Transaction Management

A Banking Example

- Cathy wants to transfer $1000 from her saving account to her checking account
  - Debit $1000 from the saving account
  - Credit $1000 from the checking account
  - Cathy's total balance should remain the same
- Implementation in a database?

```
UPDATE account SET balance = balance – 1000
WHERE user = "Cathy" AND type = "saving"
UPDATE account SET balance = balance + 1000
WHERE user = "Cathy" AND type = "checking"
```

What If An Error Happens?

UPDATE account SET balance = balance – 1000
WHERE user = “Cathy” AND type = “saving”

What if a system failure happens here?

UPDATE account SET balance = balance + 1000
WHERE user = “Cathy” AND type = “checking”

- Cathy’s total balance is $1000 less
  - Cathy should not lose $1000 due to the system failure!

Transaction

- The two updates should be bounded into a unit – either both operations succeed, or none of them take effect
  - Cathy's total balance should be maintained consistently
- Transaction: a unit of program execution that accesses and possibly updates various data

```
UPDATE account SET balance = balance – 1000
WHERE user = "Cathy" AND type = "saving"
UPDATE account SET balance = balance + 1000
WHERE user = "Cathy" AND type = "checking"
```

Consistency of Transactions

- Consistency requirements: constraints (business rules) that should be respected by transactions
  - Example: if an amount is transferred from one account to another, the total balance of the two accounts should not change
- Execution of a transaction independently preserves the consistency of the database

Atomicity of Transactions

- In a transaction, either all operations of the transaction are reflected properly in the database, or none are
  - In a fund transfer transaction, either the fund is transferred (i.e., both accounts are adjusted properly) or not at all (i.e., both accounts are not updated)
- A database may have an inconsistent state at some point, but an inconsistent state should not be visible to users

Ensuring Atomicity

- What if a system failure happens during fund transfer?
  - Recover the balances of accounts before the transaction
- A database system keeps track of the old consistent values and restores those values if a transaction fails

Durability of Transactions

- If a fund transfer transaction succeeds, even if a system failure later should not lead to the loss of the transaction effect
- Once a transaction completes successfully, all the updates that it carried out on the database persist, even if there is a system failure after the transaction completes execution

Ensuring Durability

- Using a reliable data storage (e.g., hard disk or tape)
- The updates carried out by the transaction have been written to disk before the transaction completes
- Information about the updates carried out by the transaction and written to disk is sufficient to enable the database to reconstruct the updates when the database system is restarted after failure

Concurrent Transactions

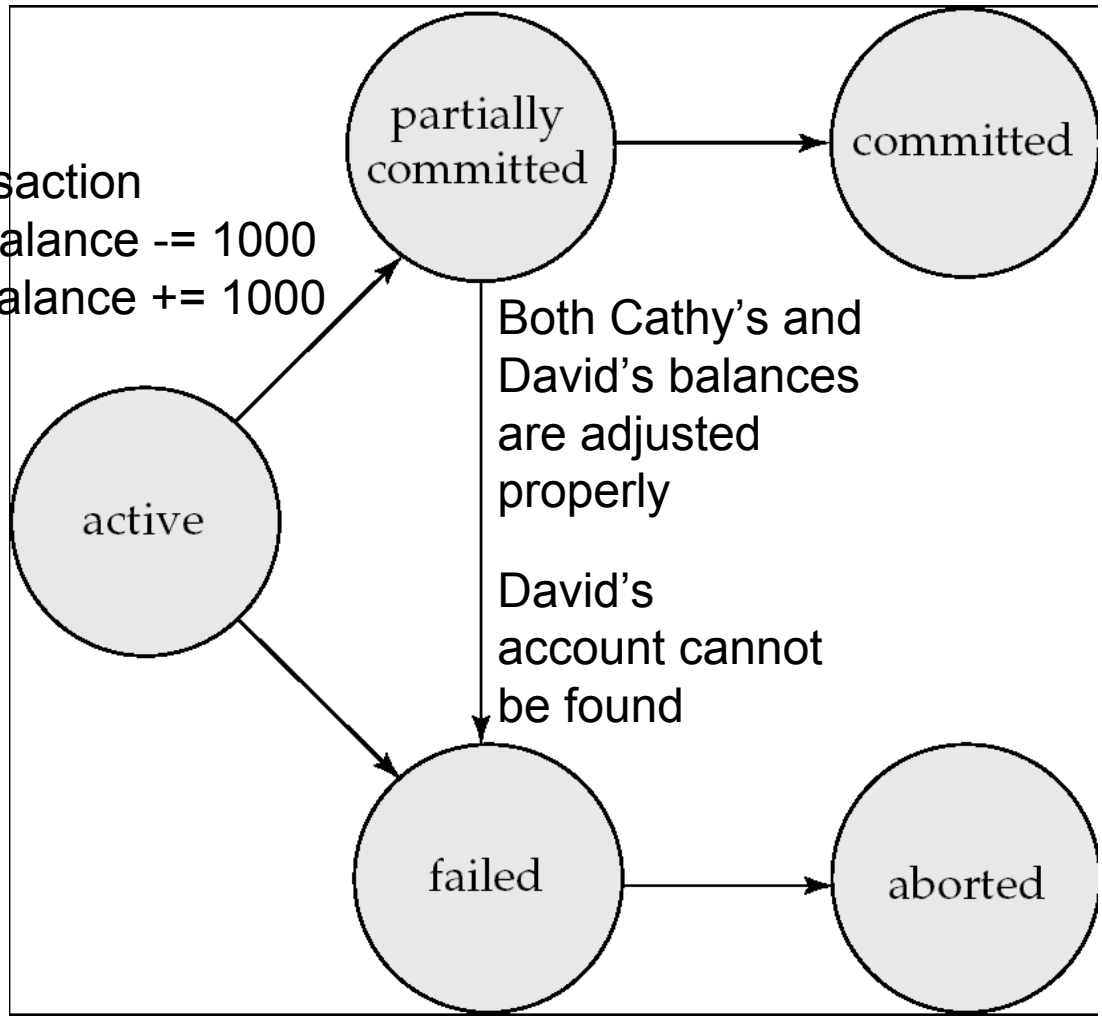
- Cathy transfers $1000 to David
  - Let v be the balance of David's account
  - Set v = v + 1000
- Ben transfers $100 to David
- What if the two transactions interleave?

Cathy's transfer	**Ben's transfer**
Get v (v = 500)	
	Get v (v = 500)
Set v = v + 1000 = 1500	
	Set v = v + 100 = 600

Isolation of Transactions

- The concurrent execution of transactions results in a system state that is equivalent to a state that could have been obtained had these transactions executed one at a time in some order
- One naïve solution: executing transactions serially – one transaction at a time
  - Concurrency is preferred due to performance concern …

State Diagram of a Transaction



Begin transaction
Cathy's balance -= 1000
David's balance += 1000
Commit

Both Cathy's and David's balances are adjusted properly

David's account cannot be found

Enough information is stored so that the balances can be recovered even a system failure happens

The adjustment to Cathy's balance is undone

Transaction States

- Active – the initial state; the transaction stays in this state while it is executing
- Partially committed – after the final statement has been executed
- Failed – after the discovery that normal execution can no longer proceed
- Aborted – after the transaction has been rolled back and the database restored to its state prior to the start of the transaction
- Committed – after successful completion

