Transaction Management

Watch video: https://youtu.be/dH2avf3x4Xg?t=322

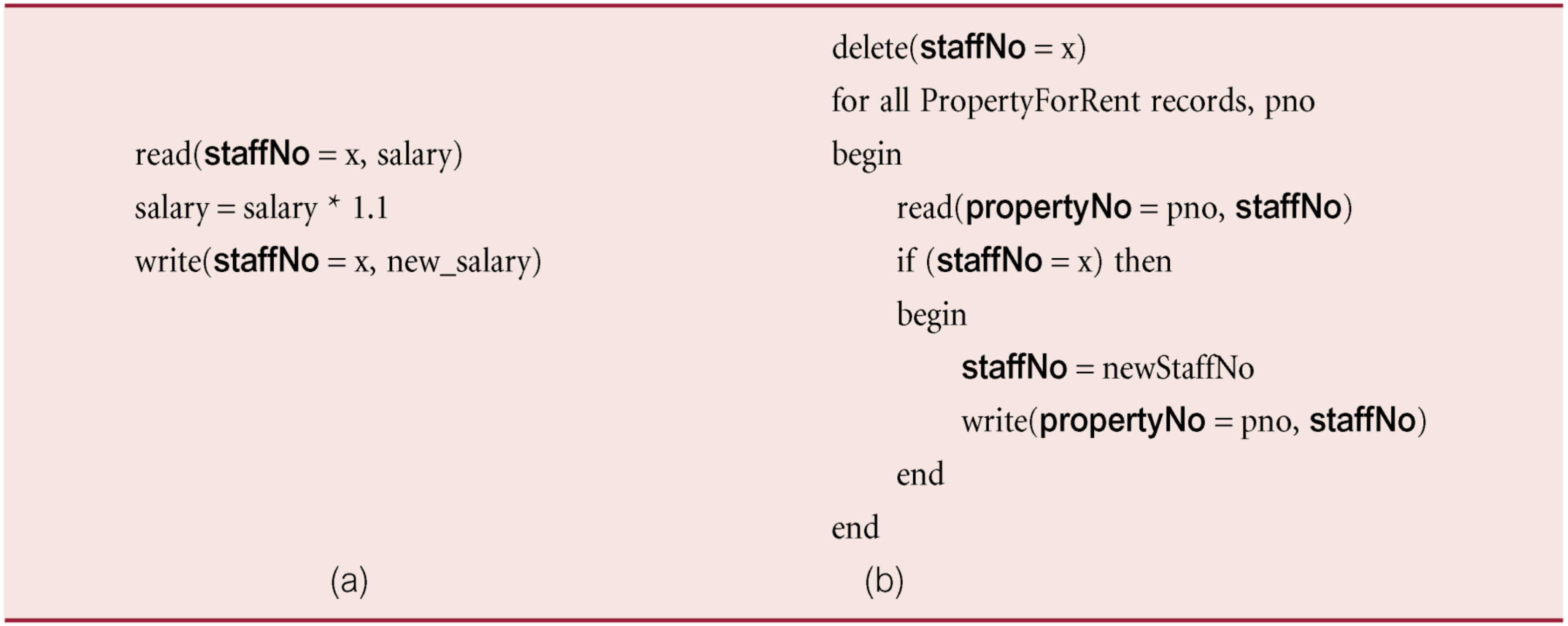
Agenda

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

Transaction (definition)

- 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.
- Transforms database from one consistent state to another, although consistency may be violated during transaction.

Example Transactions



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

(a)

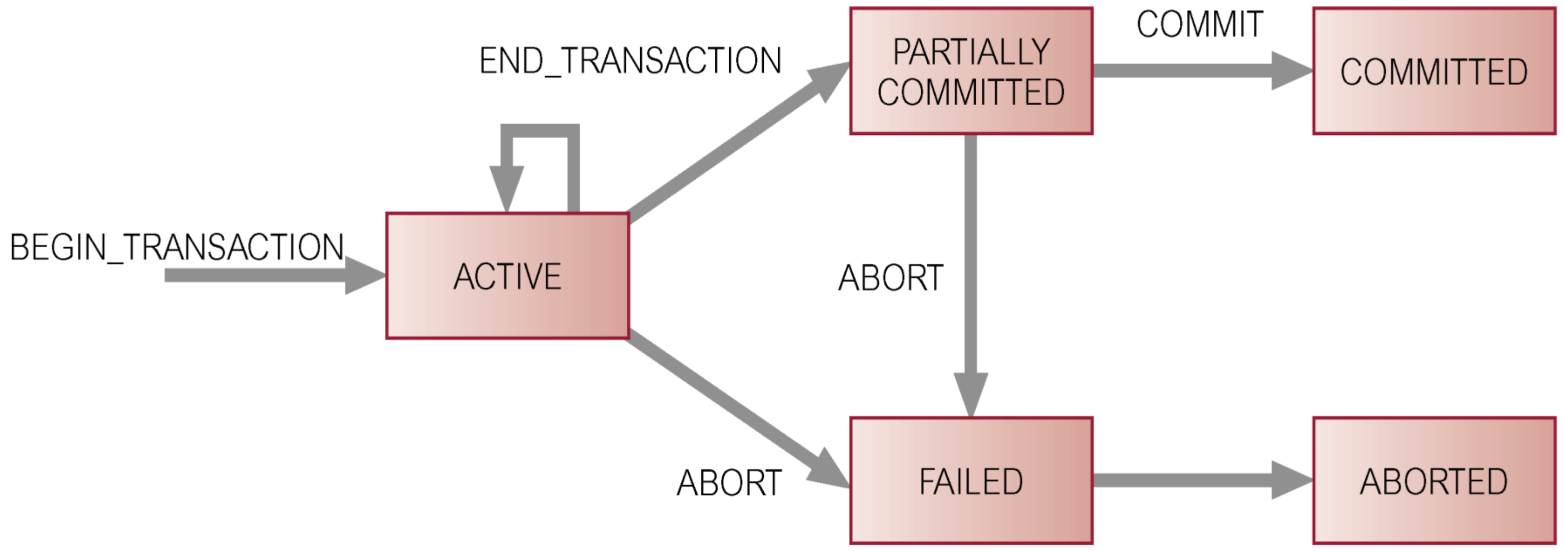
```
delete(staffNo = x)
for all PropertyForRent records, pno
begin
    read(propertyNo = pno, staffNo)
    if (staffNo = x) then
    begin
        staffNo = newStaffNo
        write(propertyNo = pno, staffNo)
    end
end
```

