Chapter 22

Transaction Management

Chapter 22 - Objectives

- Function and importance of transactions.
- Properties of transactions.
- Concurrency Control
  - Meaning of serializability.
  - How locking can ensure serializability.
  - Deadlock and how it can be resolved.
  - How timestamping can ensure serializability.
  - Optimistic concurrency control.
  - Granularity of locking.

Chapter 22 - Objectives

- Recovery Control
  - Some causes of database failure.
  - Purpose of transaction log file.
  - Purpose of checkpointing.
  - How to recover following database failure.
- Alternative models for long duration transactions.

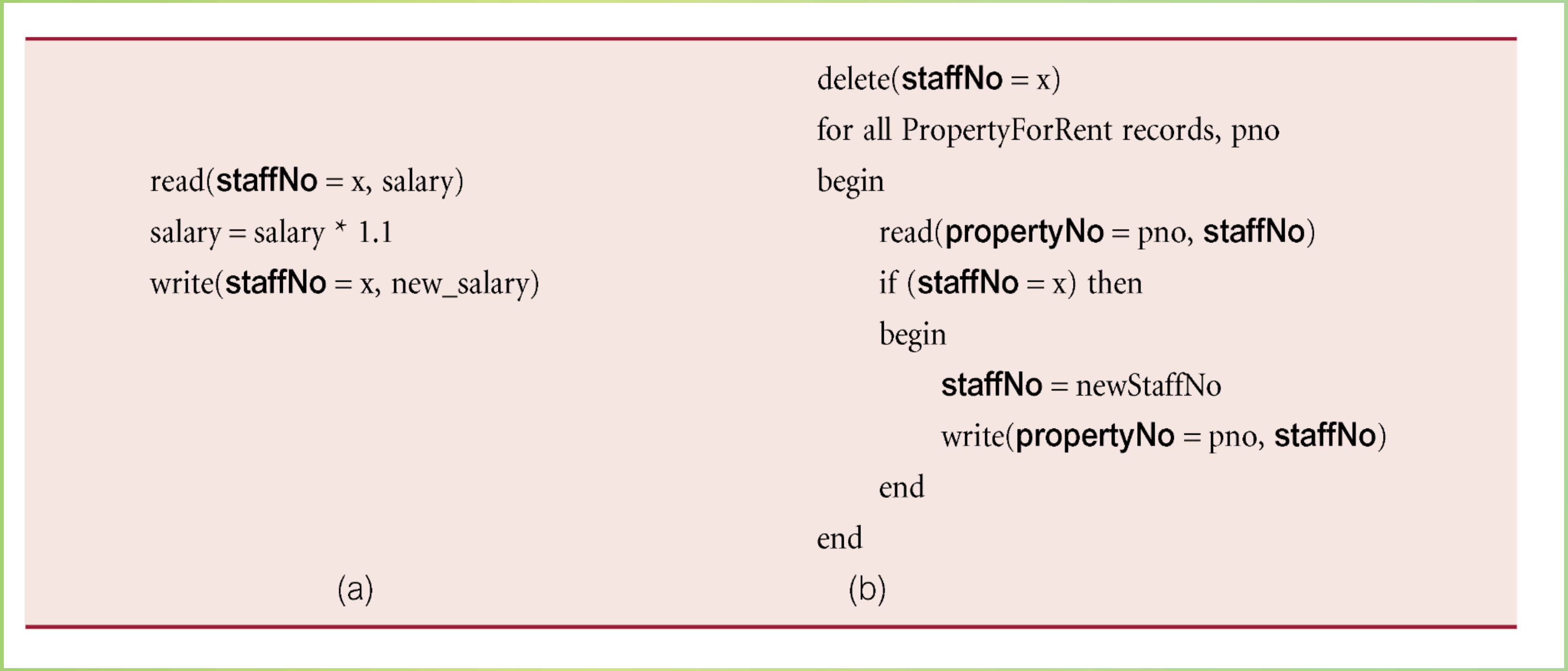
Transaction Support

<u>Transaction</u>

Action, or series of actions, carried out by user or application, which reads or updates contents of database.

- Logical unit of work on the database.
- Application program is series of transactions with non-database processing in between.
- Transforms database from one consistent state to another, although consistency may be violated during transaction.

Example Transaction



```
read(staffNo = x, salary)
salary = salary * 1.1
write(staffNo = x, new_salary)
```

(a)

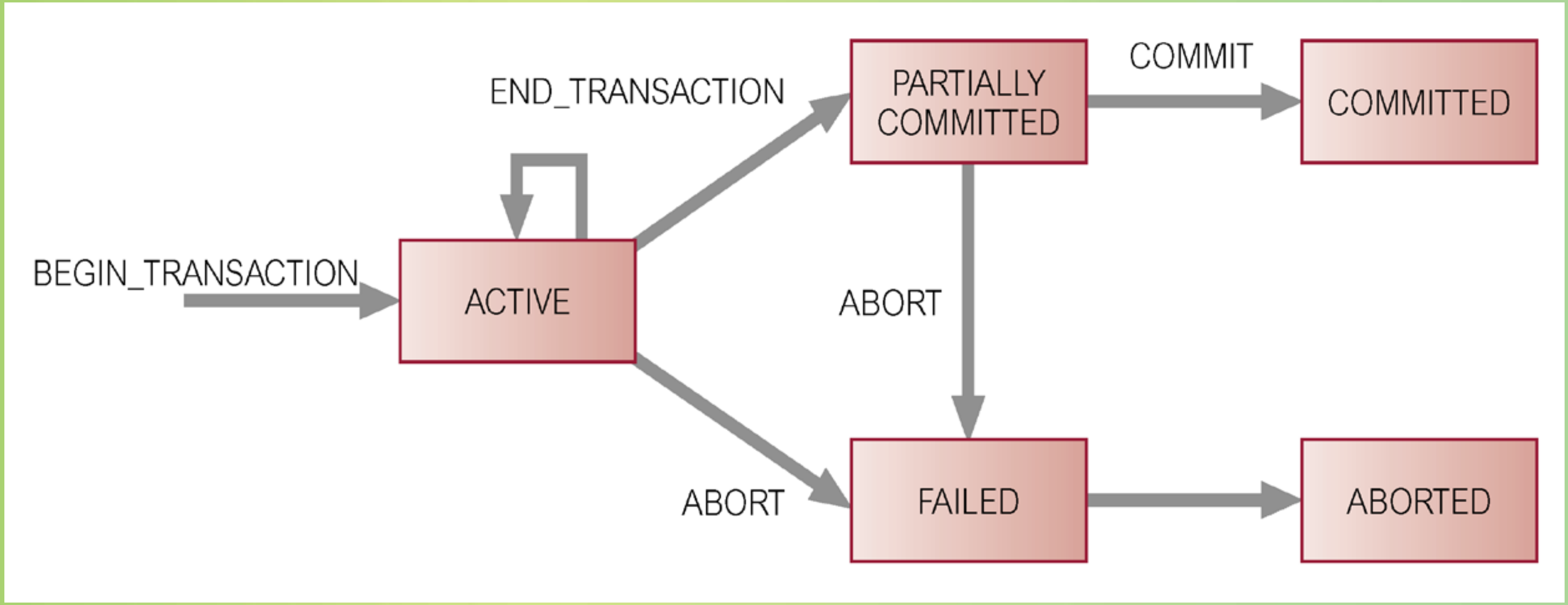
```
delete(staffNo = x)
for all PropertyForRent records, pno
begin
    read(propertyNo = pno, staffNo)
    if (staffNo = x) then
    begin
        staffNo = newStaffNo
        write(propertyNo = pno, staffNo)
    end
end
```

(b)

Transaction Support

- Can have one of two outcomes:
  - Success - transaction *commits* and database reaches a new consistent state.
  - Failure - transaction *aborts*, and database must be restored to consistent state before it started.
  - Such a transaction is *rolled back* or *undone*.
- Committed transaction cannot be aborted.
- Aborted transaction that is rolled back can be restarted later.

State Transition Diagram for Transaction



Properties of Transactions

- **Four basic *(ACID)* properties that define a transaction are:**

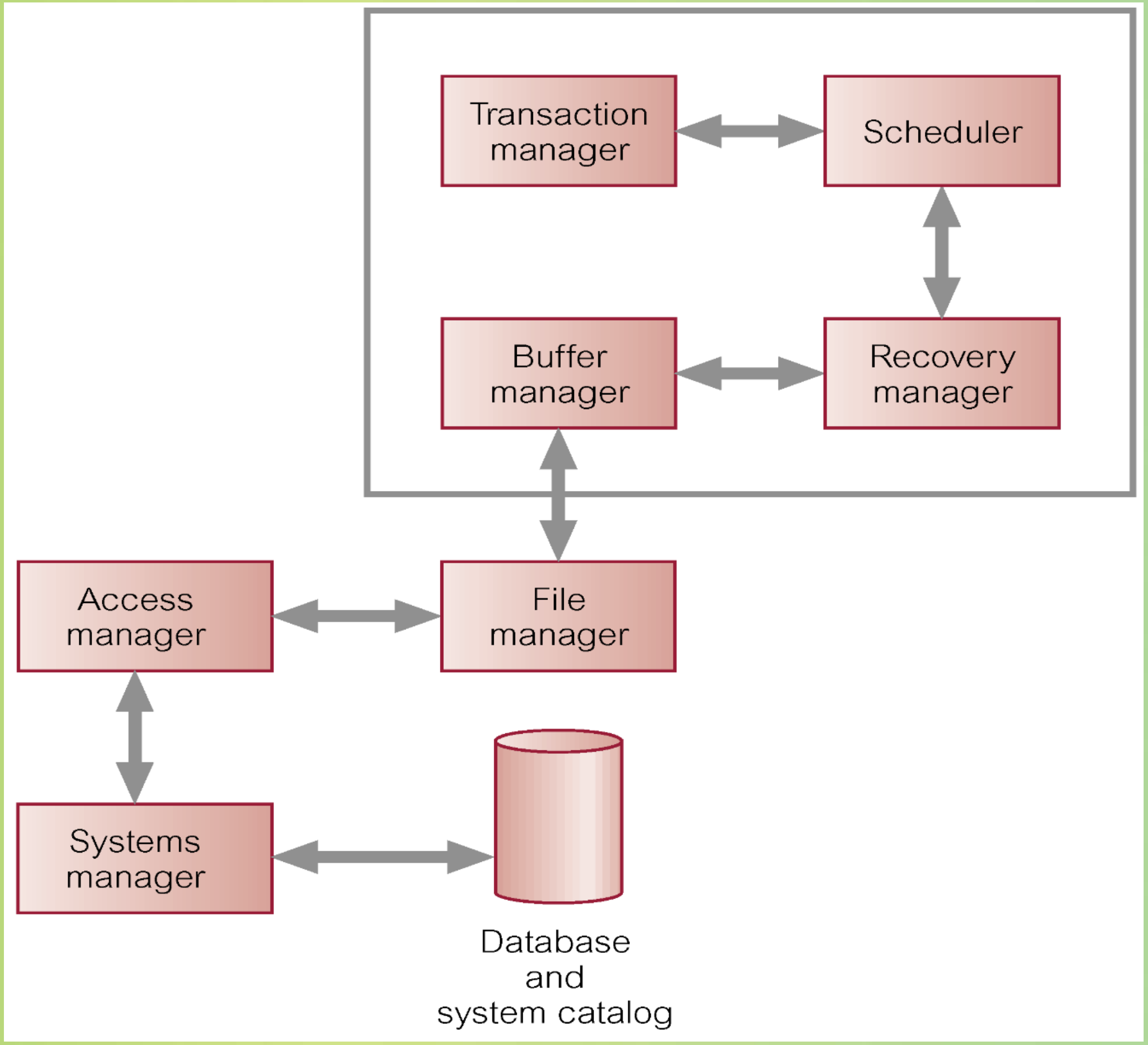
Atomicity **'All or nothing' property.**

Consistency **Must transform database from one consistent state to another.**

Isolation **Partial effects of incomplete transactions should not be visible to other transactions.**

Durability **Effects of a committed transaction are permanent and must not be lost because of later failure.**

DBMS Transaction Subsystem



Concurrency Control

Process of managing simultaneous operations on the database without having them interfere with one another.

- Prevents interference when two or more users are accessing database simultaneously and at least one is updating data.
- Although two transactions may be correct in themselves, interleaving of operations may produce an incorrect result.

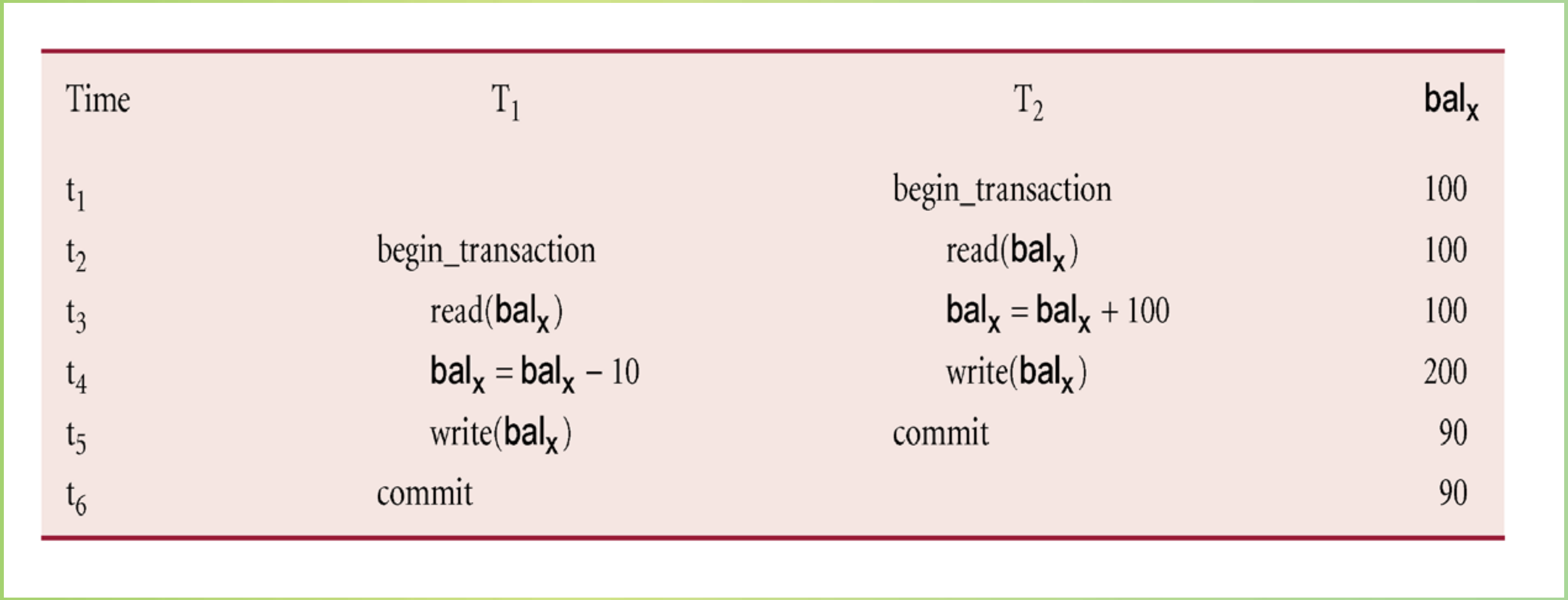
Need for Concurrency Control

- Three examples of potential problems caused by concurrency:
  - Lost update problem.
  - Uncommitted dependency problem.
  - Inconsistent analysis problem.