Handling Failed Transactions

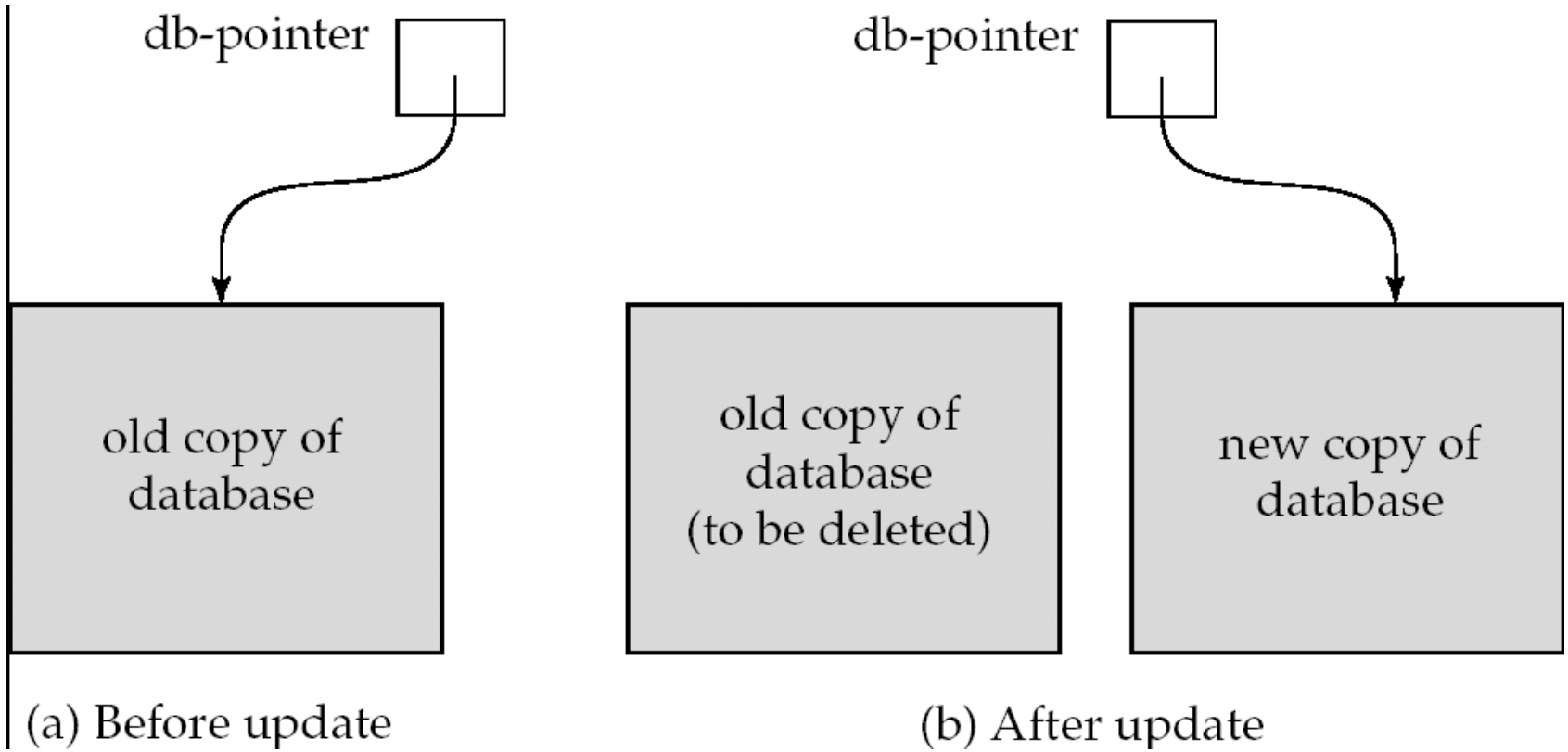
- Restart the transaction: can be done only if no internal logical error
  - Example: the disk storing David's account fails, the transaction can be restarted after the disk is recovered
- Kill the transaction: some internal logical error (the application program needs to be revised), input was bad, the desired data were not found
  - Example: David does not have an account at all

To-Do-List

- What are the ACID properties of transactions?
- Give an example of transactions other than the ones we see in the lecture note and the textbook. Illustrate the ACID properties of transactions and the states of transactions using the example

Shadow Copy

- A simple method for atomicity and durability



Committing a Transaction

- Before a transaction starts, make a copy of the database
- Start the transaction, all updates are conducted on the new database copy
  - No touch on the shadow copy
- When the transaction completes
  - Make sure all pages of the new copy have been written out to disk
  - Update the pointer db-pointer to point to the new copy (an atomic operation in OS)
  - Delete the old copy

Handling Transaction Failures

- If the transaction fails at any time before db-pointer is updated, the shadow copy is not affected
  - Abort the transaction by deleting the new copy
- Once the transaction has been committed, all the updates in the transaction are in the new copy pointed to by db-pointer

Handling System Failures

- After a system failure, the system has to be restarted
- If the system fails at any time before the updated db-pointer is written to disk, when the system restarts, it uses the shadow copy which is consistent – the unfinished transaction is undone
- If the system fails after db-pointer has been updated on disk, when the system restarts, it uses the new copy

Is Shadow Copy Method Efficient?

- Making a copy of a large database is costly
- No concurrent transactions are allowed
  - Cathy transfers $1000 to David
  - Ben transfers $500 from his checking account to his saving account
  - The shadow copy method cannot allow these two transactions concurrent since it cannot make a shadow copy for two transactions
- Idea (to be discussed later): only the updates matter!

Why Concurrent Execution?

- Improved throughput and resource utilization
  - Throughput: the number of transactions executed in a given amount of time
  - Utilization of CPU and disks
- Reduced waiting time
  - Average response time: the average time for a transaction to be completed after it has been submitted

Schedule of Transactions

- A sequences of instructions that specify the chronological order in which instructions of concurrent transactions are executed
- Must consist of all instructions of those transactions
- Must preserve the order in which the instructions appear in each individual transaction
- A transaction that successfully completes its execution will have a commit instructions as the last statement
  - Can be omitted if it is obvious
- A transaction that fails to successfully complete its execution will have an abort instructions as the last statement
  - Can be omitted if it is obvious

Serial Schedules

- T1: transfer $50 from A to B
- T2: transfer 10% of the balance from A to B

T_1	T_2
read(A)	
$A := A - 50$	
write (A)	
read(B)	
$B := B + 50$	
write(B)	
	read(A)
	$temp := A * 0.1$
	$A := A - temp$
	write(A)
	read(B)
	$B := B + temp$
	write(B)

T_1	T_2
	read(A)
	$temp := A * 0.1$
	$A := A - temp$
	write(A)
	read(B)
	$B := B + temp$
	write(B)
read(A)	
$A := A - 50$	
write(A)	
read(B)	
$B := B + 50$	
write(B)	

Serial Schedules

- Each serial schedule consists of a sequence of instructions from various transactions, where the instructions belonging to one single transaction appear together in that schedule
- Serial schedules preserve consistency constraints
  - In our example, A + B is a constant before and after T1 and T2
- If there are n transactions, there are n! possible serial schedules

A Good Concurrent Schedule

- A concurrent schedule is good if it is equivalent to a serial schedule

T_1	T_2
read(A)	
$A := A - 50$	
write(A)	
	read(A)
	$temp := A * 0.1$
	$A := A - temp$
	write(A)
read(B)	
$B := B + 50$	
write(B)	
	read(B)
	$B := B + temp$
	write(B)

A Bad Concurrent Schedule

- A bad concurrent schedule may lead to an inconsistent state

T_1	T_2
read(A)	
$A := A - 50$	
	read(A)
	$temp := A * 0.1$
	$A := A - temp$
	write(A)
	read(B)
write(A)	
read(B)	
$B := B + 50$	
write(B)	
	$B := B + temp$
	write(B)

Serializability

- Each transaction preserves database consistency
  - An assumption about the correctness of applications
- Serial execution of a set of transactions preserves database consistency
- A (possibly concurrent) schedule is serializable if it is equivalent to a serial schedule

Read/Write Operations

- Only consider read and write operations in serializability analysis
- Between a read(Q) instruction and a write(Q) instruction on a data item Q, a transaction may perform an arbitrary sequence of operations on the copy of Q

Read/Write Operations

T_1	T_2
read(A)	
$A := A - 50$	
write(A)	
	read(A)
	$temp := A * 0.1$
	$A := A - temp$
	write(A)
read(B)	
$B := B + 50$	
write(B)	
	read(B)
	$B := B + temp$
	write(B)

T_1	T_2
read(A)	
write(A)	
	read(A)
	write(A)
read(B)	
write(B)	
	read(B)
	write(B)

Swapping Instructions

T_1	T_2
read(A)	
write(A)	
	read(A)
	write(A)
read(B)	
write(B)	
	read(B)
	write(B)

T_1	T_2
read(A)	
write(A)	
	read(A)
read(B)	
	write(A)
write(B)	
	read(B)
	write(B)

When Is Swapping Safe?