(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



Agenda

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

ACID Properties of Transactions

- Four basic (**ACID**) properties of 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

ACID examples: atomicity

- Atomicity means all or nothing. Transactions often contain multiple separate actions. For example, a transaction may insert data into one table, delete from another table, and update a third table. Atomicity ensures that either all of these actions occur or none at all.

- An example of an atomic transaction is an account transfer transaction. The money is removed from account A then placed into account B. If the system fails after removing the money from account A, then the transaction processing system will put the money back into account A, thus returning the system to its original state. This is known as a rollback.

ACID examples: consistency

- Consistency means that transactions always take the database from one consistent state to another. So, if a transaction violates the databases consistency rules, then the entire transaction will be rolled back.

- Looking again at the account transfer system, the system is consistent if the total of all accounts is constant. If an error occurs and the money is removed from account A and not added to account B, then the total in all accounts would have changed. The system would no longer be consistent. By rolling back the removal from account A, the total will again be what it should be, and the system back in a consistent state.

ACID examples: isolation

- Isolation means that concurrent transactions, and the changes made within them, are not visible to each other until they complete. This avoids many problems, including those that could lead to violation of other properties. The implementation of isolation is quite different in different DBMS. This is also the property most often related to locking problems.

- For example, a teller looking up a balance must be isolated from a concurrent transaction involving a withdrawal from the same account. Only when the withdrawal transaction commits successfully and the teller looks at the balance again will the new balance be reported.

ACID examples: durability

- Durability means that committed transactions will not be lost, even in the event of abnormal termination. That is, once a user or program has been notified that a transaction was committed, they can be certain that the data will not be lost.

- A system crash or any other failure must not be allowed to lose the results of a transaction or the contents of the database. Durability is often achieved through separate transaction logs that can "re-create" all transactions from a certain point in time.

Agenda

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

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

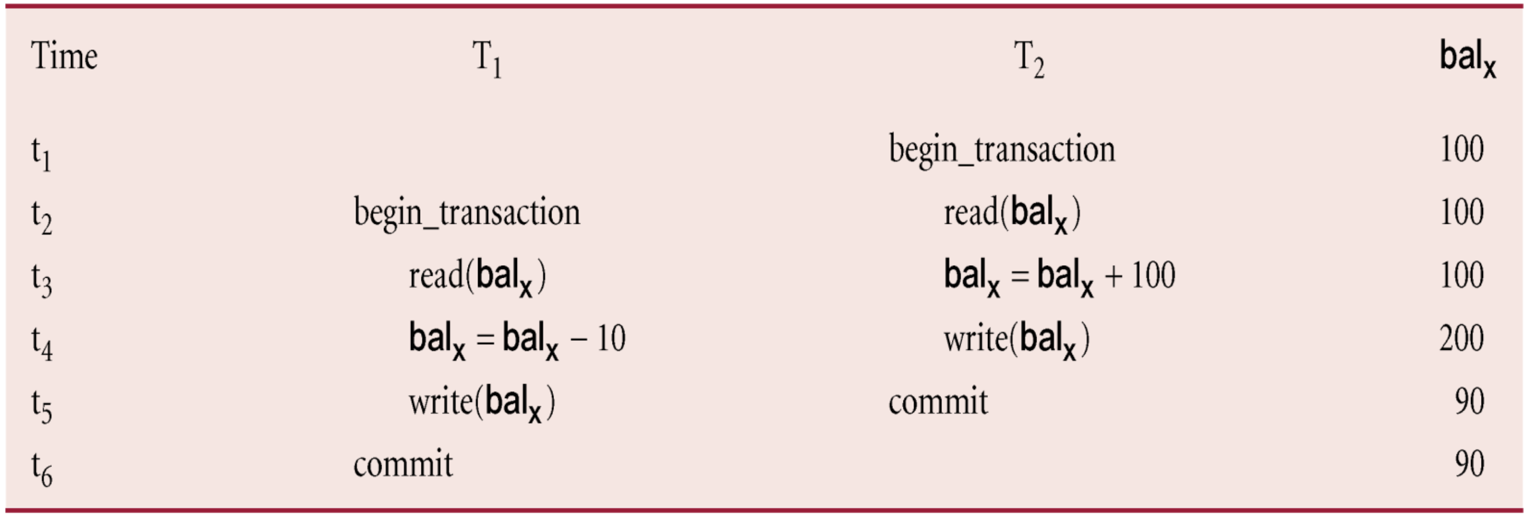
- Three examples of potential problems caused by concurrency:
  - Lost update problem.
  - Uncommitted dependency problem.
  - Inconsistent analysis problem.