Lost Update Problem

- Successfully completed update is overridden by another user.
- T_1 withdrawing £10 from an account with bal_x, initially £100.
- T_2 depositing £100 into same account.
- Serially, final balance would be £190.

Lost Update Problem



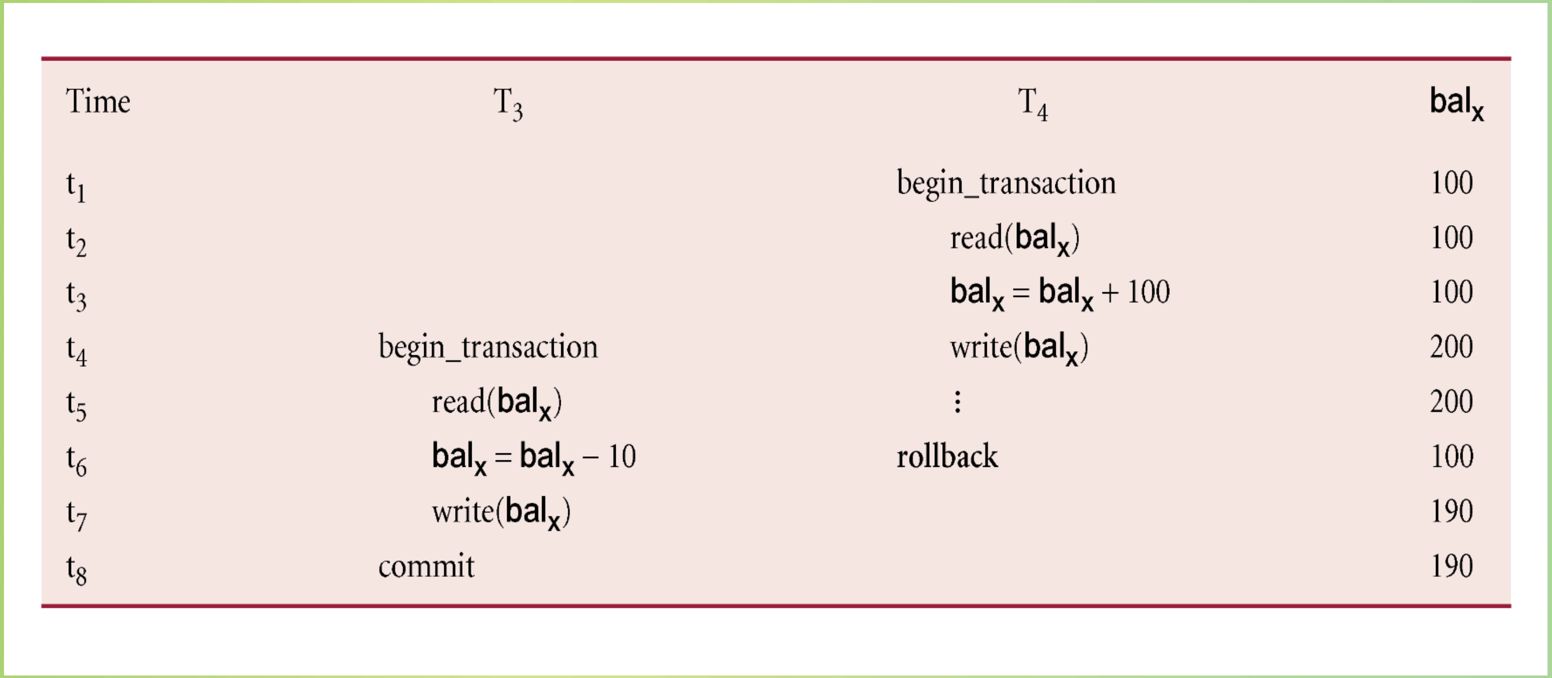
Time	T_1	T_2	bal_x
t_1		begin_transaction	100
t_2	begin_transaction	read(bal_x)	100
t_3	read(bal_x)	$bal_x = bal_x + 100$	100
t_4	$bal_x = bal_x - 10$	write(bal_x)	200
t_5	write(bal_x)	commit	90
t_6	commit		90

- **Loss of T_2's update avoided by preventing T_1 from reading bal_x until after update.**

Uncommitted Dependency Problem

- Occurs when one transaction can see intermediate results of another transaction before it has committed.
- T_4 updates bal_x to £200 but it aborts, so bal_x should be back at original value of £100.
- T_3 has read new value of bal_x (£200) and uses value as basis of £10 reduction, giving a new balance of £190, instead of £90.

Uncommitted Dependency Problem



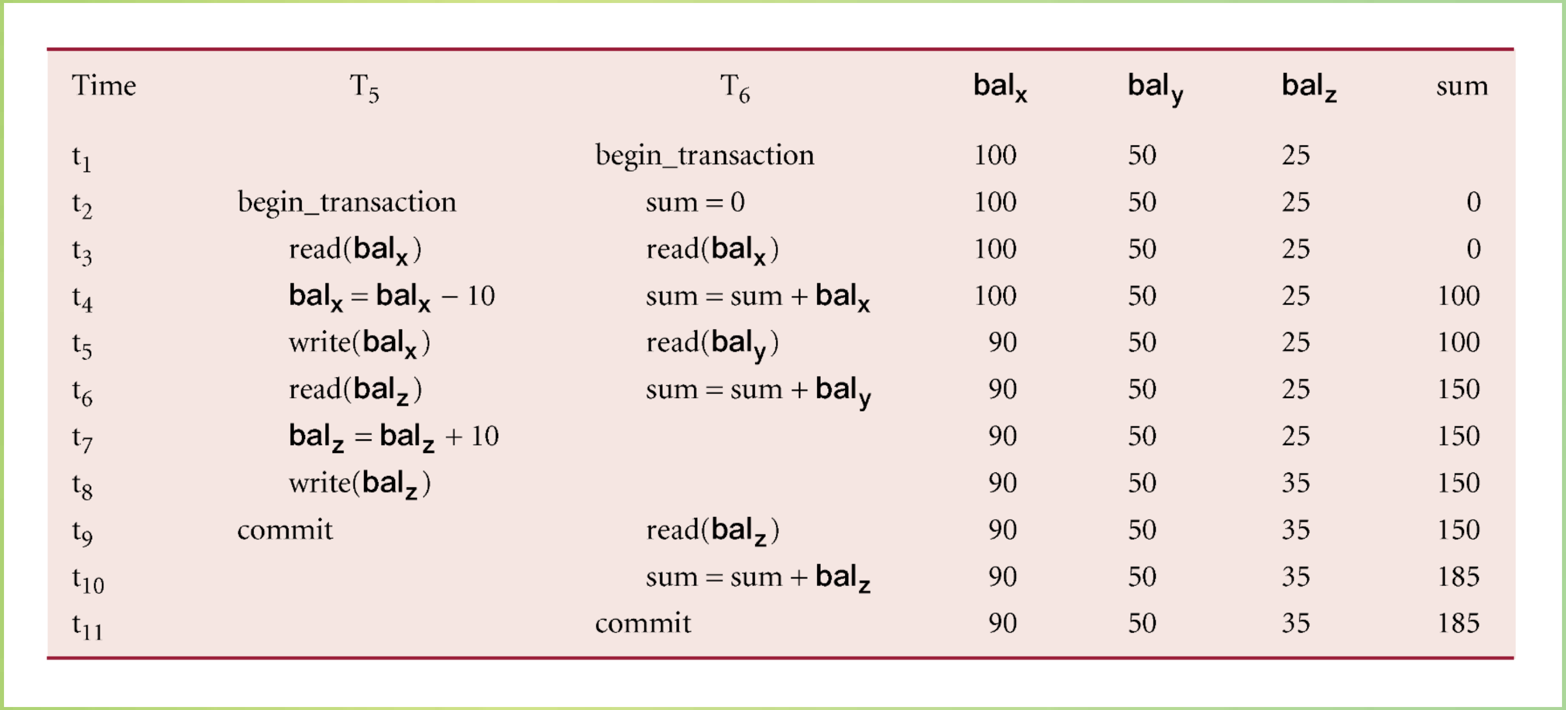
Time	T_3	T_4	bal_x
t_1		begin_transaction	100
t_2		read(bal_x)	100
t_3		$bal_x = bal_x + 100$	100
t_4	begin_transaction	write(bal_x)	200
t_5	read(bal_x)	⋮	200
t_6	$bal_x = bal_x - 10$	rollback	100
t_7	write(bal_x)		190
t_8	commit		190

- **Problem avoided by preventing T_3 from reading bal_x until after T_4 commits or aborts.**

Inconsistent Analysis Problem

- **Occurs when transaction reads several values but second transaction updates some of them during execution of first.**
- **Sometimes referred to as *dirty read* or *unrepeatable read*.**
- **T_6 is totaling balances of account x (£100), account y (£50), and account z (£25).**
- **Meantime, T_5 has transferred £10 from bal_x to bal_z, so T_6 now has wrong result (£10 too high).**

Inconsistent Analysis Problem



Time	T_5	T_6	bal_x	bal_y	bal_z	sum
t_1		begin_transaction	100	50	25	
t_2	begin_transaction	sum = 0	100	50	25	0
t_3	read(bal_x)	read(bal_x)	100	50	25	0
t_4	$bal_x = bal_x - 10$	sum = sum + bal_x	100	50	25	100
t_5	write(bal_x)	read(bal_y)	90	50	25	100
t_6	read(bal_z)	sum = sum + bal_y	90	50	25	150
t_7	$bal_z = bal_z + 10$		90	50	25	150
t_8	write(bal_z)		90	50	35	150
t_9	commit	read(bal_z)	90	50	35	150
t_{10}		sum = sum + bal_z	90	50	35	185
t_{11}		commit	90	50	35	185

- **Problem avoided by preventing T_6 from reading bal_x and bal_z until after T_5 completed updates.**

Serializability

- Objective of a concurrency control protocol is to schedule transactions in such a way as to avoid any interference.
- Could run transactions serially, but this limits degree of concurrency or parallelism in system.
- Serializability identifies those executions of transactions guaranteed to ensure consistency.

Serializability

<u>Schedule</u>

Sequence of reads/writes by set of concurrent transactions.

<u>Serial Schedule</u>

Schedule where operations of each transaction are executed consecutively without any interleaved operations from other transactions.

- No guarantee that results of all serial executions of a given set of transactions will be identical.

Nonserial Schedule

- Schedule where operations from set of concurrent transactions are interleaved.
- Objective of serializability is to find nonserial schedules that allow transactions to execute concurrently without interfering with one another.
- In other words, want to find nonserial schedules that are equivalent to *some* serial schedule. Such a schedule is called ***serializable***.

Serializability

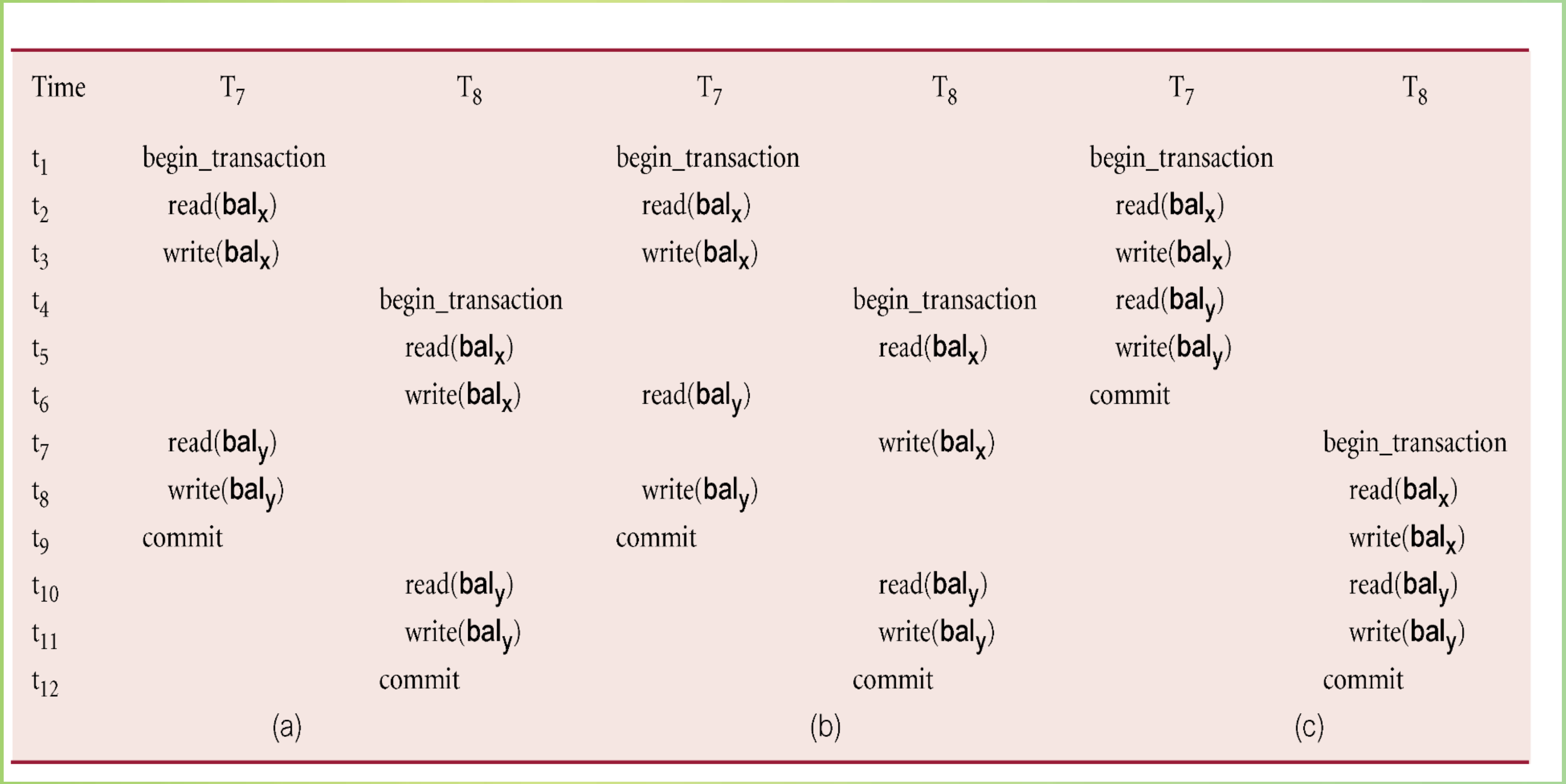
- In serializability, ordering of read/writes is important:

  (a) If two transactions only read a data item, they do not conflict and order is not important.