- For instructions I1 in transaction T1 and I2 in transaction T2, when <I1, I2> and <I2, I1> have the same effect?
- I1 = op(X) and I2 = op(Y), X ≠ Y, yes
- I1 = read(Q) and I2 = read(Q), yes
- I1 = read(Q) and I2 = write(Q), no
- I1 = write(Q) and I2 = read(Q), no
- I1 = write(Q) and I2 = write(Q), no

Conflict Operations

- I1 and I2 are conflict if
  - I1 and I2 are operations in different transactions
  - I1 and I2 are on the same data item
  - At least one of these operations is a write operation
- If a schedule S can be transformed into a schedule S' by a series of non-conflicting instructions, S and S' are conflict equivalent

Conflict Equivalent Schedules

T_1	T_2
read(A)	
write(A)	
	read(A)
	write(A)
read(B)	
write(B)	
	read(B)
	write(B)

T_1	T_2
read(A)	
write(A)	
read(B)	
write(B)	
	read(A)
	write(A)
	read(B)
	write(B)

Conflict Serializable Schedules

- A schedule S is conflict serializable if it is conflict equivalent to a serial schedule
- Not every schedule is conflict serializable

T_3	T_4
read(Q)	
	write(Q)
write(Q)	

Precedence Graphs for Serial Schedules

- Precedence graph: in a schedule, we draw an edge T1 → T2 if one of the following holds
  - T1 executes write(Q) before T2 executes read(Q)
  - T1 executes read(Q) before T2 executes write(Q)
  - T1 executes write(Q) before T2 executes write(Q)
- The precedence graph of a serial schedule is a directed acyclic graph

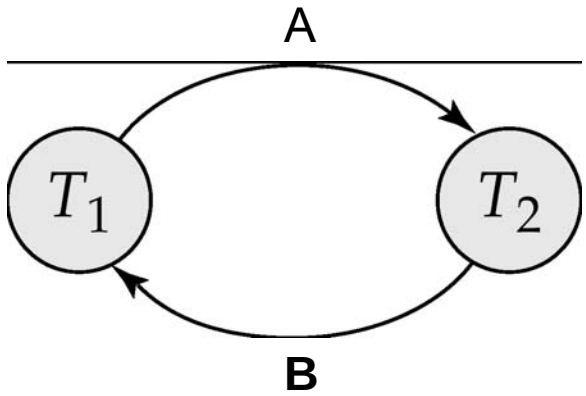
Examples

T_1	T_2
read(A)	
$A := A - 50$	
write (A)	
read(B)	
$B := B + 50$	
write(B)	
	read(A)
	$temp := A * 0.1$
	$A := A - temp$
	write(A)
	read(B)
	$B := B + temp$
	write(B)

T_1	T_2
	read(A)
	$temp := A * 0.1$
	$A := A - temp$
	write(A)
	read(B)
	$B := B + temp$
	write(B)
read(A)	
$A := A - 50$	
write(A)	
read(B)	
$B := B + 50$	
write(B)	

Testing for Conflict Serializability

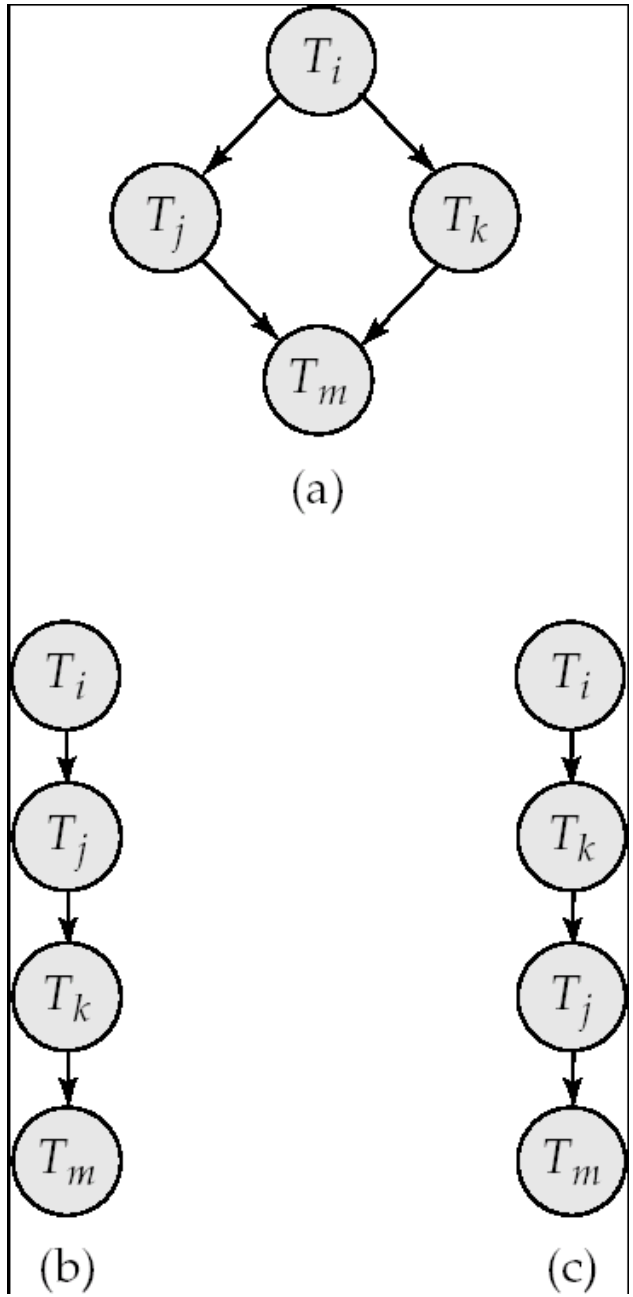
T_1	T_2
read(A)	
$A := A - 50$	
	read(A)
	$temp := A * 0.1$
	$A := A - temp$
	write(A)
	read(B)
write(A)	
read(B)	
$B := B + 50$	
write(B)	
	$B := B + temp$
	write(B)



The schedule cannot be conflict equivalent to a serial schedule

Serializability Order

- If the precedence graph does not have a cycle, we can obtain a serializability order, which is a serial order of transactions, by topological sorting



Equivalent Schedules or Results?

- Two schedules may produce the same outcome, but are not conflict equivalent

T_1	T_5
read(A)	
$A := A - 50$	
write(A)	
	read(B)
	$B := B - 10$
	write(B)
read(B)	
$B := B + 50$	
write(B)	
	read(A)
	$A := A + 10$
	write(A)

<T1, T5>

Result: A = A – 40, B = B + 40

View Equivalent Schedules

A = A0, B = B0

T_1	T_2
read(A)	
$A := A - 50$	
write (A)	
read(B)	
$B := B + 50$	
write(B)	
	read(A)
	$temp := A * 0.1$
	$A := A - temp$
	write(A)
	read(B)
	$B := B + temp$
	write(B)

T_1	T_2
read(A)	
$A := A - 50$	
write(A)	
	read(A)
	$temp := A * 0.1$
	$A := A - temp$
	write(A)
read(B)	
$B := B + 50$	
write(B)	
	read(B)
	$B := B + temp$
	write(B)

A = (A0 – 50) * 0.9, B = B0 + 50 + (A0 – 50) * 0.9

View Equivalence Conditions (1)

- Consider schedules S and S'
  - For each data item Q, if transaction T1 reads the initial value of Q in schedule S, then transaction T1 must also do so in S'
- For each data item, the same transaction reads the item in both S and S'

View Equivalence Conditions (2)

- Consider schedules S and S'
  - For each data item Q, if transaction T1 executes read(Q) in schedule S, and if that value was produced by a write(Q) operation executed by transaction T2, then in S' the read(Q) operation of T1 must also read the value of Q that was produced by the same write(Q) operation of transaction T2
- For each data item, the sequences of updates made by the transactions are the same in both S and S'