Lost Update Problem

- Successfully completed update is overridden by another user.
- T1 withdrawing €10 from an account with bal_x, initially €100.
- T2 depositing €100 into same account.
- Serially, final balance would be €190.

Lost Update Problem

- Loss of T2’s update could be avoided by preventing T1 from reading bal_x until after update.



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

Uncommitted Dependency Problem

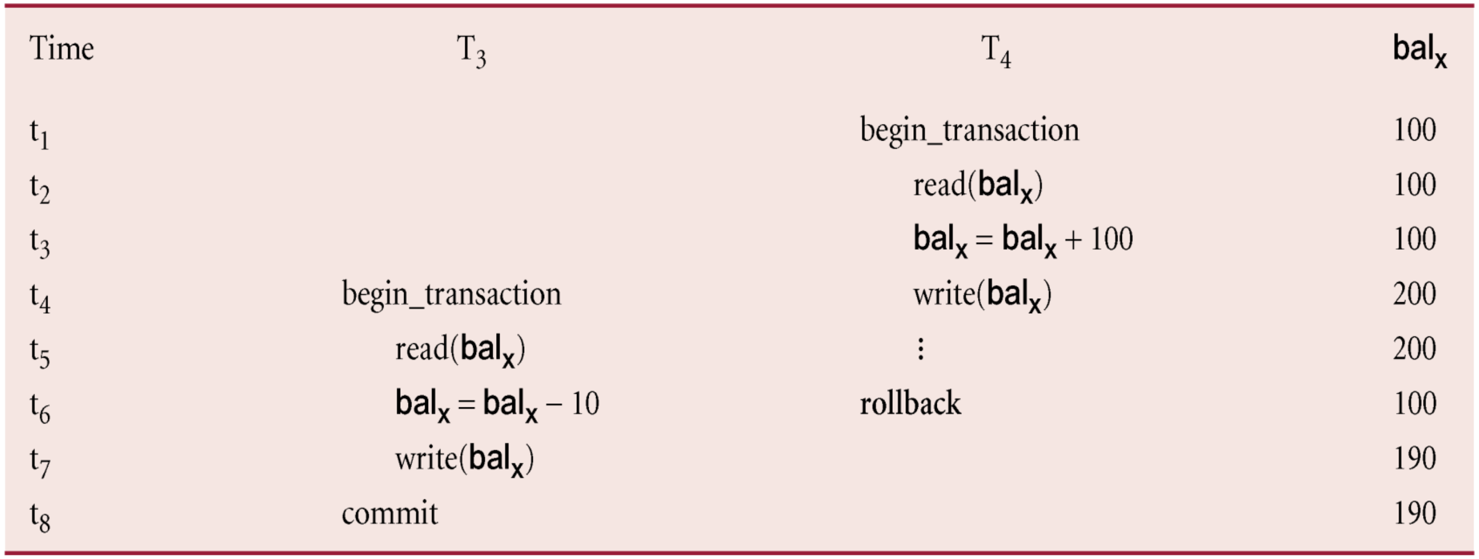
- Occurs when one transaction can see intermediate results of another transaction before it has committed.

- T4 updates bal_x to €200 but it aborts, so bal_x should be back at original value of €100.

- T3 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

- Problem could be avoided by preventing T3 from reading bal_x until after T4 commits or aborts.



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

Inconsistent Analysis Problem

- Occurs when transaction reads several values but second transaction updates some of them during execution of first.

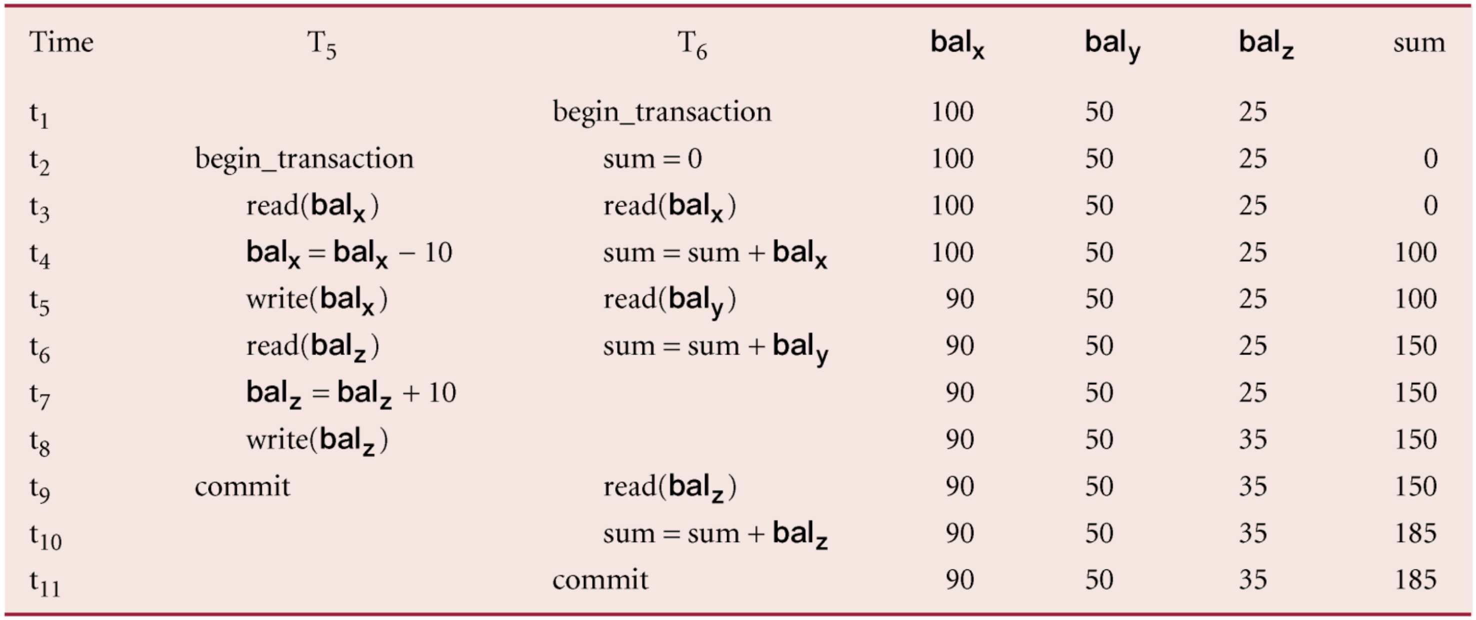
- Sometimes referred to as dirty read or unrepeatable read.

