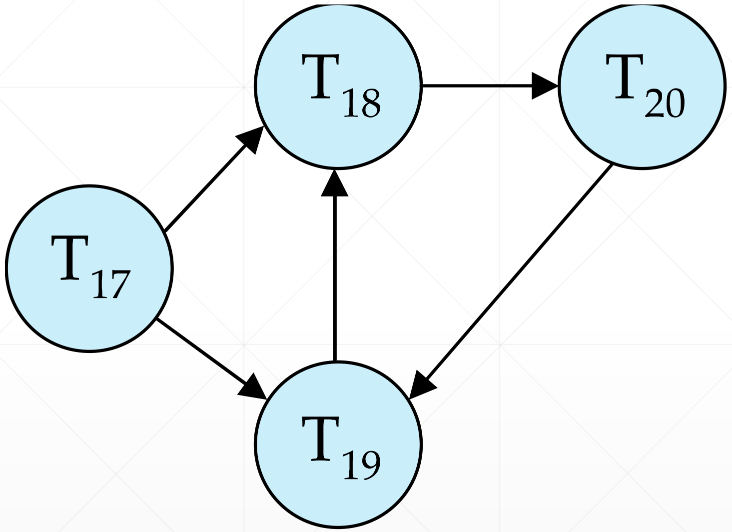
Deadlock Detection

- If deadlocks are not prevented, the system must deal with them by using a deadlock detection and recovery scheme.
- **Wait-for graph**
  - *Vertices:* transactions
  - *Edge from* $T_i \rightarrow T_j$. : if T_i is waiting for a lock held in conflicting mode by T_j
- The system is in a deadlock state if and only if the wait-for graph has a cycle.
- Invoke a deadlock-detection algorithm periodically to look for cycles.

Deadlock Detection (Cont.)



Wait-for graph without a cycle



Wait-for graph with a cycle

Deadlock!

Isolation Levels

- The isolation level indicates the degree of interaction that is allowed from other transactions during the execution of transaction.
  - **Read uncommitted:** Even uncommitted records may be read.
  - **Read committed:** Only committed records can be read, but successive reads of record may return different (but committed) values.
  - **Repeatable read:** Ensures that a read is repeatable throughout the transaction.
  - **Serializable:** Usually ensures serializable execution.
- The higher the level of isolation, the less interference, and the lower concurrency.

Dirty Read, Nonrepeatable Read and Phantom

- **Dirty read:** A transaction reads values written by another transaction that hasn't committed yet.
- **Nonrepeatable read:** A transaction reads the same object twice during execution and finds a different value the second time, although the transaction has not changed the value in the meantime.
- **Phantom read:** A transaction re-executes a query returning a set of rows that satisfy a search condition and finds that the set of rows satisfying the condition has changed as a result of another recently committed transaction.

Isolation Levels and Problems

ISOLATION LEVEL	DIRTY READ	NONREPEATABLE READ	PHANTOM READ
READ UNCOMMITTED	Y	Y	Y
READ COMMITTED	N	Y	Y
REPEATABLE READ	N	N	Y
SERIALIZABLE	N	N	N

References

- A. Silberschatz, HF. Korth, S. Sudarshan, Database System Concepts, 7th Ed., McGraw-Hill, 2019.
  - Chapter 18, https://www.db-book.com/db7/slides-dir/PPTX-dir/ch18.pptx (modified)
- Edward Sciore, Database Design and Implementation, Wiley, 2009.
  - Chapter 14
- C.J. Date, An Introduction to Database Systems, 8th Ed., 2003.
  - Chapter 16