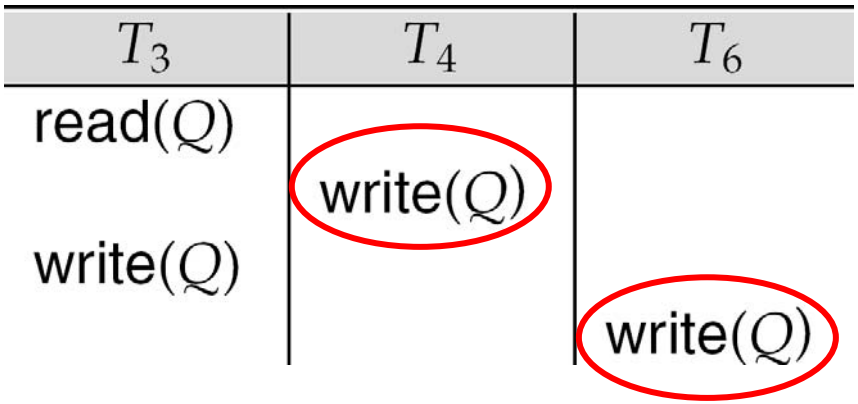
View Equivalence Conditions (3)

- Consider schedules S and S'
  - For each data item Q, the transaction (if any) that performs the final write(Q) operation in schedule S must perform the final write(Q) operation in schedule S'
- For each data item, the same transaction generates the final result

View Serializability

- If two schedules are view equivalent, for each data item, the item is read and updated by the same sequence of transactions in both schedule
- A schedule S is view serializable if it is view equivalent to a serial schedule
- Every conflict serializable schedule is view serializable

Example



View equivalent to <T3, T4, T6>

Not conflict serializable

Blind write: a write operation on a data item without having a read operation on the same item beforehand

If a schedule S is view serializable but not conflict serializable, S contains a blind write

To-Do-List

- What is the intuition behind the differences between conflict serializable schedules and view serializable schedules?
- Prove that every conflict serializable schedule is view serializable
- Prove that If a schedule S is view serializable but not conflict serializable, S contains a blind write
- Can you give an example of two schedules such that they generate the same outcome, but they are still not conflict or view equivalent?

Non-recoverable Schedules

T_8	T_9
read(A)	
write(A)	
	read(A)
read(B)	

T9 commits here!

If T8 fails after T9 commits, we cannot abort T9 since it commits

Recoverable Schedules

- A schedule S is recoverable if for each pair of transactions T1 and T2 in S such that T1 reads a data item previously written by T2, the commit operation of T2 appears before the commit operation of T1
- If T1 fails, T2 can be aborted

Cascading Rollback

T_{10}	T_{11}	T_{12}
read(A)		
read(B)		
write(A)		
	read(A)	
	write(A)	
		read(A)

T10 has not committed yet …

T11 and T12 cannot commit

Drawbacks of cascading rollbacks:

- Once T10 rolls back, T11 and T12 have to roll back
- There can be many active but no progress uncommitted yet transactions – they occupy resources

Summary (Transactions)

- Concept of transactions
- ACID properties of transactions
- Serializability of schedules
- Recoverability of schedules

How to Make up Schedules?

- Given a set of transactions, compute a serializable schedule, and then execute the schedule?
  - Transactions keep arriving
  - Predefined schedule may not fully utilize resources
- We need a mechanism to coordinate transactions such that the resulting schedules are always serializable

Locks

- If several transactions access a data item in an interleaving way, likely the resulting schedule is not serializable
- Idea: data items should be accessed in a mutually exclusive way
  - Lock is an effective mechanism to enforce mutual access to a resource
- Locks
  - Shared-mode lock: if a transaction T has an S lock on item Q, T can read but cannot write Q
  - Exclusive-mode lock: if a transaction has an X lock on item Q, T can both read and write Q

Concurrency Control Manager

- Each transaction should obtain an appropriate lock before it conducts an operation
- Lock manager: concurrency control manager

T1	T2	Concurrency-control manager
Lock-X(B)		
		Grant-X(B, T1)
Read(B)		
B := B – 50		
Write(B)		
Unlock(B)		
	Lock-S(A)	
		Grant-S(A, T2)
	Read(B)	
	Unlock(B)	
	Display(A + B)	
Lock-X(A)		
		Grant-X(A, T1)
Read(A)		
A := A + 50		
Write(A)		
Unlock(A)		

Can Locks Be Shared?

- If T1 locks data item Q, can T2 also locks data item Q?
  - Depending on the types of the locks of T1 and T2
  - Modeled by a compatibility function
- If Q is locked by T1 in an incompatible mode, the concurrency-control manager will not grant the lock, T2 has to wait until all incompatible locks on Q are released

	S	X
S	true	false
X	false	false

Example

- Suppose A = 100, B = 200
- In schedules <T1, T2> and <T2, T1>, output (A + B) is 300

T1:
Lock-X(B);
Read(B);
B := B – 50;
Write(B);
Unlock(B);
Lock_X(A);
Read(A);
A := A + 50;
Write(A);
Unlock(A)

T2:
Lock-S(A);
Read(A);
Unlock(A);
Lock-S(B);
Read(B);
Unlock(B);
Display(A + B)

T1	T2
Lock-X(B)	
Read(B)	
B := B – 50	
Write(B)	
Unlock(B)	
	Lock-S(A)
	Read(B)
	Unlock(B)
	Display(A + B) [250]
Lock-X(A)	T1 releases the lock too early so that T2 sees an inconsistent state in T1!
Read(A)	
A := A + 50	
Write(A)	
Unlock(A)	

Delayed Unlocking

- A transaction can delay unlocking until the end of the transaction

T1:
Lock-X(B);
Read(B);
B := B – 50;
Write(B);
Unlock(B);
Lock_X(A);
Read(A);
A := A + 50;
Write(A);
Unlock(A)

T1':
Lock-X(B);
Read(B);
B := B – 50;
Write(B);
Lock_X(A);
Read(A);
A := A + 50;
Write(A);
Unlock(B);
Unlock(A)

T2' never outputs an inconsistent sum of A and B

T2':
Lock-S(A);
Read(A);
Lock-S(B);
Read(B);
Display(A + B);
Unlock(A);
Unlock(B)

T2:
Lock-S(A);
Read(A);
Unlock(A);
Lock-S(B);
Read(B);
Unlock(B);
Display(A + B)

Deadlock

- Delayed unlocking may lead to deadlock
  - The order of locks matters
- Deadlock: two or more transactions wait for each other, and neither of them can proceed with its normal execution
- Locking protocol: a set of rules that all transactions should follow so that no deadlock happens and the resulting schedule is serializable

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)	

Neither T3 nor T4 can proceed – they wait for each other!

A Simple Locking Protocol

- On an item Q, if transaction T1 locks Q before T2 attempts to lock Q in an incompatible mode, then T1 must access before T2 every item that T2 accesses
  - T1 precedes T2, T1 → T2
- In this protocol, all schedules are conflict serializable
- A simple implementation: a transaction claims all items it wants to access at the beginning

Avoiding Starvation

- Starvation: a transaction has to waited for a long time and may not get the lock on a desired item
  - T1: lock-S(Q)
  - T: lock-X(Q) – wait for Q
  - T2: lock-S(Q) – get it, now Q is S-locked by T1 and T2
  - T1: unlock-S(Q)
  - T3: lock-S(Q) – get it, now Q is S-locked by T2 and T3
  - T2: unlock-S(Q)
  - …
  - T may never get an X-lock on Q
- Solution: an lock on Q is granted only if there is no other transaction that is waiting for a lock on Q

Two-Phase Locking

- Protocol
  - Growing phase: a transaction may obtain locks, but may not release any lock
  - Shrinking phase: a transaction may release locks, but may not obtain any new locks
- The two-phase locking protocol ensures conflict serializability
  - Lock point: the point where the transaction obtains its last lock (i.e., end of its growing phase)
  - Transactions can be serialized in the order of their lock points
  - But several transactions may get into a deadlock

T1	T2
Lock-S(Q1)	
Read(Q1)	
Unlock-S(Q1)	
	…
	Lock-X(Q1)
	Write(Q1)
	Unlock-X(Q1)
	…
	Lock-X(Q2)
	Write(Q2)
	Unlock-X(Q2)
…	
Lock-X(Q2)	
Write(Q2)	
Unlock-X(Q2)	

The above schedule is not allowed in two-phase locking

Cascading Rollback

T_5	T_6	T_7
lock-X(A)		
read(A)		
lock-S(B)		
read(B)		
write(A)		
unlock(A)		
	lock-X(A)	
	read(A)	
	write(A)	
	unlock(A)	
		lock-S(A)
		read(A)

T5 has not committed yet!