- T6 is totaling balances of account x (€100), account y (€50), and account z (€25).

- Meantime, T5 has transferred €10 from bal_x to bal_z, so T6 now has wrong result (€10 too high).

Inconsistent Analysis Problem

- Problem could be avoided by preventing T6 from reading bal_x and bal_z until after T5 completed updates.



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

Agenda

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

Serialisability

- 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.

- The goal of serialisability is to allow transactions to run concurrently while still having the same results as if they were run sequentially (i.e. separately, one after the other)

Concurrency Control Techniques

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

Agenda

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

Locking

- Transaction uses locks to deny access to other transactions and so prevent incorrect updates.
- Most widely used approach to ensure serialisability.
- 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, it can read but not update item.
- If transaction has **exclusive** lock on item, it 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 (i.e. the item cannot even be read by other transactions).

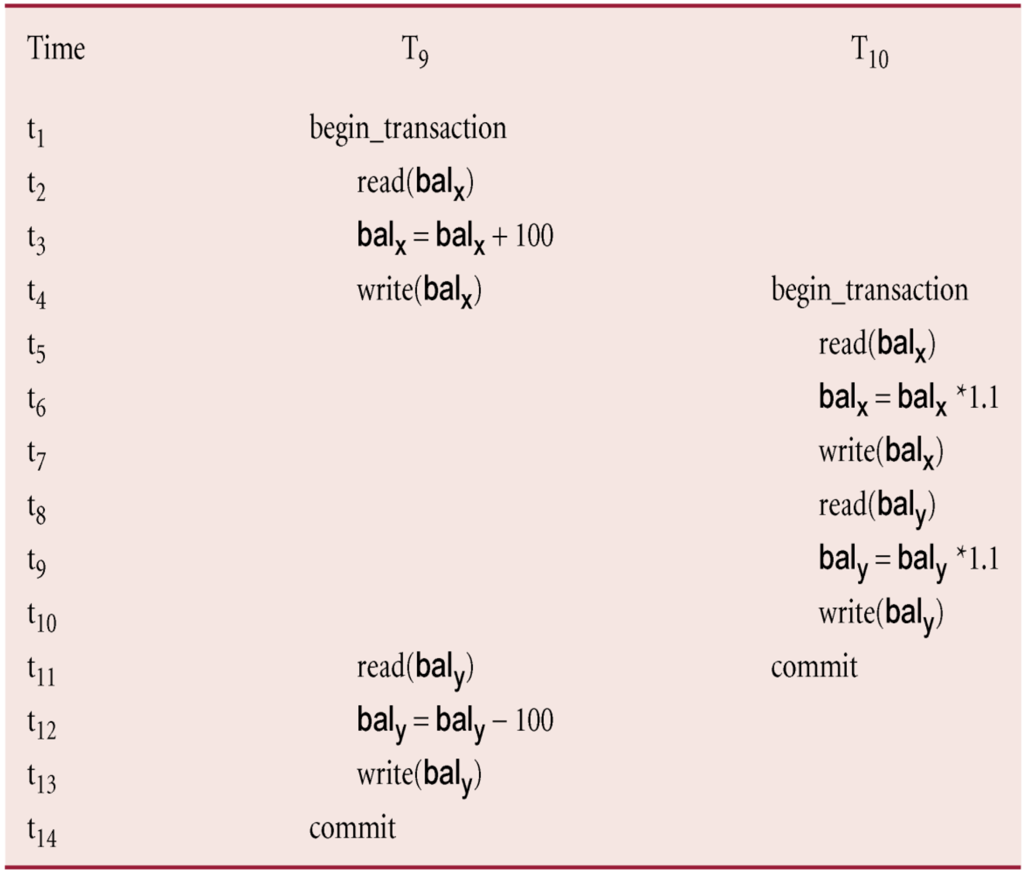
Locking - Basic Rules

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

Locking

- Locking does not allow guarantee serialisability
- In the example on the next slide, if the lock on bal_x is released by T9 as soon as the update is completed, T10 can proceed to read the value of bal_x
- However, this could cause inconsistencies if T9 was later rolled back instead of committed