  (b) If two transactions either read or write separate data items, they do not conflict and order is not important.

  (c) If one transaction writes a data item and another reads or writes same data item, order of execution is important.

Example of Conflict Serializability



Time	T_7	T_8	T_7	T_8	T_7	T_8
t_1	begin_transaction		begin_transaction		begin_transaction	
t_2	read(bal_x)		read(bal_x)		read(bal_x)	
t_3	write(bal_x)		write(bal_x)		write(bal_x)	
t_4		begin_transaction		begin_transaction	read(bal_y)	
t_5		read(bal_x)		read(bal_x)	write(bal_y)	
t_6		write(bal_x)	read(bal_y)		commit	
t_7	read(bal_y)			write(bal_x)		begin_transaction
t_8	write(bal_y)		write(bal_y)			read(bal_x)
t_9	commit		commit			write(bal_x)
t_{10}		read(bal_y)		read(bal_y)		read(bal_y)
t_{11}		write(bal_y)		write(bal_y)		write(bal_y)
t_{12}		commit		commit		commit
	(a)		(b)		(c)	

Serializability

- **Conflict serializable schedule orders any conflicting operations in same way as some serial execution.**
- **Under *constrained write rule* (transaction updates data item based on its old value, which is first read), use *precedence graph* to test for serializability.**

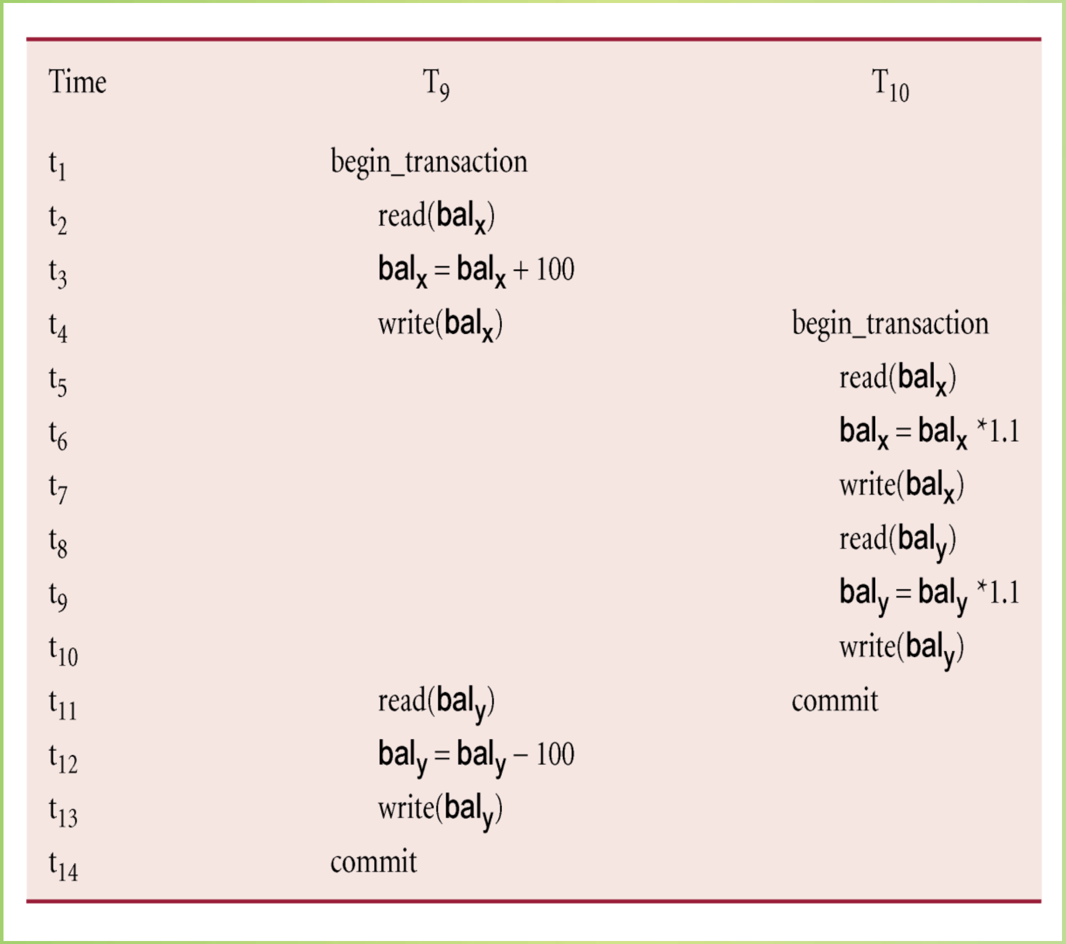
Precedence Graph

- Create:
  - node for each transaction;
  - a directed edge $T_i \rightarrow T_j$, if T_j reads the value of an item written by T_I;
  - a directed edge $T_i \rightarrow T_j$, if T_j writes a value into an item after it has been read by T_i.
- If precedence graph contains cycle schedule is not conflict serializable.

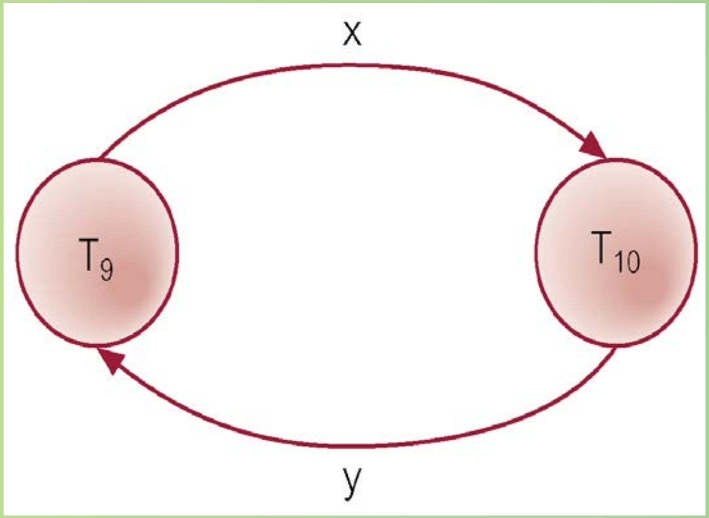
Example - Non-conflict serializable schedule

- T_9 is transferring £100 from one account with balance bal_x to another account with balance bal_y.
- T_{10} is increasing balance of these two accounts by 10%.
- Precedence graph has a cycle and so is not serializable.

Example - Non-conflict serializable schedule



Time	T_9	T_{10}
t_1	begin_transaction	
t_2	read(bal_x)	
t_3	$bal_x = bal_x + 100$	
t_4	write(bal_x)	begin_transaction
t_5		read(bal_x)
t_6		$bal_x = bal_x *1.1$
t_7		write(bal_x)
t_8		read(bal_y)
t_9		$bal_y = bal_y *1.1$
t_{10}		write(bal_y)
t_{11}	read(bal_y)	commit
t_{12}	$bal_y = bal_y - 100$	
t_{13}	write(bal_y)	
t_{14}	commit	



$$\begin{tikzcd}
T_9 \arrow[r, bend left, "x"] & T_{10} \arrow[l, bend left, "y"]
\end{tikzcd}$$

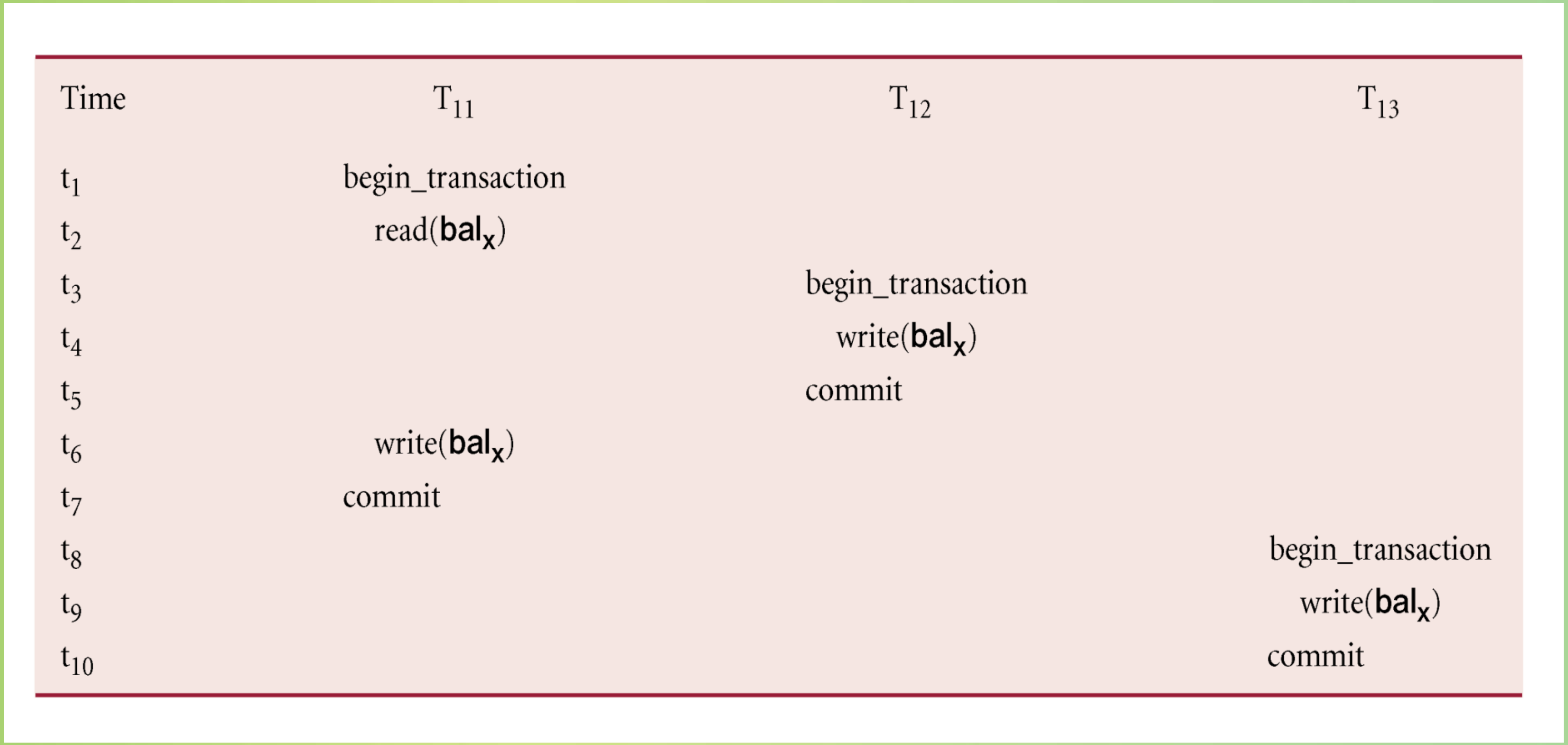
View Serializability

- **Offers less stringent definition of schedule equivalence than conflict serializability.**
- **Two schedules S_1 and S_2 are view equivalent if:**
  - **For each data item x, if T_i reads initial value of x in S_1, T_i must also read initial value of x in S_2.**
  - **For each read on x by T_i in S_1, if value read by T_i is written by T_j, T_i must also read value of x produced by T_j in S_2.**
  - **For each data item x, if last write on x performed by T_i in S_1, same transaction must perform final write on x in S_2.**

View Serializability

- Schedule is view serializable if it is view equivalent to a serial schedule.
- Every conflict serializable schedule is view serializable, although converse is not true.
- It can be shown that any view serializable schedule that is not conflict serializable contains one or more blind writes.
- In general, testing whether schedule is serializable is NP-complete.

Example - View Serializable schedule



Time	T_{11}	T_{12}	T_{13}
t_1	begin_transaction		
t_2	read(**bal$_x$**)		
t_3		begin_transaction	
t_4		write(**bal$_x$**)	
t_5		commit	
t_6	write(**bal$_x$**)		
t_7	commit		
t_8			begin_transaction
t_9			write(**bal$_x$**)
t_{10}			commit

Recoverability

- Serializability identifies schedules that maintain database consistency, assuming no transaction fails.
- Could also examine recoverability of transactions within schedule.
- If transaction fails, atomicity requires effects of transaction to be undone.
- Durability states that once transaction commits, its changes cannot be undone (without running another, compensating, transaction).

Recoverable Schedule

A schedule where, for each pair of transactions T_i and T_j, if T_j reads a data item previously written by T_i, then the commit operation of T_i precedes the commit operation of T_j.

Concurrency Control Techniques

- **Two basic concurrency control techniques:**
  - **Locking,**
  - **Timestamping.**
- **Both are conservative approaches: delay transactions in case they conflict with other transactions.**
- **Optimistic methods assume conflict is rare and only check for conflicts at commit.**

Locking

Transaction uses locks to deny access to other transactions and so prevent incorrect updates.

- Most widely used approach to ensure serializability.
- Generally, a transaction must claim a *shared* (*read*) or *exclusive* (*write*) lock on a data item before read or write.
- Lock prevents another transaction from modifying item or even reading it, in the case of a write lock.