If T5 rolls back later, T6 and T7 have to be rolled back

- Strict two-phase locking
  - All X-locks taken by a transaction should be held until that transaction commits
  - No cascading rollbacks in strict two-phase locking: any data written by an uncommitted transaction cannot be read by any other transactions
- Rigorous two-phase locking
  - All locks taken by a transaction should be held until that transaction commits
  - Transactions can be serialized in the order in which they commit

Lock Conversions

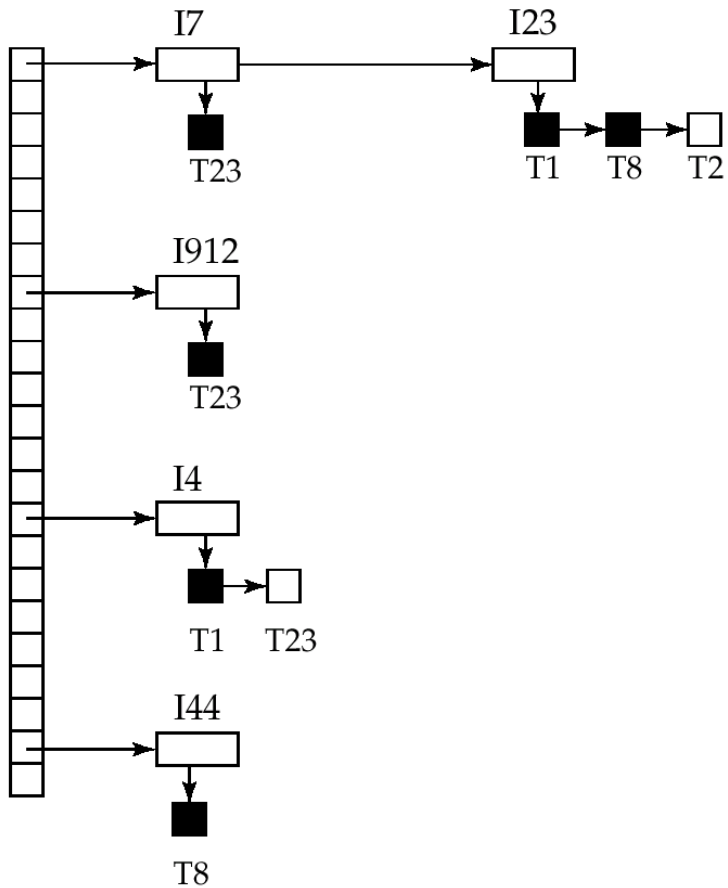
- X-locks exclude concurrency
- conversions
  - Upgrade: from S-lock to X-lock, can happen only in the growing phase, may need to wait
  - Downgrade: from X-lock to S-lock, can happen only in the shrinking phase
- Generate only conflict serializable schedules
- Implementation
  - When a transaction T issues a read(Q) operation, lock-S(Q)
  - When T issues a write(Q) operation, if T holds S-lock(Q), upgrade(Q), else lock-X(Q)
  - All locks obtained by T are unlocked after T commits or aborts

T8:	T9:
Read(a_1);	Read(a_1);
Read(a_2);	Read(a_2);
…	Display(a_1 + a_2)
Read(a_n);	
Write(a_1)	

T_8	T_9
lock-S(a_1)	
	lock-S(a_1)
lock-S(a_2)	
	lock-S(a_2)
lock-S(a_3)	
lock-S(a_4)	
	unlock(a_1)
	unlock(a_2)
lock-S(a_n)	
upgrade(a_1)	

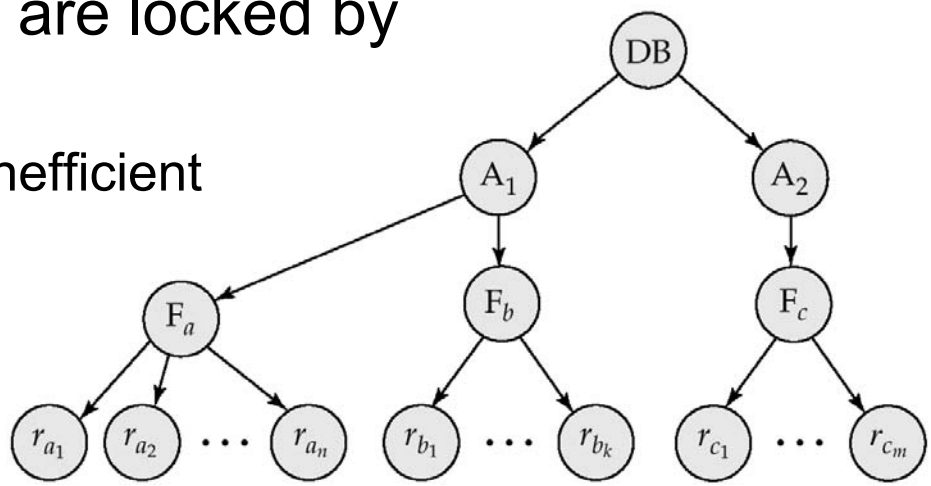
Lock Manager Using a Lock Table

- When a lock request arrives …
- When an unlock request arrives …
- When a transaction aborts …



Multiple Granularity

- What if a transaction needs to access multiple data items?
  - Using many locks is inefficient – lock management overhead, more likely to conflict with other transactions
- Granularity hierarchy
  - When a node is locked, all descendants are locked
- If a transaction wants to lock A1, how can it determine whether some descendants are locked by some other transactions?
  - Checking all descendants is inefficient



Intention Locks

- If a transaction wants to lock a node, all ancestors of the node should be locked in the intention lock mode
  - Intention-shared mode (IS)
  - Intention-exclusive mode (IX)
  - Shared and intention-exclusive mode (SIX)

	IS	IX	S	SIX	X
IS	true	true	true	true	false
IX	true	true	false	false	false
S	true	false	true	false	false
SIX	true	false	false	false	false
X	false	false	false	false	false

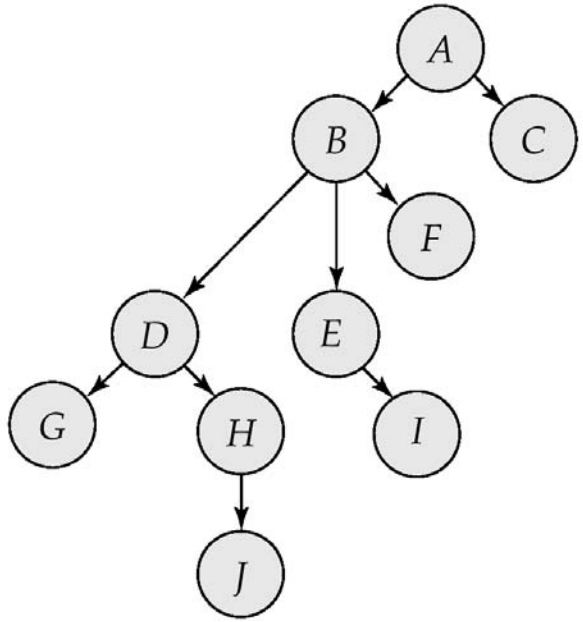
Multiple-granularity Locking Protocol

- A transaction T wants to lock a node Q
  - T must observe the lock-compatibility function
  - T must lock the root first, and can lock it in any mode
  - T can lock Q in S or IS mode only if T currently has the parent of Q locked in either IX or IS mode
  - T can lock Q in X, SIX or IX mode only if T currently has the parent of Q locked in either IX or SIX mode
  - T can lock Q only if T has not previously unlocked any node
  - T can unlock Q only if T currently has none of the children of Q locked
- Multiple-granularity locking enhances concurrency and reduces lock overhead, and is particularly useful when the transaction workload is a mix of short and long transactions