Example

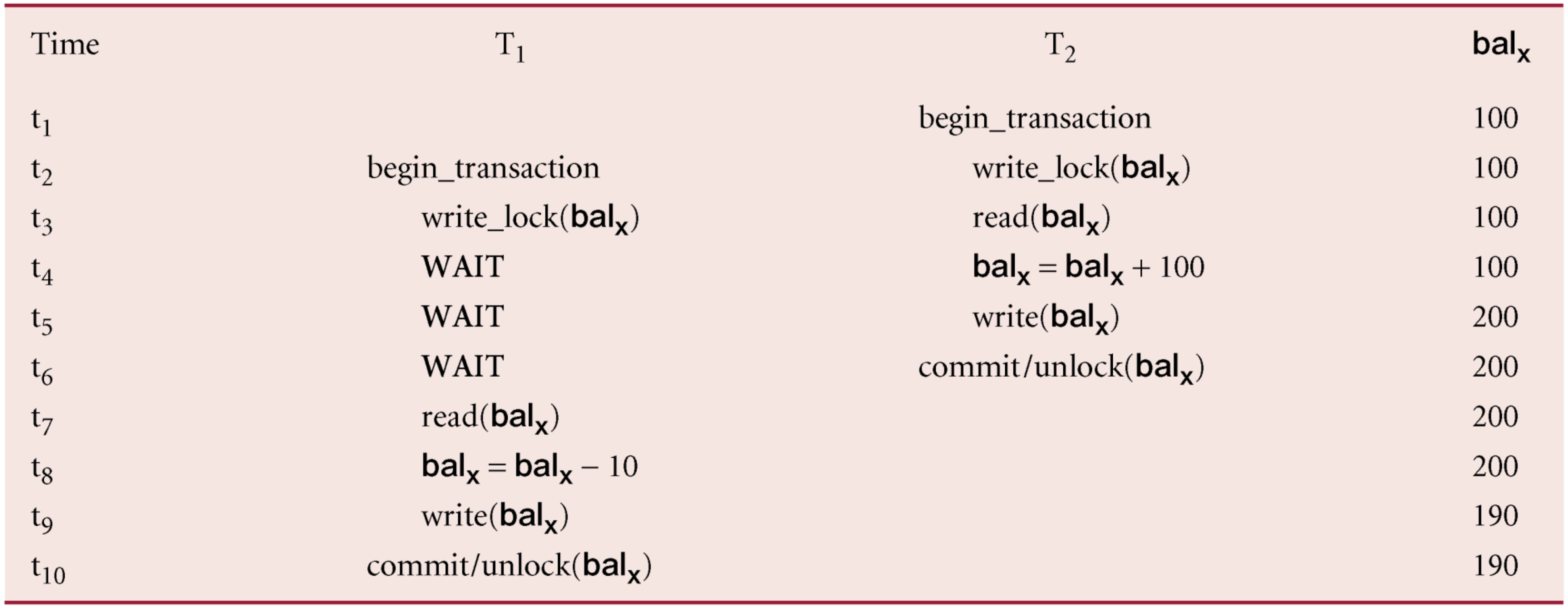


Time	T_9	T_{10}
t_1	begin_transaction	
t_2	read(**bal_x**)	
t_3	**bal_x** = **bal_x** + 100	
t_4	write(**bal_x**)	begin_transaction
t_5		read(**bal_x**)
t_6		**bal_x** = **bal_x** *1.1
t_7		write(**bal_x**)
t_8		read(**bal_y**)
t_9		**bal_y** = **bal_y** *1.1
t_{10}		write(**bal_y**)
t_{11}	read(**bal_y**)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

Two-Phase Locking (2PL)

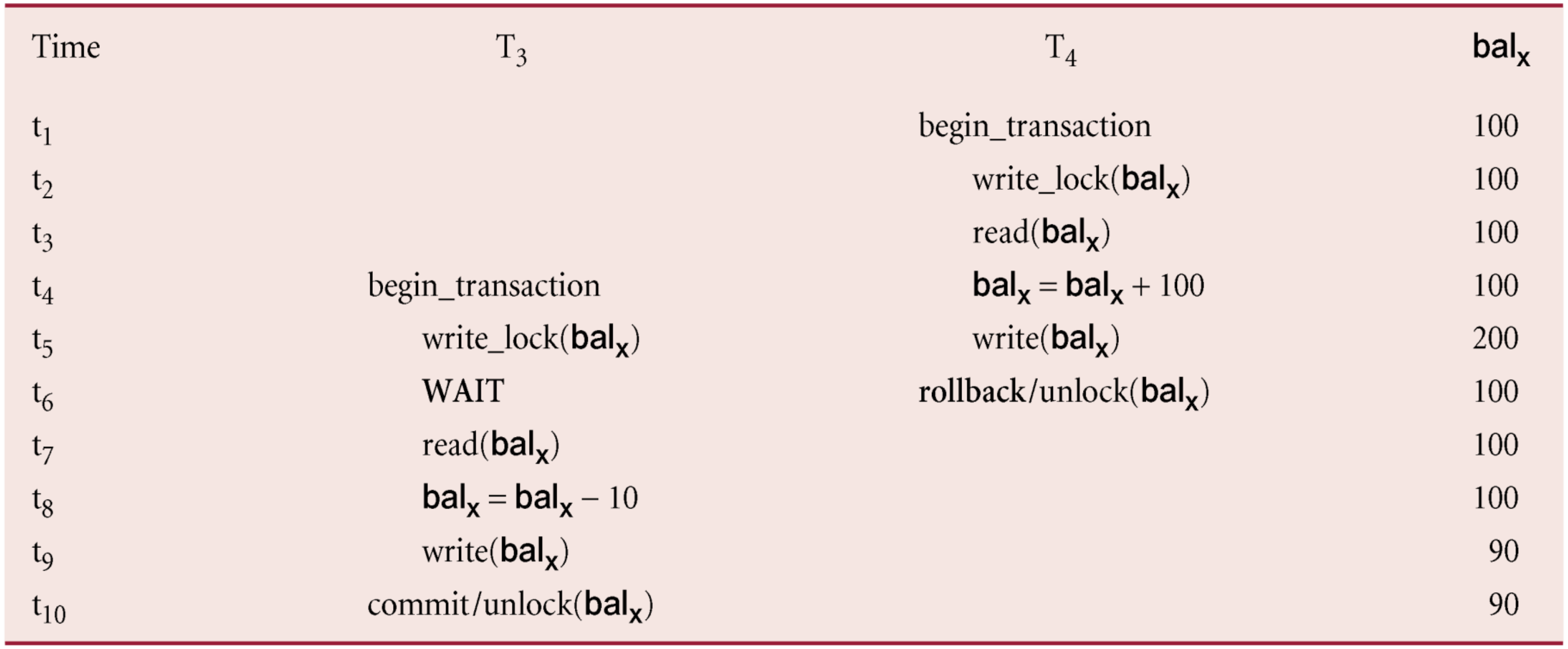
- Transaction follows 2PL protocol if all locking operations precede first unlock operation in the transaction.
    - In other words, all locks are held until the transaction ends (either committed or rolled back)
- Two phases for transaction:
    - Growing phase - acquires all locks but cannot release any locks.
    - Shrinking phase - releases locks but cannot acquire any new locks.