Locking - Basic Rules

- **If transaction has shared lock on item, can read but not update item.**
- **If transaction has exclusive lock on item, can both read and update item.**
- **Reads cannot conflict, so more than one transaction can hold shared locks simultaneously on same item.**
- **Exclusive lock gives transaction exclusive access to that item.**

Locking - Basic Rules

- **Some systems allow transaction to upgrade read lock to an exclusive lock, or downgrade exclusive lock to a shared lock.**

Example - Incorrect Locking Schedule

- **For two transactions above, a valid schedule using these rules is:**

S = {write_lock(T_9, bal_x), read(T_9, bal_x), write(T_9, bal_x), unlock(T_9, bal_x), write_lock(T_{10}, bal_x), read(T_{10}, bal_x), write(T_{10}, bal_x), unlock(T_{10}, bal_x), write_lock(T_{10}, bal_y), read(T_{10}, bal_y), write(T_{10}, bal_y), unlock(T_{10}, bal_y), commit(T_{10}), write_lock(T_9, bal_y), read(T_9, bal_y), write(T_9, bal_y), unlock(T_9, bal_y), commit(T_9) }

Example - Incorrect Locking Schedule

- If at start, $bal_x = 100$, $bal_y = 400$, result should be:
  - $bal_x = 220$, $bal_y = 330$, if T_9 executes before T_{10}, or
  - $bal_x = 210$, $bal_y = 340$, if T_{10} executes before T_9.
- However, result gives $bal_x = 220$ & $bal_y = 340$.
- S is not a serializable schedule.

Example - Incorrect Locking Schedule

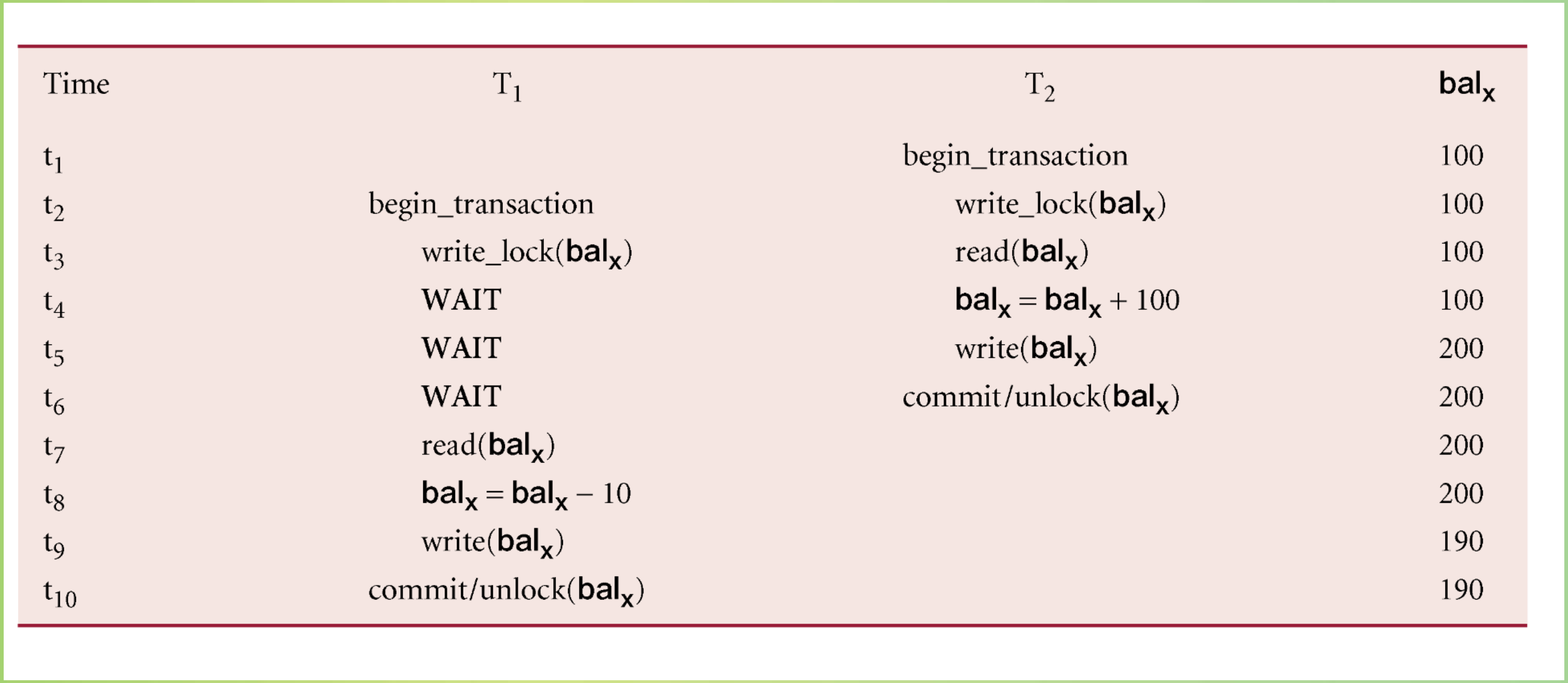
- **Problem is that transactions release locks too soon, resulting in loss of total isolation and atomicity.**
- **To guarantee serializability, need an additional protocol concerning the positioning of lock and unlock operations in every transaction.**

Two-Phase Locking (2PL)

Transaction follows 2PL protocol if all locking operations precede first unlock operation in the transaction.

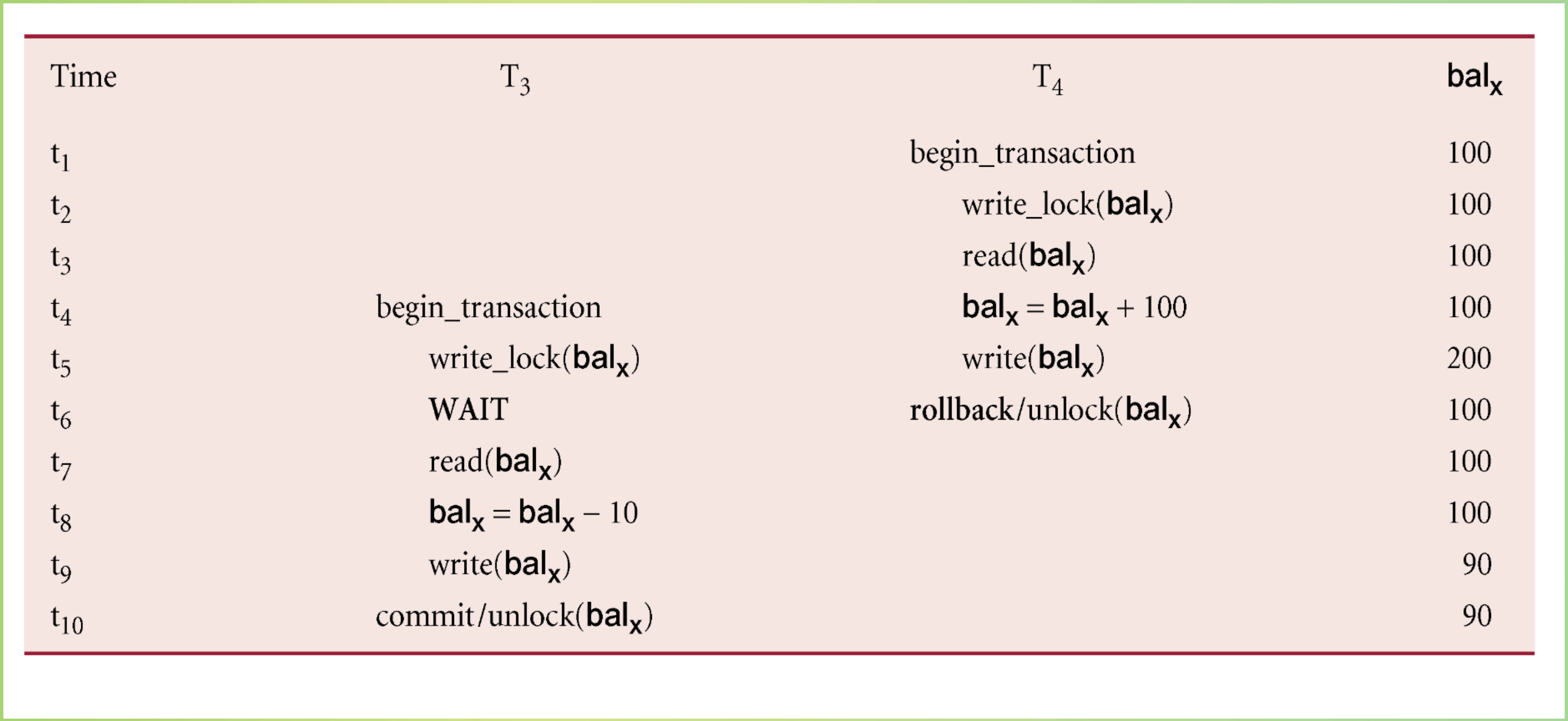
- Two phases for transaction:
  - Growing phase - acquires all locks but cannot release any locks.
  - Shrinking phase - releases locks but cannot acquire any new locks.

Preventing Lost Update Problem using 2PL



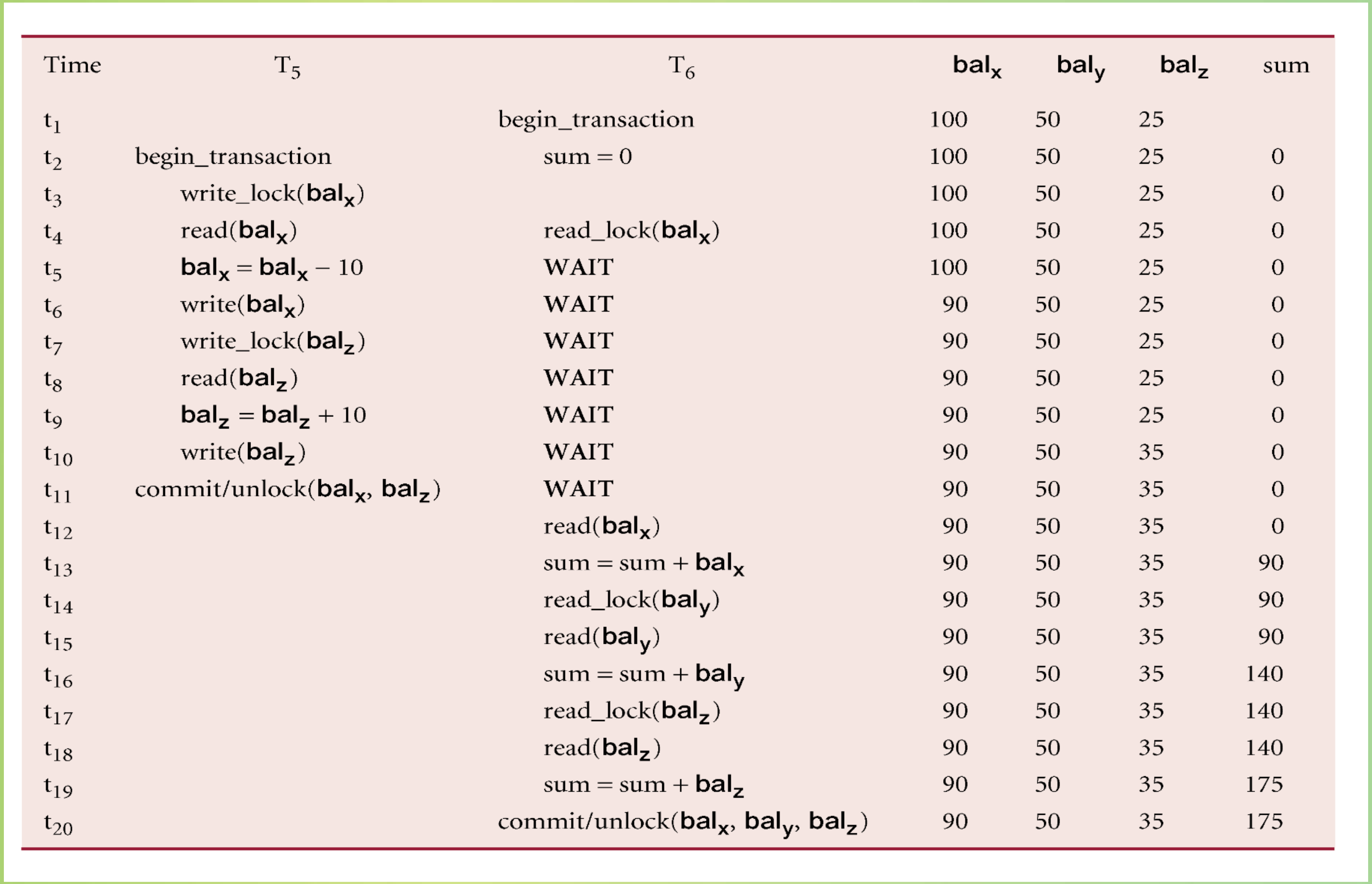
Time	T_1	T_2	bal_x
t_1		begin_transaction	100
t_2	begin_transaction	write_lock(bal_x)	100
t_3	write_lock(bal_x)	read(bal_x)	100
t_4	WAIT	$bal_x = bal_x + 100$	100
t_5	WAIT	write(bal_x)	200
t_6	WAIT	commit/unlock(bal_x)	200
t_7	read(bal_x)		200
t_8	$bal_x = bal_x - 10$		200
t_9	write(bal_x)		190
t_{10}	commit/unlock(bal_x)		190

Preventing Uncommitted Dependency Problem using 2PL



Time	T_3	T_4	bal_x
t_1		begin_transaction	100
t_2		write_lock(bal_x)	100
t_3		read(bal_x)	100
t_4	begin_transaction	$bal_x = bal_x + 100$	100
t_5	write_lock(bal_x)	write(bal_x)	200
t_6	WAIT	rollback/unlock(bal_x)	100
t_7	read(bal_x)		100
t_8	$bal_x = bal_x - 10$		100
t_9	write(bal_x)		90
t_{10}	commit/unlock(bal_x)		90