Tree Protocol

- Intuition: if all transactions access data items in the same order, no deadlock can happen and the transactions are conflict serializable
- Only lock-X is allowed
- Each transaction T can lock a data item at most once respecting the following rules
  - The first lock may be on any item
  - Subsequently, a data item Q can be locked only if the parent of Q is currently locked by T
  - Data items can be unlocked at any time
  - A data item that has been locked and unlocked by T cannot subsequently be relocked by T

Tree Protocol Example



T_{10}	T_{11}	T_{12}	T_{13}
lock-X(B)			
	lock-X(D)		
	lock-X(H)		
	unlock(D)		
lock-X(E)			
lock-X(D)			
unlock(B)			
unlock(E)			
		lock-X(B)	
		lock-X(E)	
	unlock(H)		
lock-X(G)			
unlock(D)			
			lock-X(D)
			lock-X(H)
			unlock(D)
			unlock(H)
		unlock(E)	
		unlock(B)	
unlock (G)			

Pros and Cons of Tree Protocol

- Advantages
  - Deadlock free
  - Unlocking may occur earlier
- Disadvantages
  - A transaction may have to lock data items that it does not access
- Enhancements
  - Cascadeless: hold all X-locks until committing
  - Recoverability only: record the transaction performed the last write and a commit dependency

Timestamp-based Protocols

- When a transaction starts, a timestamp is assigned to the transaction
  - Use the value of the system clock
  - Use a logical counter
- For each data item Q, two timestamps are associated with Q
  - W-timestamp(Q): the largest timestamp of any transaction that writes Q successfully
  - R-timestamp(Q): the largest timestamp of any transaction that reads Q successfully
- Intuition: if transactions are organized in a way that at any time a transaction T accesses a data item Q, the timestamp of T is larger than or equal to Q, then the schedule is conflict serializable
  - If all transactions are serialized in the timestamp ascending order, when a transaction T accesses a data item Q, T should have a timestamp larger than that of Q

Examples

T14:
Read(B);
Read(A);
Display(A+B);

T15:
Read(B);
B := B – 50;
Write(B);
Read(A);
A := A + 50;
Write(A);
Display(A+B);

TS(T14)=1	TS(T14)=2	W/R-TS(A, B)=0
Read(B)		R-TS(B)=1
	Read(B)	R-TS(B)=2
	B := B – 50	
	Write(B)	W-TS(B)=2
	Read(A)	R-TS(A)=2
	A := A + 50	
	Write(A)	W-TS(A)=2
Read(A)		Conflict! T14 tries to read an value of A inconsistent with the version it should see

T_{14}	T_{15}
read (B)	
	read (B)
	$B := B - 50$
	write (B)
read (A)	
	read (A)
display $(A + B)$	
	$A := A + 50$
	write (A)
	display $(A + B)$

Timestamp-ordering Protocol

- When transaction T issues read(Q)
  - If TS(T) < W-TS(Q), the read operation is rejected and T is rolled back
    - T needs to read a value of Q that was already overwritten
  - If TS(T) ≥ W-TS(Q), the read operation is executed, and R-TS(Q) is set to max(R-TS(Q), TS(T))
- When transaction T issues write(Q)
  - If TS(T) < R-TS(Q), the write operation is rejected and T is rolled back
    - The value of Q that T is producing was needed previously, and the system assumed that the value would never be produced
  - If TS(T) < W-TS(Q), the write operation is rejected and T is rolled back
    - T is attempting to write an obsolete value of Q
  - Otherwise, the write operation is executed, and W-TS(Q) := TS(Q)

Pros and Cons

- Timestamp-ordering protocol ensures conflict serializability
- No deadlocks
- Starvation may happen to (long) transactions
- May generate nonrecoverable schedules, but can be improved in one of the following ways
  - All writes are conducted at the end of the transaction
  - Reads of uncommitted items are postponed until the transaction that updated the item commits
  - Tracking uncommitted writes: a transaction T can commit only after the commit of any transactions that wrote a value that T read

Unnecessary Rollbacks

T_{16}	T_{17}
read(Q)	
	write(Q)
write(Q)	

- Write(Q) in T_{16} is rejected
  - $TS(T_{16}) < W\text{-}TS(Q)$
  - T_{16} is rolled back
- Does T_{16} really need to be rolled back?
  - If we simply ignore the write(Q) operation in T_{16}, the result is equivalent to $<T_{16}, T_{17}>$
  - Some write operations can be ignored under certain circumstances
- Thomas' write rules: suppose T issues write(Q)
  - If $TS(T) < R\text{-}TS(Q)$, the write operation is rejected and T is rolled back
  - If $TS(T) < W\text{-}TS(Q)$, the write operation can be ignored
  - Otherwise, the write operation is executed and $W\text{-}TS(Q) := TS(T)$
- Thomas' write rules may generate view serializable schedules with better concurrency

Optimistic Concurrency Control

- In all the concurrency control protocols discussed so far, we assume that likely transactions conflict
  - Many write operations
  - Pessimistic concurrency control
- If most of transactions are read-only, then the conflicts happen in a low frequency
  - Even if we do not use any concurrency control, many transactions may not temper the consistency of the database
  - Optimistic concurrency control: a lightweight control may reduce overhead

Validation-based Protocol

- Each trasaction T executes in the following phases
  - Read phase: T reads data, and writes on variables local to T, no writes on the actual database
  - Validation phase: T checks whether it can copy to the database the updates without causing a violation of serializability
  - Write phase: if T succeeds in validation, the updates are applied on the actual database, otherwise, T is rolled back
- T is associated with three timestamps
  - Start(T): the time when T started
  - Validation(T): the time when T finished its read phase and started its validation phase
  - Finish(T): the time when T finished its write phase

Validation

- Validation test for transaction T: for all transactions T' with TS(T') < TS(T)
  - Finish(T') < Start(T); or
  - The set of data items written by T' does not intersect with the set of data items read by T, and T' completes its write phase before T starts its validation phase (i.e., Start(T) < Finish(T') < Validation(T))
    - The writes of T and T' do not overlap
- No cascading rollbacks, but a long transaction may be starved

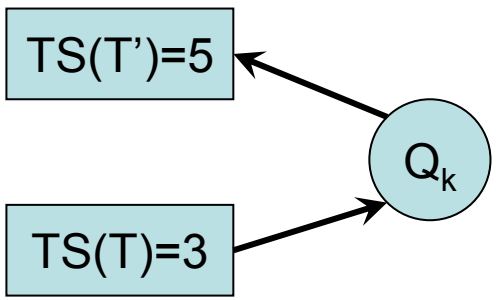
T_{14}	T_{15}
read(B)	
	read(B)
	$B := B - 50$
	read(A)
	$A := A + 50$
read(A)	
$\langle validate \rangle$	
display($A + B$)	
	$\langle validate \rangle$
	write(B)
	write(A)

Multiversion Schemes

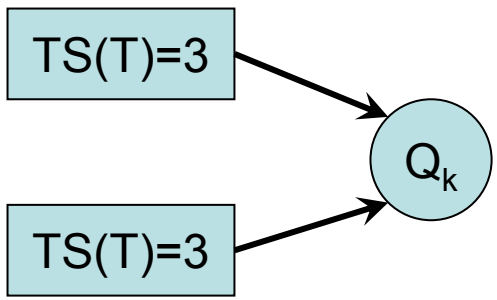
- If in an application read operations are more important than write operation, it would be nice if every read operation succeeds
- Shadow copy: make a copy for every transaction
  - Critical idea: only the data items updated need to be copied
- Multiversion concurrency control themes
  - Each write(Q) operation creates a new version of Q
  - When a transaction issues a read(Q) operation, the concurrency control manager selects one of the versions of Q to be read such that serializability is ensured
- A version of a data item Q
  - The content: the value Q_k of Q
  - W-timestamp(Q_k): the timestamp of the transaction that created version Q_k
  - R-timestamp(Q_k): the largest timestamp of any transaction that successfully reads version Q_k