Preventing Lost Update Problem using 2PL



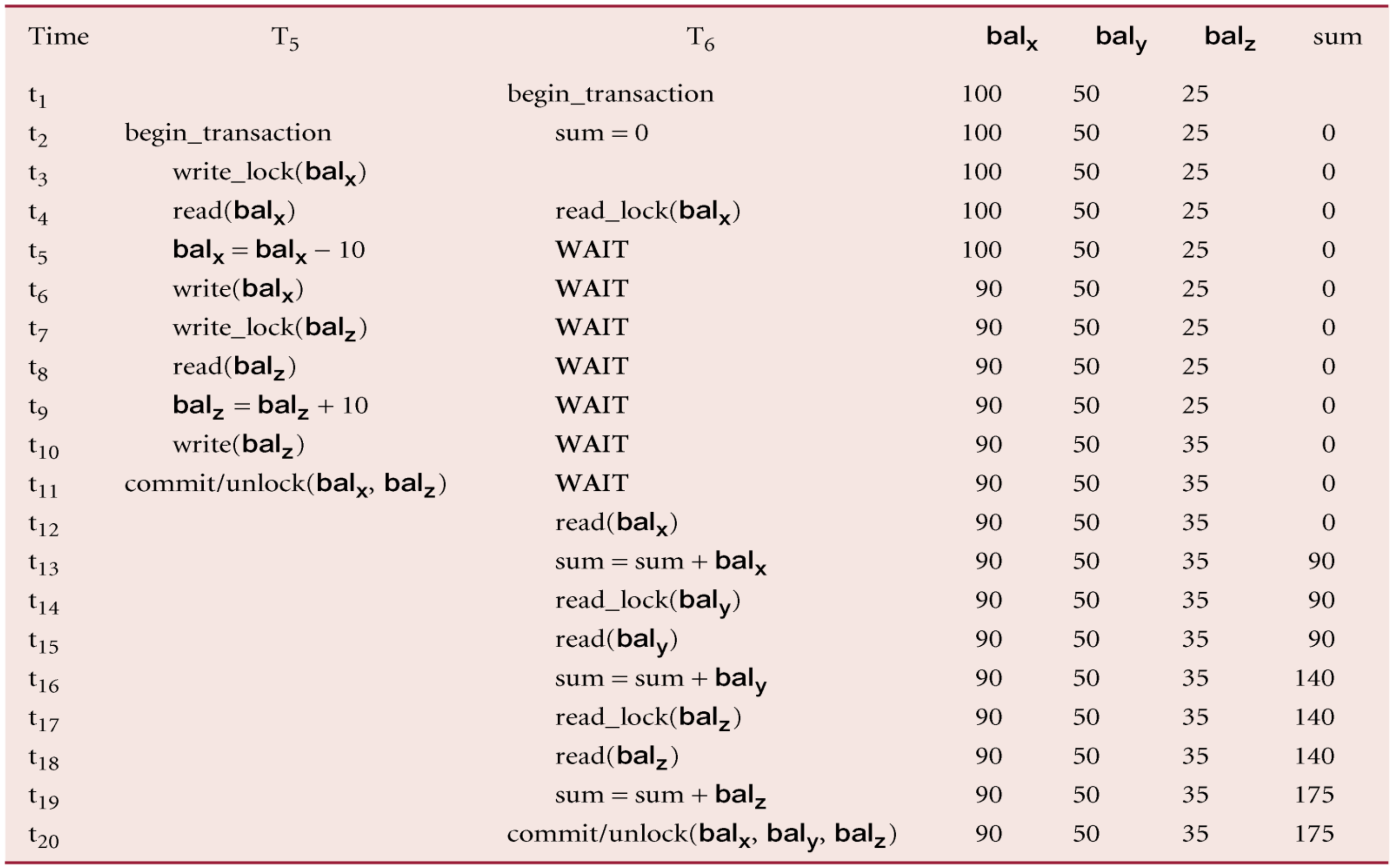
Time	T_1	T_2	bal_x
t_1		begin_transaction	100
t_2	begin_transaction	write_lock(bal_x)	100
t_3	write_lock(bal_x)	read(bal_x)	100
t_4	WAIT	$bal_x = bal_x + 100$	100
t_5	WAIT	write(bal_x)	200
t_6	WAIT	commit/unlock(bal_x)	200
t_7	read(bal_x)		200
t_8	$bal_x = bal_x - 10$		200
t_9	write(bal_x)		190
t_{10}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