Preventing Inconsistent Analysis Problem using 2PL

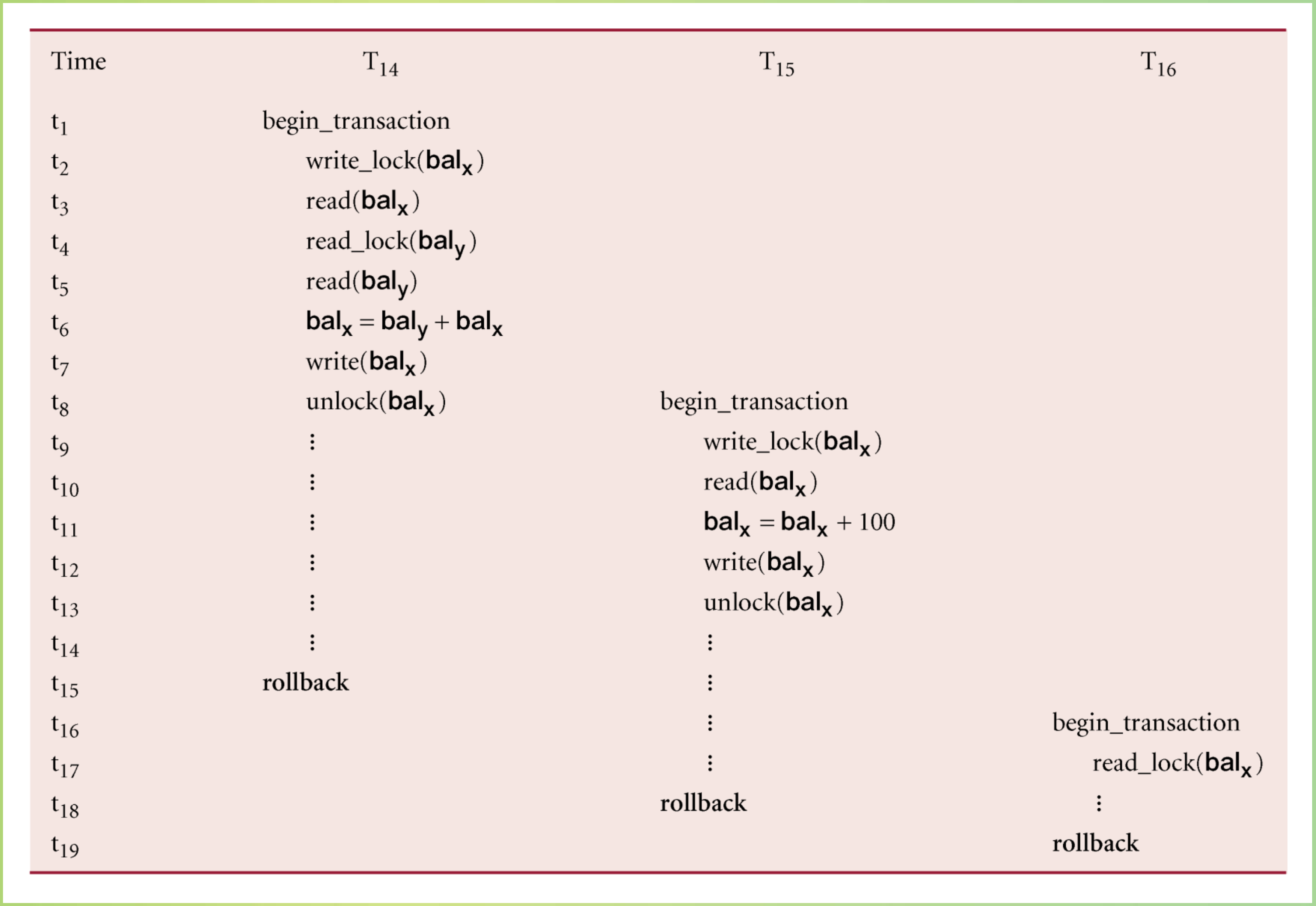


Time	T_5	T_6	bal_x	bal_y	bal_z	sum
t_1		begin_transaction	100	50	25	
t_2	begin_transaction	sum = 0	100	50	25	0
t_3	write_lock(bal_x)		100	50	25	0
t_4	read(bal_x)	read_lock(bal_x)	100	50	25	0
t_5	$bal_x = bal_x - 10$	WAIT	100	50	25	0
t_6	write(bal_x)	WAIT	90	50	25	0
t_7	write_lock(bal_z)	WAIT	90	50	25	0
t_8	read(bal_z)	WAIT	90	50	25	0
t_9	$bal_z = bal_z + 10$	WAIT	90	50	25	0
t_{10}	write(bal_z)	WAIT	90	50	35	0
t_{11}	commit/unlock(bal_x, bal_z)	WAIT	90	50	35	0
t_{12}		read(bal_x)	90	50	35	0
t_{13}		sum = sum + bal_x	90	50	35	90
t_{14}		read_lock(bal_y)	90	50	35	90
t_{15}		read(bal_y)	90	50	35	90
t_{16}		sum = sum + bal_y	90	50	35	140
t_{17}		read_lock(bal_z)	90	50	35	140
t_{18}		read(bal_z)	90	50	35	140
t_{19}		sum = sum + bal_z	90	50	35	175
t_{20}		commit/unlock(bal_x, bal_y, bal_z)	90	50	35	175

Cascading Rollback

- If *every* transaction in a schedule follows 2PL, schedule is serializable.
- However, problems can occur with interpretation of when locks can be released.

Cascading Rollback



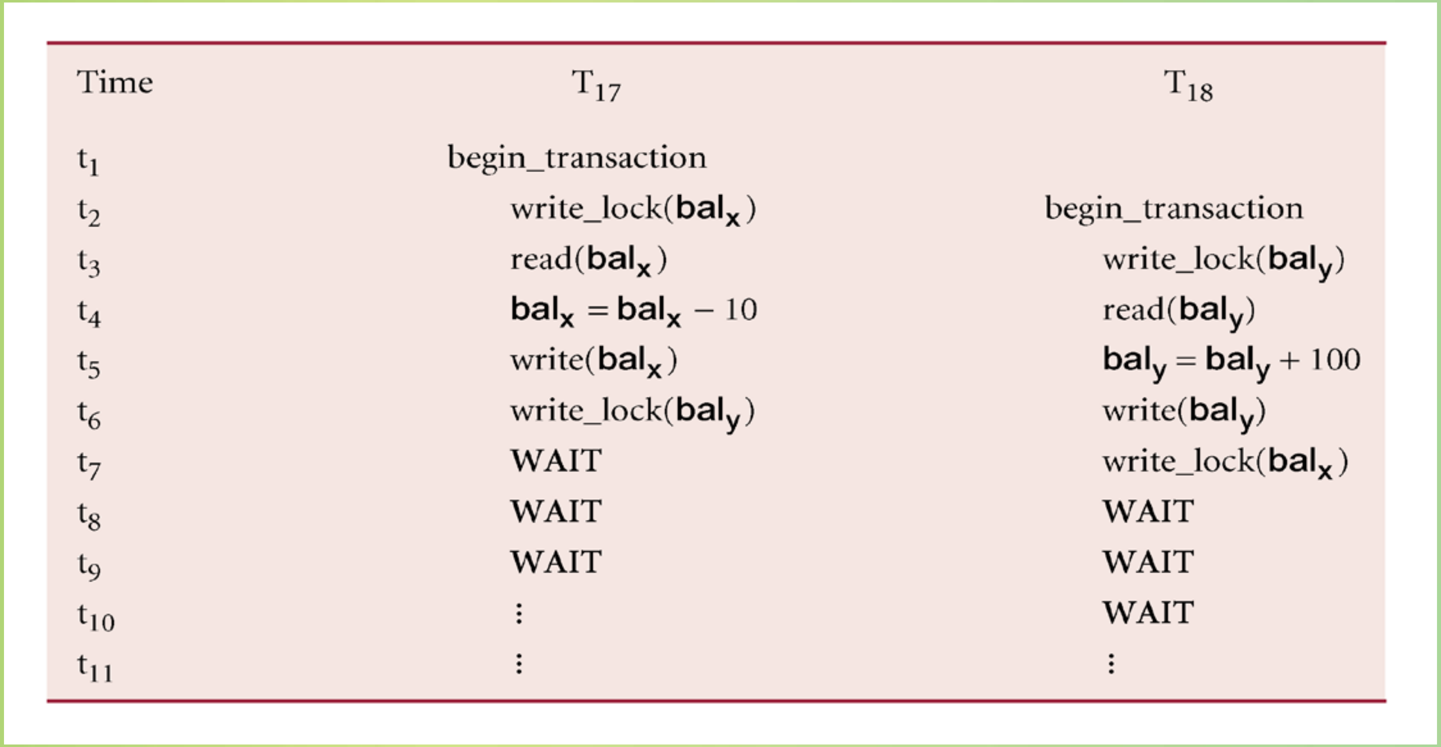
Time	T_{14}	T_{15}	T_{16}
t_1	begin_transaction		
t_2	write_lock($\mathbf{bal_x}$)		
t_3	read($\mathbf{bal_x}$)		
t_4	read_lock($\mathbf{bal_y}$)		
t_5	read($\mathbf{bal_y}$)		
t_6	$\mathbf{bal_x} = \mathbf{bal_y} + \mathbf{bal_x}$		
t_7	write($\mathbf{bal_x}$)		
t_8	unlock($\mathbf{bal_x}$)	begin_transaction	
t_9	⋮	write_lock($\mathbf{bal_x}$)	
t_{10}	⋮	read($\mathbf{bal_x}$)	
t_{11}	⋮	$\mathbf{bal_x} = \mathbf{bal_x} + 100$	
t_{12}	⋮	write($\mathbf{bal_x}$)	
t_{13}	⋮	unlock($\mathbf{bal_x}$)	
t_{14}	⋮	⋮	
t_{15}	rollback	⋮	
t_{16}		⋮	begin_transaction
t_{17}		⋮	read_lock($\mathbf{bal_x}$)
t_{18}		rollback	⋮
t_{19}			rollback

Cascading Rollback

- **Transactions conform to 2PL.**
- **T_{14} aborts.**
- **Since T_{15} is dependent on T_{14}, T_{15} must also be rolled back. Since T_{16} is dependent on T_{15}, it too must be rolled back.**
- **This is called *cascading rollback*.**
- **To prevent this with 2PL, leave release of *all* locks until end of transaction.**

Deadlock

An impasse that may result when two (or more) transactions are each waiting for locks held by the other to be released.



Time	T_{17}	T_{18}
t_1	begin_transaction	
t_2	write_lock(**bal_x**)	begin_transaction
t_3	read(**bal_x**)	write_lock(**bal_y**)
t_4	$bal_x = bal_x - 10$	read(**bal_y**)
t_5	write(**bal_x**)	$bal_y = bal_y + 100$
t_6	write_lock(**bal_y**)	write(**bal_y**)
t_7	WAIT	write_lock(**bal_x**)
t_8	WAIT	WAIT
t_9	WAIT	WAIT
t_{10}	⋮	WAIT
t_{11}	⋮	⋮

Deadlock

- Only one way to break deadlock: abort one or more of the transactions.
- Deadlock should be transparent to user, so DBMS should restart transaction(s).
- However, in practice DBMS cannot restart aborted transaction since it is unaware of transaction logic even if it was aware of the transaction history (unless there is no user input in the transaction or the input is not a function of the database state).

Deadlock

- **Three general techniques for handling deadlock:**
  - **Timeouts.**
  - **Deadlock prevention.**
  - **Deadlock detection and recovery.**

Timeouts

- Transaction that requests lock will only wait for a system-defined period of time.
- If lock has not been granted within this period, lock request times out.
- In this case, DBMS assumes transaction may be deadlocked, even though it may not be, and it aborts and automatically restarts the transaction.

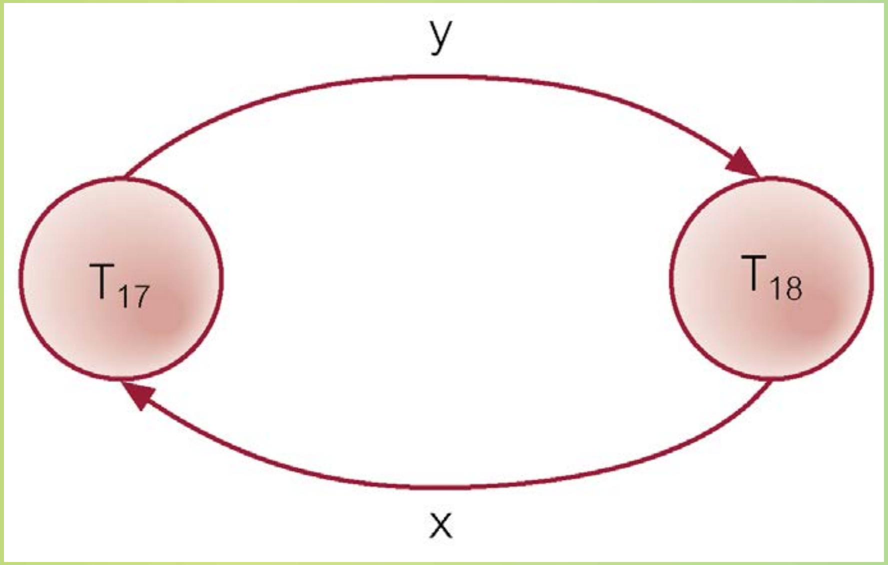
Deadlock Prevention

- **DBMS looks ahead to see if transaction would cause deadlock and never allows deadlock to occur.**
- **Could order transactions using transaction timestamps:**
  - **Wait-Die - only an older transaction can wait for younger one, otherwise transaction is aborted (*dies*) and restarted with same timestamp.**
  - **Wound-Wait - only a younger transaction can wait for an older one. If older transaction requests lock held by younger one, younger one is aborted (*wounded*).**

Deadlock Detection and Recovery

- DBMS allows deadlock to occur but recognizes it and breaks it.
- Usually handled by construction of wait-for graph (WFG) showing transaction dependencies:
  - Create a node for each transaction.
  - Create edge T_i -> T_j, if T_i waiting to lock item locked by T_j.
- Deadlock exists if and only if WFG contains cycle.
- WFG is created at regular intervals.

Example - Wait-For-Graph (WFG)



Recovery from Deadlock Detection

- Several issues:
  - choice of deadlock victim;
  - how far to roll a transaction back;
  - avoiding starvation.

Timestamping

- Transactions ordered globally so that older transactions, transactions with *smaller* timestamps, get priority in the event of conflict.
- Conflict is resolved by rolling back and restarting transaction.
- No locks so no deadlock.

Timestamping

<u>Timestamp</u>

A unique identifier created by DBMS that indicates relative starting time of a transaction.

- Can be generated by using system clock at time transaction started, or by incrementing a logical counter every time a new transaction starts.

Timestamping

- Read/write proceeds only if *last update on that data item* was carried out by an older transaction.
- Otherwise, transaction requesting read/write is restarted and given a new timestamp.
- Also timestamps for data items:
  - read-timestamp - timestamp of last transaction to read item;
  - write-timestamp - timestamp of last transaction to write item.