Multiversion Timestamp Ordering

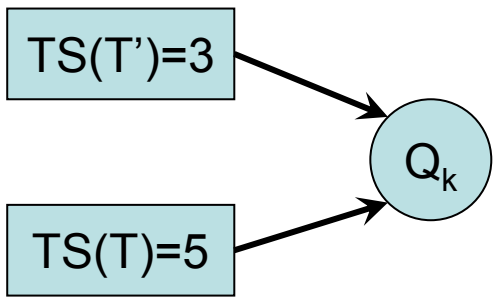
- Protocol
  - If the content of Q_k is written by T, initialize W-timestamp(Q_k) and R-timestamp(Q_k) to TS(T)
  - If T issues a read(Q), return the value of Q_k
    - Never decline a read operation
  - If T issues a write(Q),
    - If TS(T) < R-timestamp(Q_k), roll back T
    - If TS(T) = W-timestamp(Q_k), overwrite the content of Q_k
    - Otherwise, create a new version of Q_k



If T is allowed to write Q_k, T' reads a wrong value



Version Q_k is created by T



A new write operation: create a new version

How Long a Version Should Be Kept

- Q_1 and Q_2 for a data item Q both have a W-timestamp older than the timestamp of the oldest transaction – the transactions creating those two versions committed or aborted
- The older of the two versions can be deleted – when a new transaction reads Q, the latest version should be used
- Pros and cons
  - A read request never fails and never waits
  - Reading a data item needs to update the R-timestamp
  - Conflicts of transactions lead to rolling back
  - Non-recoverable transactions
  - Cascading rollbacks

Multiversion Two-phase Locking

- Integrating multiversion concurrency control and two-phase locking to achieve recoverability and cascadelessness
  - Rigorous two-phase locking
  - Serializable in the commit order
- Timestamp: a global counter ts-counter – incremented by one after a transaction commits
  - TS(T) is the current value of ts-counter
- A read-only transaction T issues Read(Q): return the content of the version whose timestamp is the largest up to TS(T)
  - A read-only transaction never waits
- Update transaction T
  - Read(Q): put an S-lock on Q, read the latest version of Q
  - Write(Q): put an X-lock on Q, create a new version of Q of timestamp ∞ and write the new version
  - Commit: only one update transaction can commit at a time, set the timestamp on every version it has created to ts-counter + 1, increment ts-counter += 1 – updates are available for reading after commit

To-Do-List

- If we lock an item properly right before a data access operation, and release the lock immediately after the access operation, is deadlock still possible to happen? Why?
- Why are all schedules conflict serializable in the simple locking protocol?
- Can you give an example how a long transaction may be starved in the timestamp-ordering protocol?
- Compare the timestamp-ordering protocol and the multiversion timestamp ordering

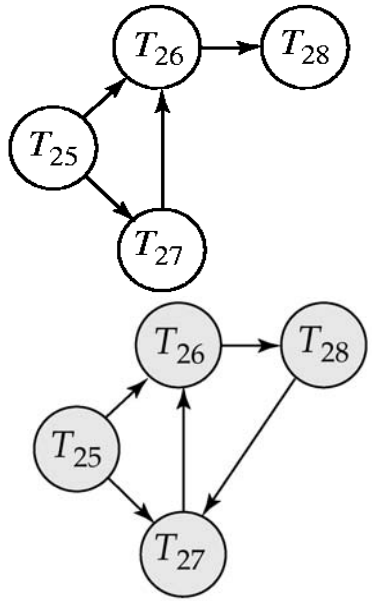
Deadlock Handling

- Deadlock: a set of transactions where each transaction is waiting for another in the set, and none of them can make progress
- Two major methods to handle deadlocks
  - Deadlock prevention: ensure that the system will never enter a deadlock state – commonly used when the probability that the system would enter a deadlock state is high
  - Deadlock detection and deadlock recovery
  - Both methods have to roll back some transactions

Deadlock Prevention

- No cyclic waits
  - Method 1: each transaction locks all data items it needs before its execution
    - Many data items may be locked but unused for a long time
    - Hard to predict all data items a transaction will use precisely
  - Method 2: impose an order of all data items, and lock items in the order (e.g., the tree protocol)
- Preemption: if T2 requests a lock held by T1, we may roll back T1 and grant the lock to T2
  - Wait-die (non-preemption, older waits for younger): if T2 requests a lock held by T1, T2 waits only if TS(T2) is older than TS(T1), otherwise, T2 is rolled back
  - Wound-wait (preemption, younger waits for older): if T2 requests a lock held by T1, T2 waits only if TS(T2) is younger than TS(T1), otherwise, T1 is rolled back
  - No starvation, unnecessary rollbacks may happen

Deadlock Detection & Recovery



- Detection: using a wait-for graph
  - Each vertex is a transaction
  - Edge T1 → T2: T1 waits for a lock held by T2
  - Deadlock: a cycle in the wait-for graph
- Recovery: rolling back some transactions
  - Select a transaction in a deadlock to rollback
    - How much a transaction has done, how much to do
    - How many data items a transaction accessed, how many more to access
    - How many transactions will be involved in the rollback
  - Rollback
    - Total rollback: abort the transaction and restart it
    - Partial rollback:
  - Starvation prevention: no always victim

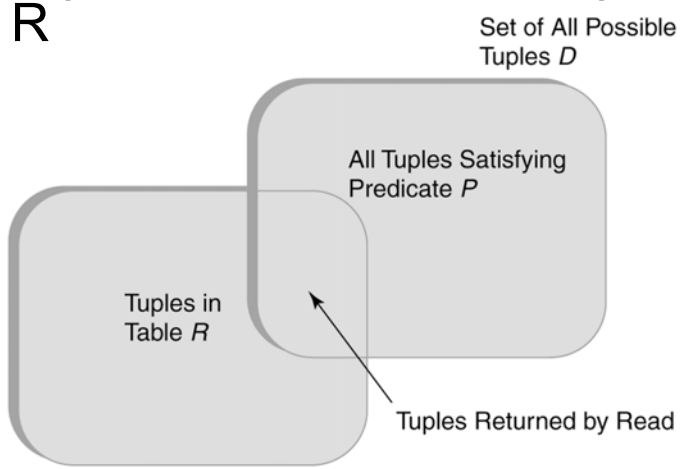
Phantom Tuples

T1	T2
Read account1	
	Insert a new account3
Read account2	
Output the sum of balance	

- T1 and T2 conflict!
  - No common tuples
  - Tuple account3 is a phantom tuple
- When transaction T1 accessing tuples using predicate p, a concurrent transaction T2 insert a tuple t that satisfies p but is not accessed by T1. Tuple t is called a phantom tuple
- Phantom: all tuples satisfying t but are not present in the table when T1 is accessing
- Prevention of phantom
  - Lock the whole table
  - Implement some specific protocol
- Each SQL statement has an associated predicate
  - SELECT or DELETE statement: the condition in the WHERE clause
    - Can be quite complicated in nested statements
  - An UPDATE statement can be rewritten as a DELETE statement and an INSERT statement
    - DELETE statement: condition in the WHERE clause
    - INSERT statement: condition in the SET clause

Predicates and Predicate Locks

- Conflict predicates
  - SELECT * FROM A WHERE Name='Mary'
  - DELETE FROM A WHERE Balance < 10
  - A tuple Name='Mary' and Balance=8 satisfies both predicates
- Two operations conflict if
  - At least one is a write operation
  - The sets of tuples described by the predicates associated with the operations have non-null intersections
- A predicate lock associated with table R on predicate P locks all tuples specified by P, whether or not they are in R



SQL Isolation Levels

- READ UNCOMMITTED: dirty reads (reading uncommitted records) are possible
- READ COMMITTED: dirty reads are not permitted, but successive reads of the same tuple by a particular transaction might yield different values
- REPEATABLE READ: successive reads of the same tuple executed by a particular transaction do not yield different values, but phantoms are possible
- SERIALIZABLE: phantoms are not permitted, transaction execution must be serializable

Level	Dirty Reads	Nonrepeatable Reads	Phantoms
READ UNCOMMITTED	Y	Y	Y
READ COMMITTED	N	Y	Y
REPEATABLE READ	N	N	Y
SERIALIZABLE	N	N	N