Preventing Uncommitted Dependency Problem using 2PL



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

Preventing Inconsistent Analysis Problem using 2PL



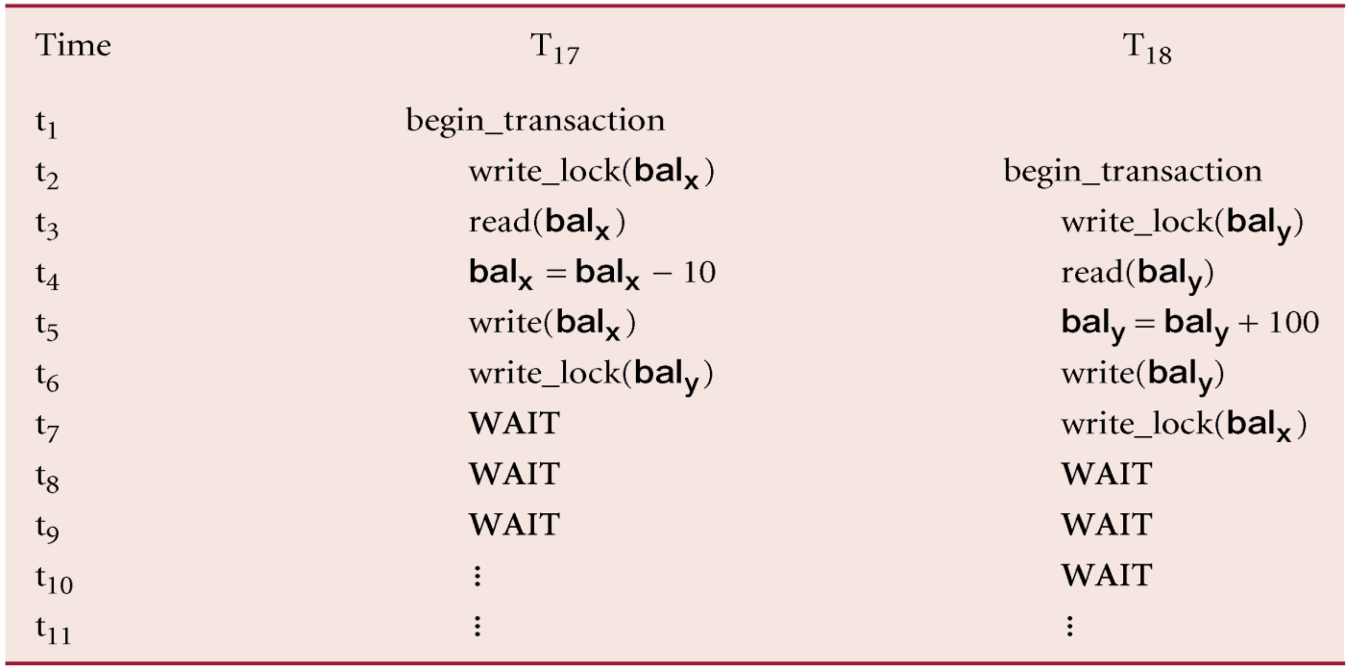
Time	T_5	T_6	$\mathbf{bal_x}$	$\mathbf{bal_y}$	$\mathbf{bal_z}$	sum
t_1		begin_transaction	100	50	25	
t_2	begin_transaction	sum = 0	100	50	25	0
t_3	write_lock($\mathbf{bal_x}$)		100	50	25	0
t_4	read($\mathbf{bal_x}$)	read_lock($\mathbf{bal_x}$)	100	50	25	0
t_5	$\mathbf{bal_x} = \mathbf{bal_x} - 10$	WAIT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

Agenda

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

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($\mathbf{bal_x}$)	begin_transaction
t_3	read($\mathbf{bal_x}$)	write_lock($\mathbf{bal_y}$)
t_4	$\mathbf{bal_x} = \mathbf{bal_x} - 10$	read($\mathbf{bal_y}$)
t_5	write($\mathbf{bal_x}$)	$\mathbf{bal_y} = \mathbf{bal_y} + 100$
t_6	write_lock($\mathbf{bal_y}$)	write($\mathbf{bal_y}$)
t_7	WAIT	write_lock($\mathbf{bal_x}$)
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).
- 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).

Wait-die vs. Wound-wait

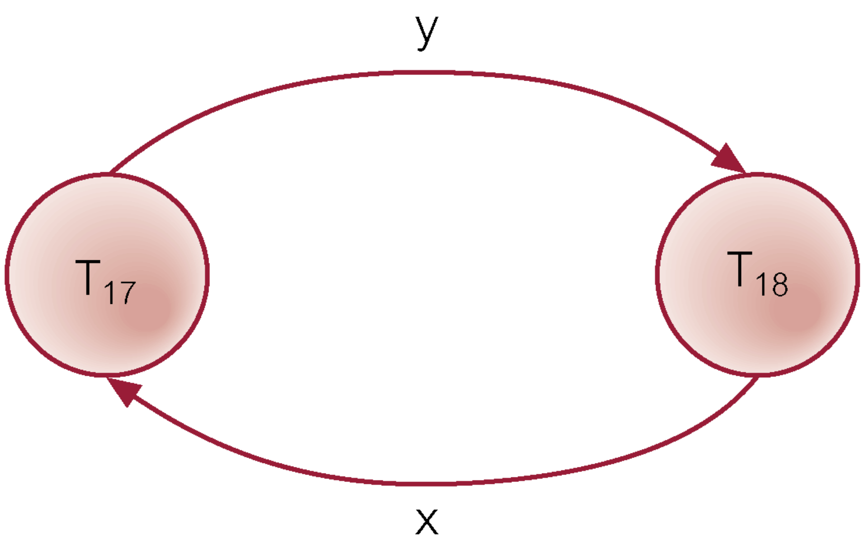
	Wait-Die	Wound-Wait
O needs resource held by Y	O waits	Y dies
Y needs resource held by O	Y dies	Y waits

O = Older transaction
Y = Younger transaction

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 Ti -> Tj, if Ti waiting to lock item locked by Tj.
- 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.