Timestamping - Read(x)

- Consider a transaction T with timestamp ts(T):

<u>**ts(T) < write_timestamp(x)**</u>

- x already updated by younger (later) transaction.
- Transaction must be aborted and restarted with a new timestamp.

<u>**ts(T) < read_timestamp(x)**</u>

- x already read by younger transaction.
- Roll back transaction and restart it using a later timestamp.

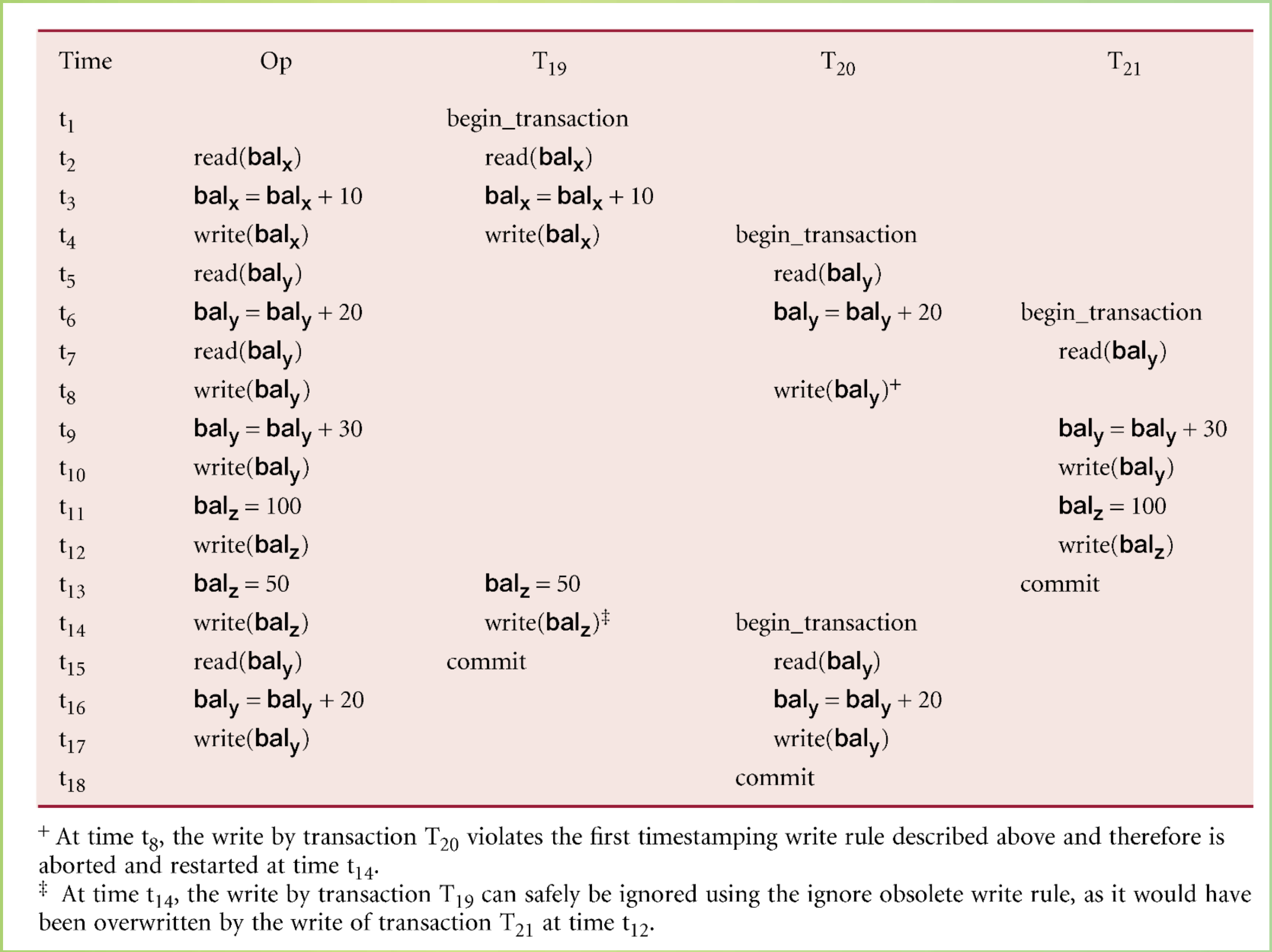
Timestamping - Write(x)

ts(T) < write_timestamp(x)

- x already written by younger transaction.
- Write can safely be ignored - *ignore obsolete write* rule.

- Otherwise, operation is accepted and executed.

Example – Basic Timestamp Ordering

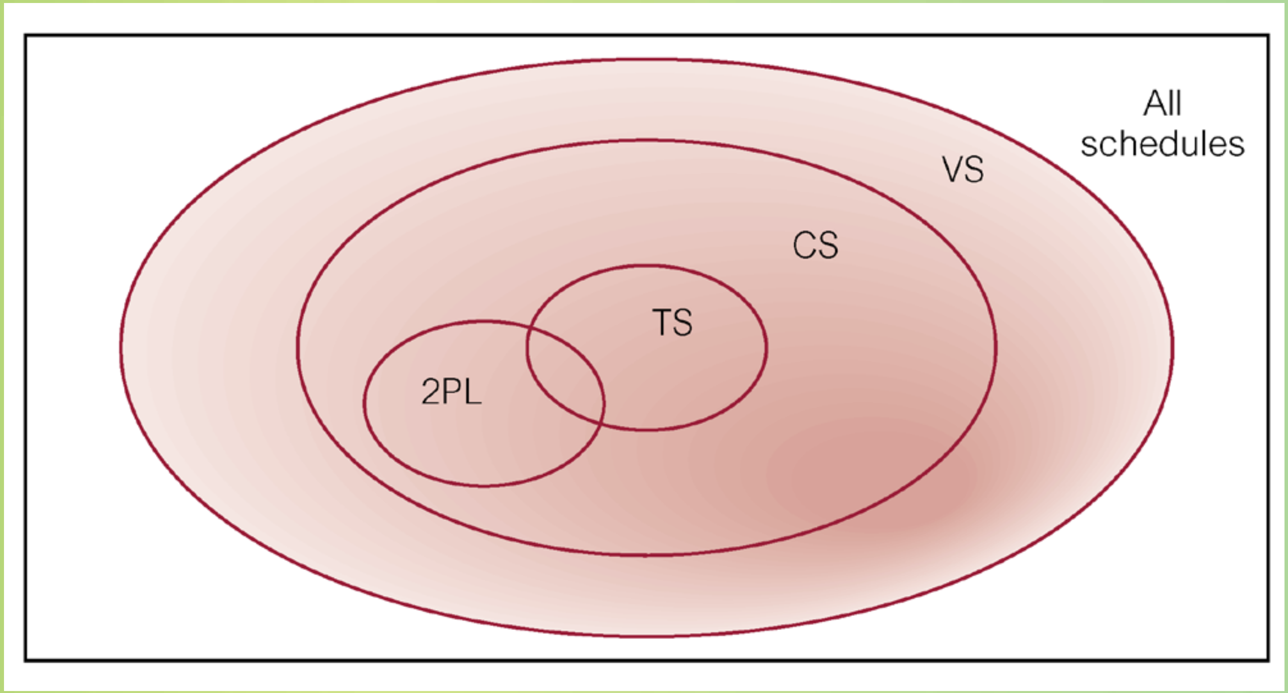


Time	Op	T_{19}	T_{20}	T_{21}
t_1		begin_transaction		
t_2	read($\mathbf{bal_x}$)	read($\mathbf{bal_x}$)		
t_3	$\mathbf{bal_x} = \mathbf{bal_x} + 10$	$\mathbf{bal_x} = \mathbf{bal_x} + 10$		
t_4	write($\mathbf{bal_x}$)	write($\mathbf{bal_x}$)	begin_transaction	
t_5	read($\mathbf{bal_y}$)		read($\mathbf{bal_y}$)	
t_6	$\mathbf{bal_y} = \mathbf{bal_y} + 20$		$\mathbf{bal_y} = \mathbf{bal_y} + 20$	begin_transaction
t_7	read($\mathbf{bal_y}$)			read($\mathbf{bal_y}$)
t_8	write($\mathbf{bal_y}$)		write($\mathbf{bal_y}$)$^+$	
t_9	$\mathbf{bal_y} = \mathbf{bal_y} + 30$			$\mathbf{bal_y} = \mathbf{bal_y} + 30$
t_{10}	write($\mathbf{bal_y}$)			write($\mathbf{bal_y}$)
t_{11}	$\mathbf{bal_z} = 100$			$\mathbf{bal_z} = 100$
t_{12}	write($\mathbf{bal_z}$)			write($\mathbf{bal_z}$)
t_{13}	$\mathbf{bal_z} = 50$	$\mathbf{bal_z} = 50$		commit
t_{14}	write($\mathbf{bal_z}$)	write($\mathbf{bal_z}$)‡	begin_transaction	
t_{15}	read($\mathbf{bal_y}$)	commit	read($\mathbf{bal_y}$)	
t_{16}	$\mathbf{bal_y} = \mathbf{bal_y} + 20$		$\mathbf{bal_y} = \mathbf{bal_y} + 20$	
t_{17}	write($\mathbf{bal_y}$)		write($\mathbf{bal_y}$)	
t_{18}			commit	

$^+$ At time t_8, the write by transaction T_{20} violates the first timestamping write rule described above and therefore is aborted and restarted at time t_{14}.

‡ At time t_{14}, the write by transaction T_{19} can safely be ignored using the ignore obsolete write rule, as it would have been overwritten by the write of transaction T_{21} at time t_{12}.

Comparison of Methods



Multiversion Timestamp Ordering

- Versioning of data can be used to increase concurrency.
- Basic timestamp ordering protocol assumes only one version of data item exists, and so only one transaction can access data item at a time.
- Can allow multiple transactions to read and write different versions of same data item, and ensure each transaction sees consistent set of versions for all data items it accesses.

Multiversion Timestamp Ordering

- In multiversion concurrency control, each write operation creates new version of data item while retaining old version.
- When transaction attempts to read data item, system selects the correct version of the data item according to the timestamp of the requesting transaction.
- Versions can be deleted once they are no longer required.

Optimistic Techniques

- Based on assumption that conflict is rare and more efficient to let transactions proceed without delays to ensure serializability.
- At commit, check is made to determine whether conflict has occurred.
- If there is a conflict, transaction must be rolled back and restarted.
- Potentially allows greater concurrency than traditional protocols.

Optimistic Techniques

- Three phases:
  - Read
  - Validation
  - Write

Optimistic Techniques - Read Phase

- **Extends from start until immediately before commit.**
- **Transaction reads values from database and stores them in local variables. Updates are applied to a local copy of the data.**

Optimistic Techniques - Validation Phase

- Follows the read phase.
- For read-only transaction, checks that data read are still current values. If no interference, transaction is committed, else aborted and restarted.
- For update transaction, checks transaction leaves database in a consistent state, with serializability maintained.

Optimistic Techniques - Write Phase

- Follows successful validation phase for update transactions.

- Updates made to local copy are applied to the database.

Granularity of Data Items

- Size of data items chosen as unit of protection by concurrency control protocol.
- Ranging from coarse to fine:
  - The entire database.
  - A file.
  - A page (or area or database spaced).
  - A record.
  - A field value of a record.

Granularity of Data Items

- Tradeoff:
  - coarser, the lower the degree of concurrency;
  - finer, more locking information that is needed to be stored.
- Best item size depends on the types of transactions.

Hierarchy of Granularity

- Could represent granularity of locks in a hierarchical structure.
- Root node represents entire database, level 1s represent files, etc.
- When node is locked, all its descendants are also locked.
- DBMS should check hierarchical path before granting lock.

Hierarchy of Granularity

- *Intention lock* could be used to lock all ancestors of a locked node.
- Intention locks can be read or write. Applied top-down, released bottom-up.

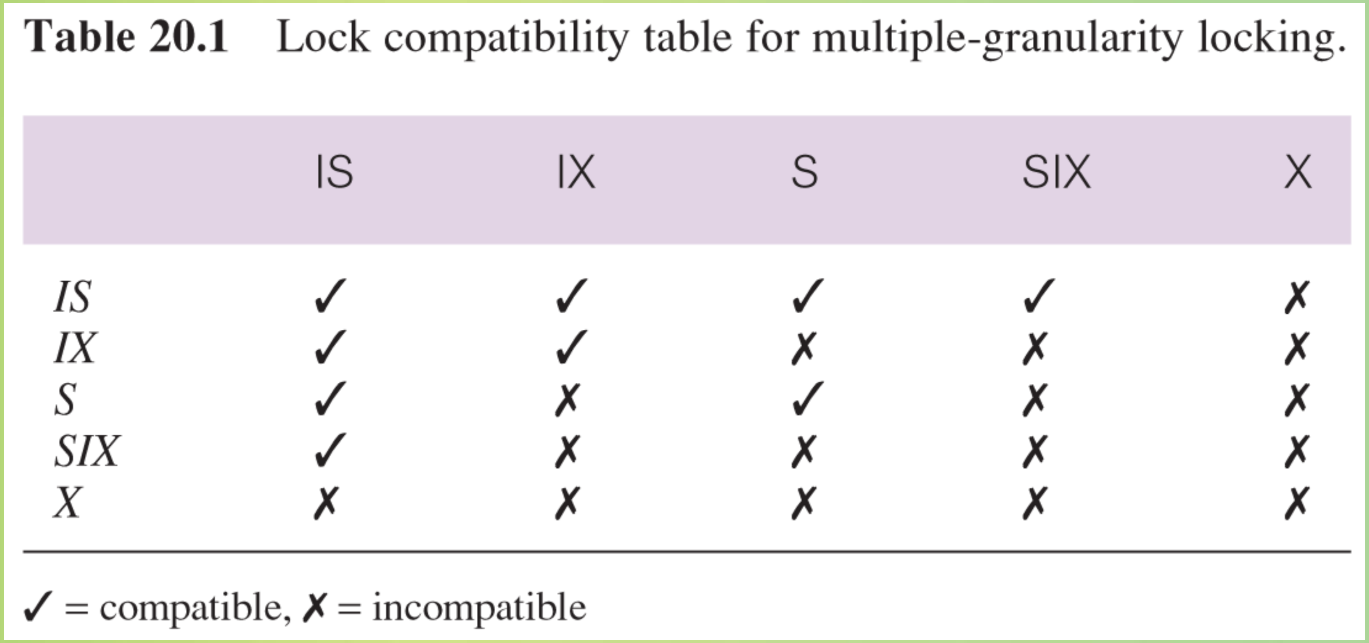
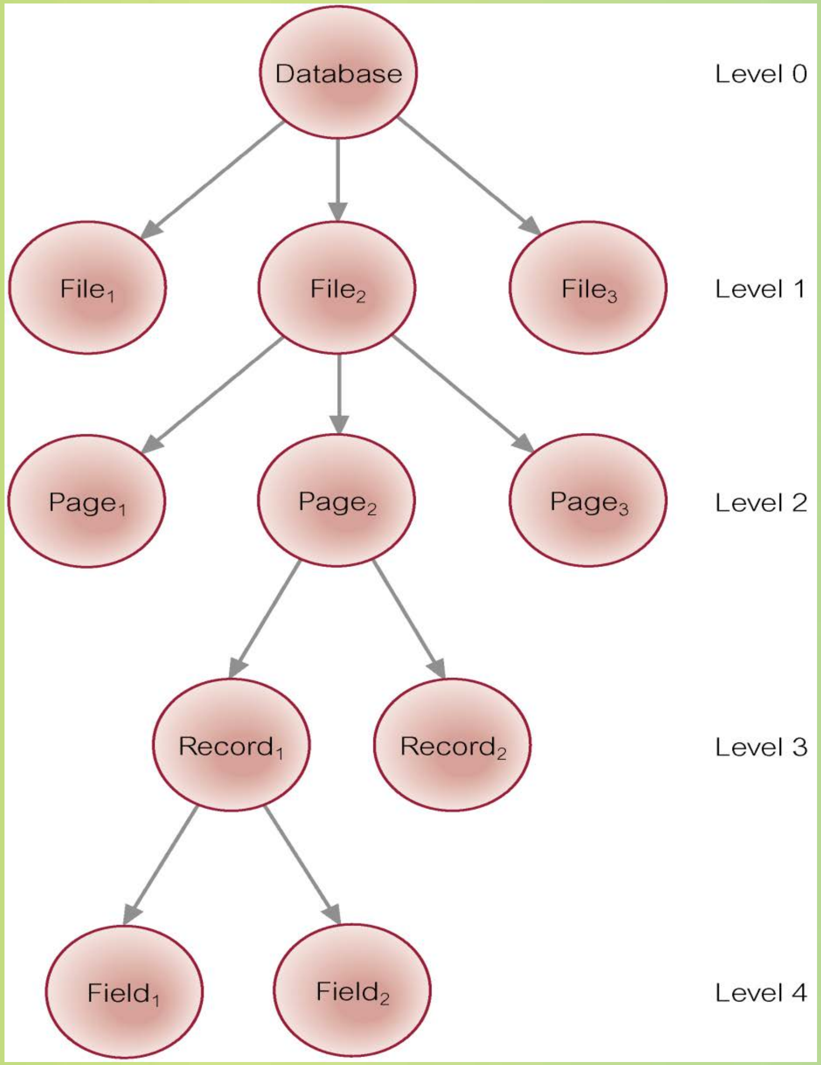


Table 20.1 Lock compatibility table for multiple-granularity locking.

	IS	IX	S	SIX	X
IS	✓	✓	✓	✓	✗
IX	✓	✓	✗	✗	✗
S	✓	✗	✓	✗	✗
SIX	✓	✗	✗	✗	✗
X	✗	✗	✗	✗	✗

✓ = compatible, ✗ = incompatible

Levels of Locking



Database Recovery

Process of restoring database to a correct state in the event of a failure.

- Need for Recovery Control
  - Two types of storage: volatile (main memory) and nonvolatile.
  - Volatile storage does not survive system crashes.
  - Stable storage represents information that has been replicated in several nonvolatile storage media with independent failure modes.

Types of Failures

- System crashes, resulting in loss of main memory.
- Media failures, resulting in loss of parts of secondary storage.
- Application software errors.
- Natural physical disasters.
- Carelessness or unintentional destruction of data or facilities.
- Sabotage.

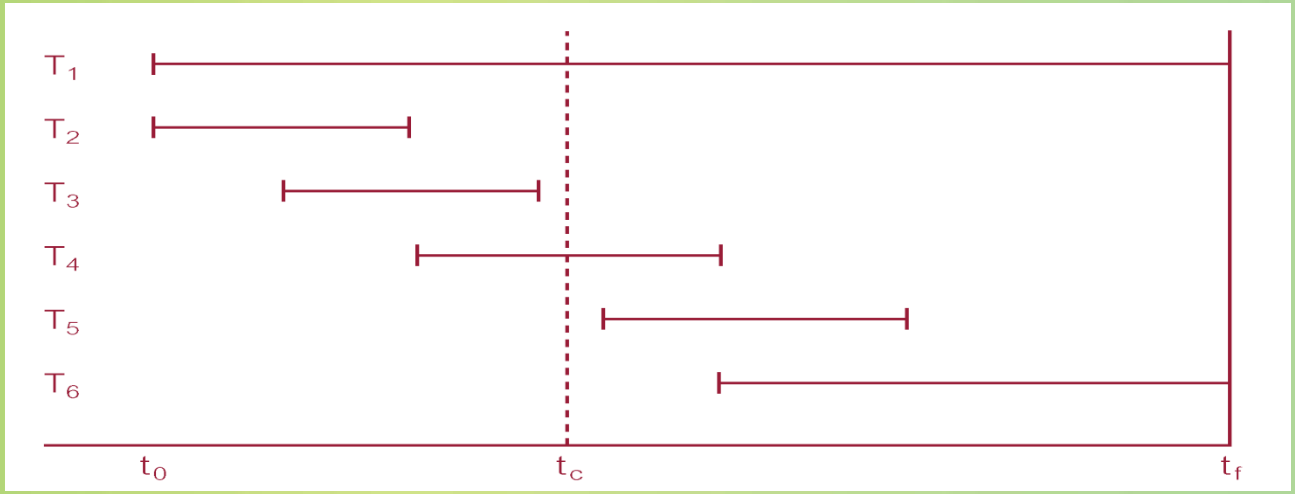
Transactions and Recovery

- Transactions represent basic unit of recovery.
- Recovery manager responsible for atomicity and durability.
- If failure occurs between commit and database buffers being flushed to secondary storage then, to ensure durability, recovery manager has to *redo* (*rollforward*) transaction's updates.

Transactions and Recovery

- **If transaction had not committed at failure time, recovery manager has to *undo* (*rollback*) any effects of that transaction for atomicity.**
- **Partial undo - only one transaction has to be undone.**
- **Global undo - all transactions have to be undone.**

Example



- **DBMS starts at time t_0, but fails at time t_f. Assume data for transactions T_2 and T_3 have been written to secondary storage.**
- **T_1 and T_6 have to be undone. In absence of any other information, recovery manager has to redo T_2, T_3, T_4, and T_5.**

Recovery Facilities

- **DBMS should provide following facilities to assist with recovery:**
  - **Backup mechanism, which makes periodic backup copies of database.**
  - **Logging facilities, which keep track of current state of transactions and database changes.**
  - **Checkpoint facility, which enables updates to database in progress to be made permanent.**
  - **Recovery manager, which allows DBMS to restore database to consistent state following a failure.**

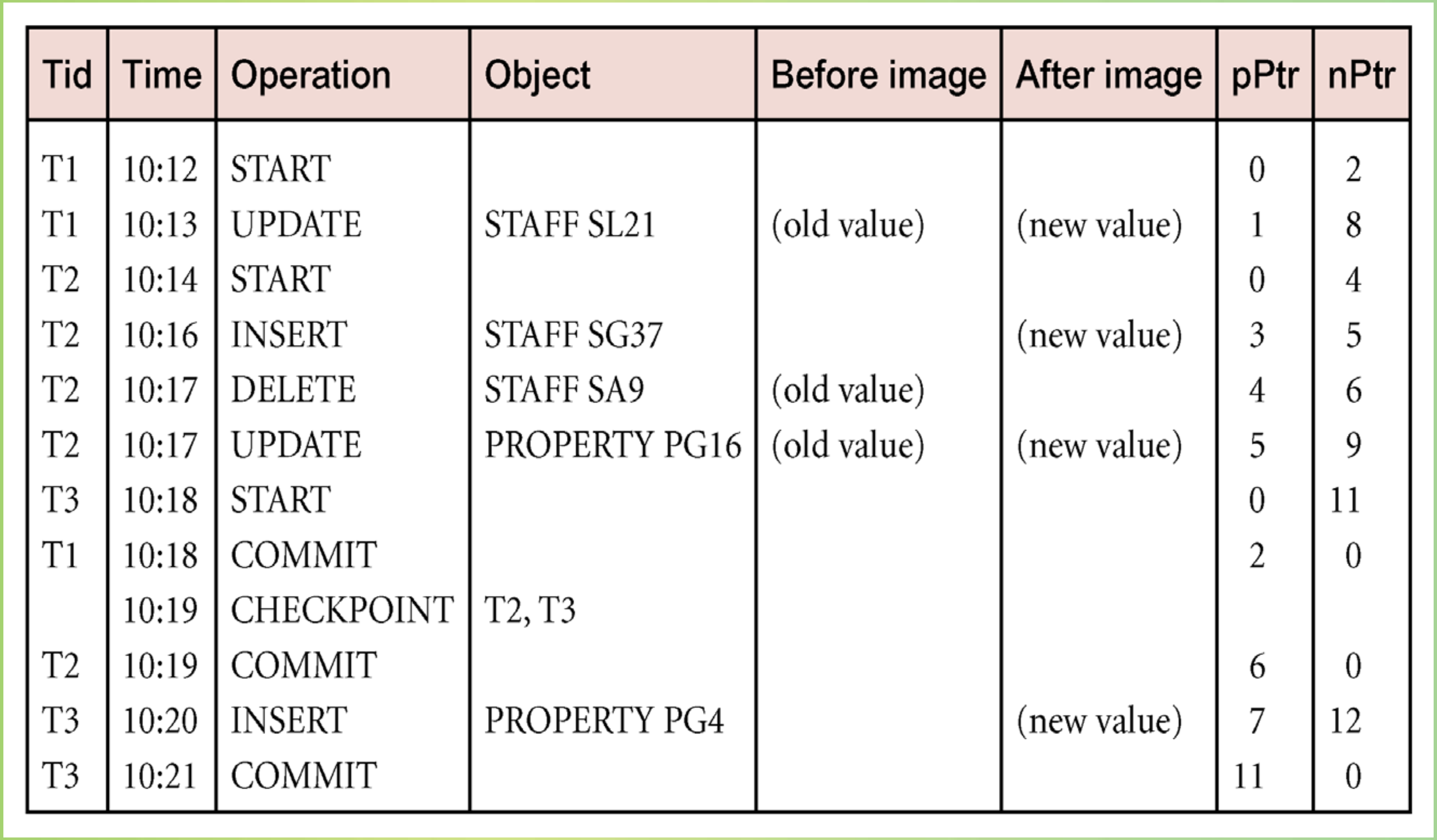
Log File

- Contains information about all updates to database:
  - Transaction records.
  - Checkpoint records.
- Often used for other purposes (for example, auditing).

Log File

- Transaction records contain:
  - Transaction identifier.
  - Type of log record, (transaction start, insert, update, delete, abort, commit).
  - Identifier of data item affected by database action (insert, delete, and update operations).
  - Before-image of data item.
  - After-image of data item.
  - Log management information.

Sample Log File



Tid	Time	Operation	Object	Before image	After image	pPtr	nPtr
T1	10:12	START				0	2
T1	10:13	UPDATE	STAFF SL21	(old value)	(new value)	1	8
T2	10:14	START				0	4
T2	10:16	INSERT	STAFF SG37		(new value)	3	5
T2	10:17	DELETE	STAFF SA9	(old value)		4	6
T2	10:17	UPDATE	PROPERTY PG16	(old value)	(new value)	5	9
T3	10:18	START				0	11
T1	10:18	COMMIT				2	0
	10:19	CHECKPOINT	T2, T3				
T2	10:19	COMMIT				6	0
T3	10:20	INSERT	PROPERTY PG4		(new value)	7	12
T3	10:21	COMMIT				11	0

Log File

- Log file may be duplexed or triplexed.
- Log file sometimes split into two separate random-access files.
- Potential bottleneck; critical in determining overall performance.

Checkpointing

<u>**Checkpoint**</u>

Point of synchronization between database and log file. All buffers are force-written to secondary storage.

- **Checkpoint record is created containing identifiers of all active transactions.**
- **When failure occurs, redo all transactions that committed since the checkpoint and undo all transactions active at time of crash.**

Checkpointing

- In previous example, with checkpoint at time t_c, changes made by T_2 and T_3 have been written to secondary storage.
- Thus:
  - only redo T_4 and T_5,
  - undo transactions T_1 and T6.

Recovery Techniques

- If database has been damaged:
  - Need to restore last backup copy of database and reapply updates of committed transactions using log file.
- If database is only inconsistent:
  - Need to undo changes that caused inconsistency. May also need to redo some transactions to ensure updates reach secondary storage.
  - Do not need backup, but can restore database using before- and after-images in the log file.

Main Recovery Techniques

- Three main recovery techniques:
  - Deferred Update
  - Immediate Update
  - Shadow Paging

Deferred Update

- **Updates are not written to the database until after a transaction has reached its commit point.**
- **If transaction fails before commit, it will not have modified database and so no undoing of changes required.**
- **May be necessary to redo updates of committed transactions as their effect may not have reached database.**

Immediate Update

- Updates are applied to database as they occur.
- Need to redo updates of committed transactions following a failure.
- May need to undo effects of transactions that had not committed at time of failure.
- Essential that log records are written before write to database. *Write-ahead log protocol*.

Immediate Update

- If no "*transaction commit*" record in log, then that transaction was active at failure and must be undone.
- Undo operations are performed *in reverse order in which they were written to log*.

Shadow Paging

- Maintain two page tables during life of a transaction: *current* page and *shadow* page table.
- When transaction starts, two pages are the same.
- Shadow page table is never changed thereafter and is used to restore database in event of failure.
- During transaction, current page table records all updates to database.
- When transaction completes, current page table becomes shadow page table.

Advanced Transaction Models

- Protocols considered so far are suitable for types of transactions that arise in traditional business applications, characterized by:
  - Data has many types, each with small number of instances.
  - Designs may be very large.
  - Design is not static but evolves through time.
  - Updates are far-reaching.
  - Cooperative engineering.