Agenda

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

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

- Timestamp
  - 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 read/write 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 example

John	Time	Marsha	Bal_x	Data Item Timestamp
read (bal_x)	t1		100	Last read by John (Older) at t1
	t2	read (bal_x)	100	Last read by Marsha (Newer) at t2
$bal_x = bal_x - 50$	t3		50	
write (bal_x) *	t4		50	
roll back	t5	$bal_x = bal_x – 10$	90	
	t6	write (bal_x)	90	Last updated by Marsha (Older) at t6
read (bal_x)	t7		90	Last read by John (Newer) at t7
	t8		90	
$bal_x = bal_x - 50$	t9		40	
write (bal_x) **	t10		40	Last update by John (Newer) at t10
	t11			
	t12			

Timestamping example

***** Problem occurs here – John (the ‘older’ transaction) has tried to update a data item which was last read by a ‘newer’ transaction (Marsha). Therefore his transaction must be rolled back - aborted and restarted, and given a new timestamp. **From time t6 and onwards Marsha is now the older transaction, whereas John’s transaction is the newer of the two (having been given a new timestamp).**

****** This time John’s update to the balance is allowed to proceed, as the data item has not been read/updated by anyone else since his transaction was restarted.

Agenda

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

Versioning

- 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.
- In multiversion concurrency control, each write operation creates new version of data item while retaining old version.
- New versions are later merged into the database; conflicts are dealt with if they arise

Optimistic Techniques

- Based on assumption that conflict is rare and more efficient to let transactions proceed without delays to ensure serialisability.
- 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.

Agenda

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

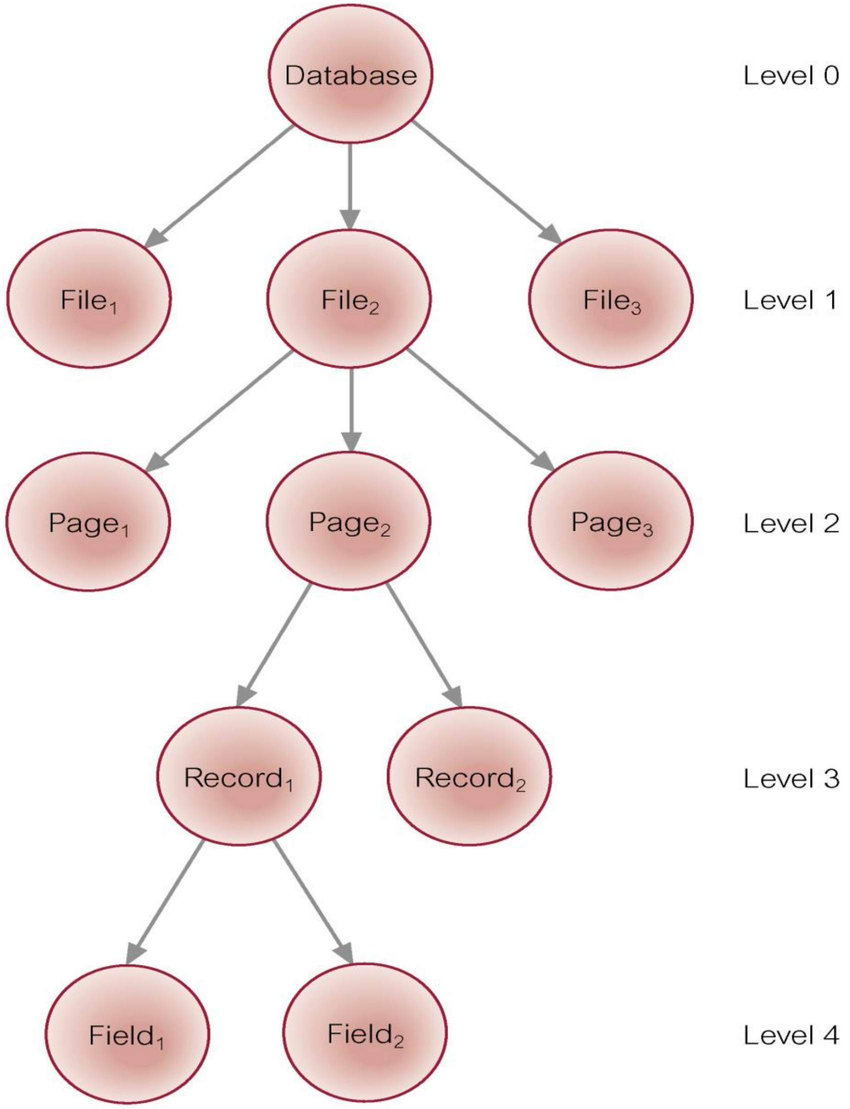
Granularity of Data Items

- 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.

Levels of Locking


Database
Level 0
File$_1$
File$_2$
File$_3$
Level 1
Page$_1$
Page$_2$
Page$_3$
Level 2
Record$_1$
Record$_2$
Level 3
Field$_1$
Field$_2$
Level 4