Advanced Transaction Models

- May result in transactions of long duration, giving rise to following problems:
  - More susceptible to failure - need to minimize amount of work lost.
  - May access large number of data items - concurrency limited if data inaccessible for long periods.
  - Deadlock more likely.
  - Cooperation through use of shared data items restricted by traditional concurrency protocols.

Advanced Transaction Models

- Look at five advanced transaction models:
  - Nested Transaction Model
  - Sagas
  - Multi-level Transaction Model
  - Dynamic Restructuring
  - Workflow Models

Nested Transaction Model

- Transaction viewed as hierarchy of subtransactions.
- Top-level transaction can have number of child transactions.
- Each child can also have nested transactions.
- In Moss's proposal, only leaf-level subtransactions allowed to perform database operations.
- Transactions have to commit from bottom upwards.
- However, transaction abort at one level does not have to affect transaction in progress at higher level.

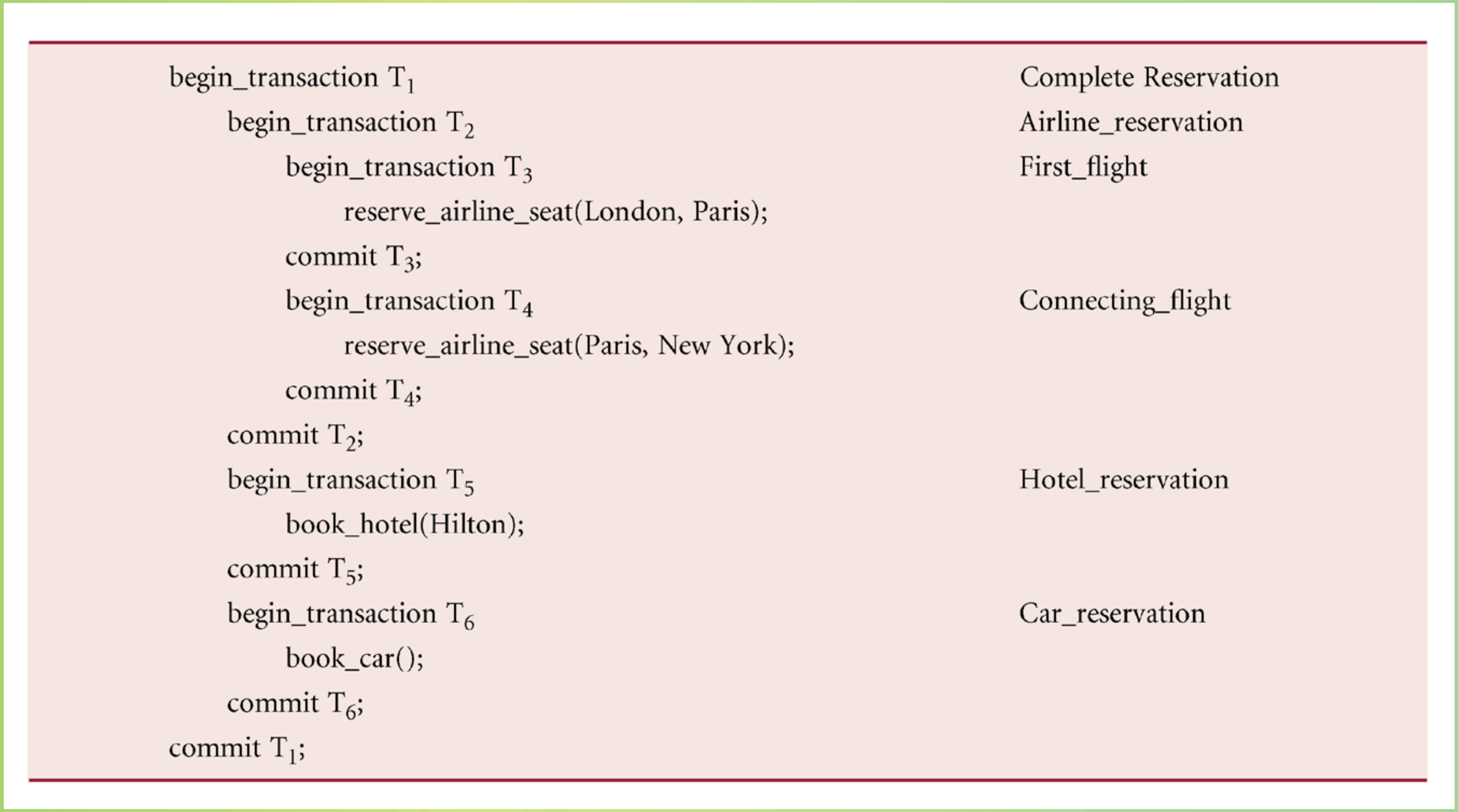
Nested Transaction Model

- Parent allowed to perform its own recovery:
  - Retry subtransaction.
  - Ignore failure, in which case subtransaction non-vital.
  - Run contingency subtransaction.
  - Abort.
- Updates of committed subtransactions at intermediate levels are visible only within scope of their immediate parents.

Nested Transaction Model

- **Further, commit of subtransaction is conditionally subject to commit or abort of its superiors.**
- **Using this model, top-level transactions conform to traditional ACID properties of flat transaction.**

Example of Nested Transactions



```
begin_transaction T$_1$                                   Complete Reservation
    begin_transaction T$_2$                               Airline_reservation
        begin_transaction T$_3$                           First_flight
            reserve_airline_seat(London, Paris);
        commit T$_3$;
        begin_transaction T$_4$                           Connecting_flight
            reserve_airline_seat(Paris, New York);
        commit T$_4$;
    commit T$_2$;
    begin_transaction T$_5$                               Hotel_reservation
        book_hotel(Hilton);
    commit T$_5$;
    begin_transaction T$_6$                               Car_reservation
        book_car();
    commit T$_6$;
commit T$_1$;
```

Nested Transaction Model - Advantages

- Modularity - transaction can be decomposed into number of subtransactions for purposes of concurrency and recovery.
- Finer level of granularity for concurrency control and recovery.
- Intra-transaction parallelism.
- Intra-transaction recovery control.

Emulating Nested Transactions using Savepoints

An identifiable point in flat transaction representing some partially consistent state.

- Can be used as restart point for transaction if subsequent problem detected.
- During execution of transaction, user can establish savepoint, which user can use to roll transaction back to.
- Unlike nested transactions, savepoints do not support any form of intra-transaction parallelism.

Sagas

“A sequence of (flat) transactions that can be interleaved with other transactions”.

- DBMS guarantees that either all transactions in saga are successfully completed or compensating transactions are run to undo partial execution.
- Saga has only one level of nesting.
- For every subtransaction defined, there is corresponding compensating transaction that will semantically undo subtransaction’s effect.

Sagas

- Relax property of isolation by allowing saga to reveal its partial results to other concurrently executing transactions before it completes.
- Useful when subtransactions are relatively independent and compensating transactions can be produced.
- May be difficult sometimes to define compensating transaction in advance, and DBMS may need to interact with user to determine compensation.

Multi-level Transaction Model

Closed nested transaction - atomicity enforced at the top-level.

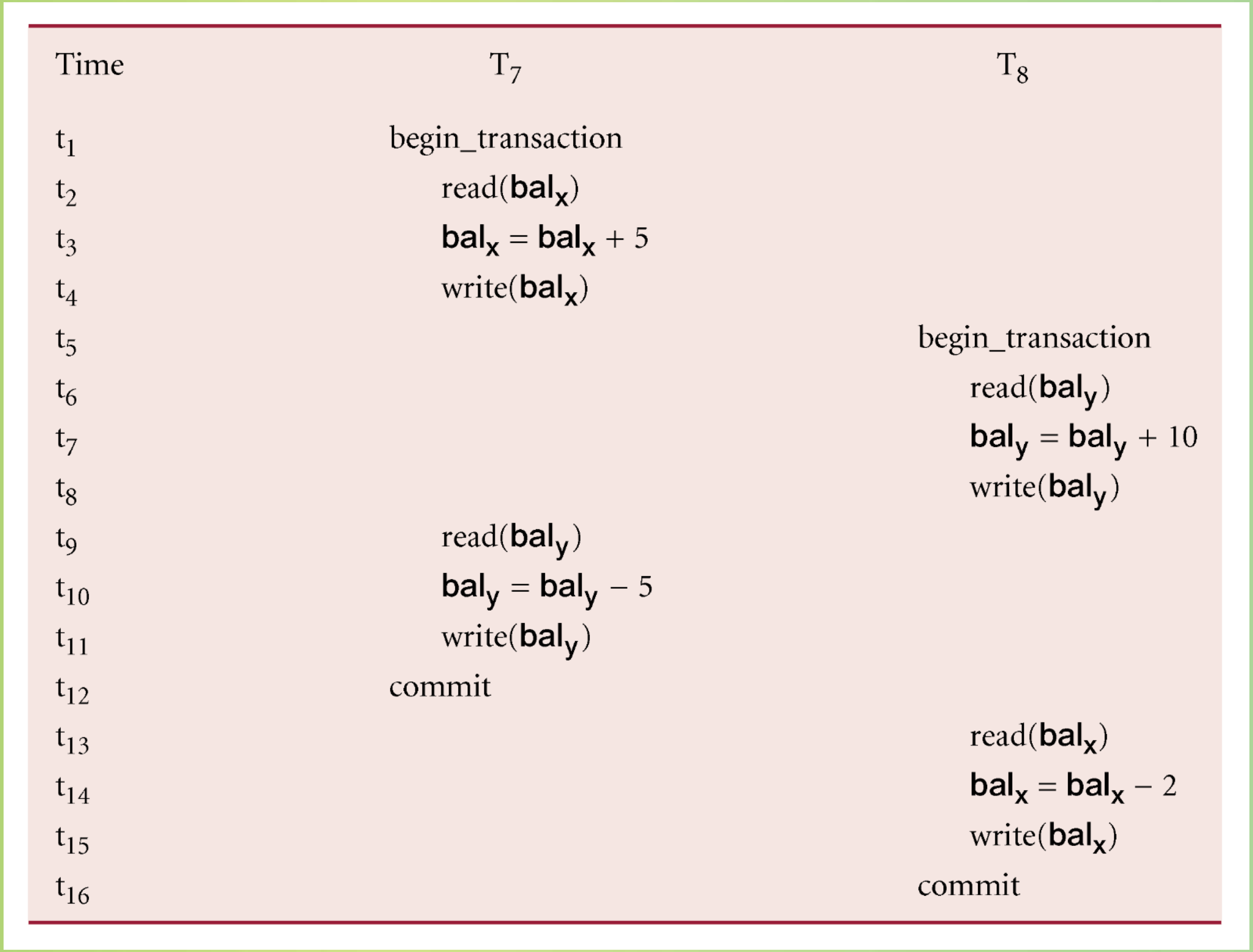
Open nested transactions - allow partial results of subtransactions to be seen outside transaction.

- **Saga model is example of open nested transaction.**
- **So is multi-level transaction model where tree of subtransactions is balanced.**
- **Nodes at same depth of tree correspond to operations of same level of abstraction in DBMS.**

Multi-level Transaction Model

- Edges represent implementation of an operation by sequence of operations at next lower level.
- Traditional flat transaction ensures no conflicts at lowest level (L0).
- In multi-level model two operations at level L*i* may not conflict even though their implementations at next lower level L*i-1* do.

Example - Multi-level Transaction Model



Time	T_7	T_8
t_1	begin_transaction	
t_2	read($\textbf{bal}_x$)	
t_3	$\textbf{bal}_x = \textbf{bal}_x + 5$	
t_4	write($\textbf{bal}_x$)	
t_5		begin_transaction
t_6		read($\textbf{bal}_y$)
t_7		$\textbf{bal}_y = \textbf{bal}_y + 10$
t_8		write($\textbf{bal}_y$)
t_9	read($\textbf{bal}_y$)	
t_{10}	$\textbf{bal}_y = \textbf{bal}_y - 5$	
t_{11}	write($\textbf{bal}_y$)	
t_{12}	commit	
t_{13}		read($\textbf{bal}_x$)
t_{14}		$\textbf{bal}_x = \textbf{bal}_x - 2$
t_{15}		write($\textbf{bal}_x$)
t_{16}		commit

Example - Multi-level Transaction Model

T_7:T_{71}, which increases bal_x by 5

T_{72}, which subtracts 5 from bal_y

T_8:T_{81}, which increases bal_y by 10

T_{82}, which subtracts 2 from bal_x

- As addition and subtraction commute, can execute these subtransactions in any order, and correct result will always be generated.

Dynamic Restructuring

- To address constraints imposed by ACID properties of flat transactions, two new operations proposed: split_transaction and join_transaction.
- <u>split-transaction</u> - splits transaction into two serializable transactions and divides its actions and resources (for example, locked data items) between new transactions.
- Resulting transactions proceed independently.

Dynamic Restructuring

- Allows partial results of transaction to be shared, while still preserving its semantics.
- Can be applied only when it is possible to generate two transactions that are serializable with each other and with all other concurrently executing transactions.

Dynamic Restructuring

- **Conditions that permit transaction to be split into A and B are:**
  - **AWriteSet $\cap$ BWriteSet $\subseteq$ BWriteLast.**

    If both A and B write to same object, B's write operations must follow A's write operations.
  - **AReadSet $\cap$ BWriteSet = $\varnothing$.**

    A cannot see any results from B.
  - **BReadSet $\cap$ AWriteSet = ShareSet.**

    B may see results of A.

Dynamic Restructuring

- **These guarantee that A is serialized before B.**
- **However, if A aborts, B must also abort.**
- **If both BWriteLast and ShareSet are empty, then A and B can be serialized in any order and both can be committed independently.**

Dynamic Restructuring

- join-transaction - performs reverse operation, merging ongoing work of two or more independent transactions, as though they had always been single transaction.

Dynamic Restructuring

- **Main advantages of dynamic restructuring are:**
  - **Adaptive recovery.**
  - **Reducing isolation.**

Workflow Models

- Has been argued that above models are still not powerful to model some business activities.
- More complex models have been proposed that are combinations of open and nested transactions.
- However, as they hardly conform to any of ACID properties, called workflow model used instead.
- Workflow is activity involving coordinated execution of multiple tasks performed by different processing entities (people or software systems).

Workflow Models

- **Two general problems involved in workflow systems:**
  - **specification of the workflow,**
  - **execution of the workflow.**
- **Both problems complicated by fact that many organizations use multiple, independently managed systems to automate different parts of the process.**