Implementation of Isolation Levels

- Many DBMSs use locks
  - Some systems are not lock-based
- Each SQL statement is executed atomically
  - Locking is to guarantee the isolation of multiple statements in one transaction
- All isolation levels use write locks in the same way
- Delayed unlocking is used for X-locks on the predicates associated with UPDATE, INSERT, and DELETE statements
  - Dirty writes are ruled out

Level	Read Locks
READ UNCOMMITTED	None
READ COMMITTED	No delayed unlocking
REPEATABLE READ	Delayed unlocking on tuples returned
SERIALIZABLE	Delayed unlocking on predicates specified in statement

READ UNCOMMITTED/COMMITTED

- READ UNCOMMITTED may corrupt a database
  - T1: w(x) abort
  - T2: r(x) y=f(x) w(y) commit
  - When to use?
    - Read-only transactions, you know the semantics of applications!
    - T1: compute the average balance of accounts
    - T2: find the top-10 accounts with the highest balance
- Implicit non-repeatable read
  - T1: r(x) w(x:=x*2) commit
  - T2: w(x:=0) commit
- Violation of integrity constraints – inconsistent states of database may be read
  - Constraint: x > y, initially, x=10, y=5
  - T1: r(x) x=10 r(y) y=15 commit
  - T2: w(x:=20) w(y:=15) commit
- READ COMMITTED is sufficient for many applications and is default in many DBMSs
  - Airline booking

Lost Updates

- At READ COMMITTED
  - T1: r(x) w(x:=x/2) commit
  - T2: r(x) w(x:=x*2) commit
  - The update of T2 is lost
- With cursors
  - Types of cursors
    - INSENSITIVE cursor: make a copy of the related data
    - Other types of cursors: use pointers to refer to the related data tuples
  - Lost updates
    - T1: claim a non-insensitive cursor C
    - T2: write a data record that covered by C and commit
    - T1: read the updated record, inconsistent!
- Read locks are not non-delayed unlocking at the READ COMMITTED level

CURSOR STABILITY

- As long as a cursor opened by a transaction points to a particular tuple, that tuple cannot be modified or deleted by another transaction
- Implementation
  - a tuple being processed by the iteration is locked in S-mode
  - Any modified tuples are locked in X-mode until the transaction commits
- Extension of READ COMMITTED, strength between READ COMMITTED and REPEATABLE READ
- Update lost still may happen
  - T1: r(x) (cursor) w(x) (cursor) commit
  - T2: r(x) w(x) commit
  - The update of T1 is lost!
  - T2 reads in the middle of T1
- Possible solutions
  - Request a write lock on an item that may be updated later

Rules of Thumb

- Transactions should execute at the lowest level of isolation consistent with application requirements
- The tradeoff between including integrity constraints in the schema so that the DBMS enforces them and encoding enforcement in the transactions should be examined carefully
  - Constraints in schema: costly, no change to transactions when constraints are changed, may allow transactions execute at a lower isolation level
  - Constraints in transactions: efficient, different transactions may not be consistent in constraint checking
- Transactions should be as short as possible
  - Limit the time that locks must be held
  - Decompose long transactions into a sequence of shorter ones
- The frequently invoked transactions should be made as efficiently as possible

Summary (Concurrency Control)

- Concurrency control protocols
  - Lock-based protocols
    - Two-phase locking
    - Graph-based protocols, tree protocol
    - Multiple granularity of locks
  - Timestamp-based protocols
    - The timestamp-ordering protocol
    - Thomas' write rule
  - Validation-based protocols
  - Multiversion schemes
    - Multiversion timestamp ordering
    - Multiversion two-phase locking
- Deadlock handling
  - Prevention
  - Detection and recovery
- Phantom tuples and weak levels of consistency
  - READ UNCOMMITTED, READ COMMITTED, REPEATABLE READ, and SERIALIZABLE

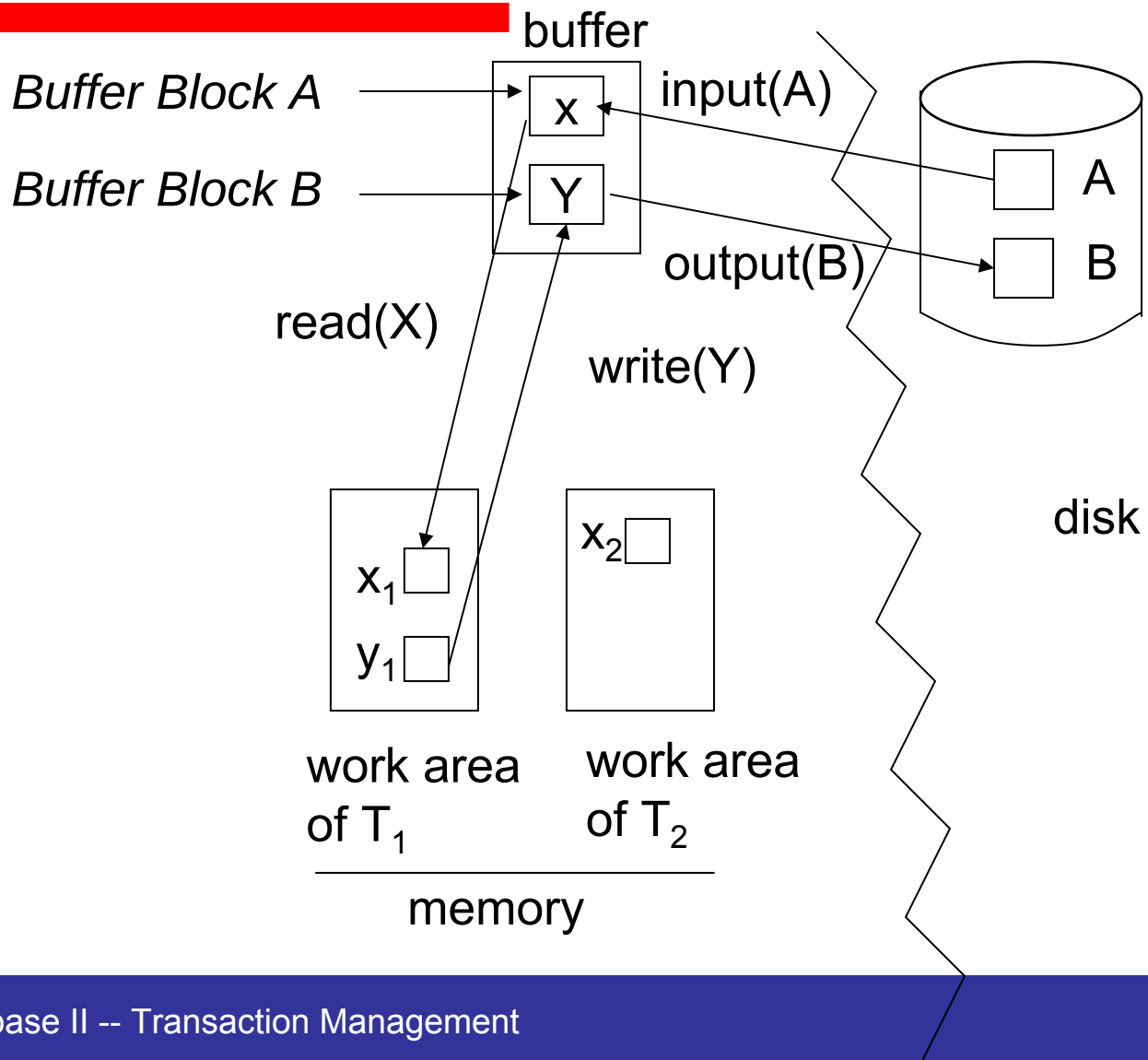
Types of Failure

- Transaction failure: a transaction fails
  - Logical error: failure due to some internal condition, e.g., bad input, data not found, overflow, resource limit exceeded
  - System error: system enters an undesirable state, e.g., deadlock
- System crash: loss of the content of volatile storage, but the content of nonvolatile storage remains intact (fail-stop assumption)
- Disk failure: copies of data on other disk or archival backups on tertiary media are used to recover
- Recovery scheme: restore the database to a consistent state (may not be the last one) that existed before the failure
- Recovery algorithms
  - Maintaining sufficient information during normal transaction processing to allow recovery from failures
  - Recovering the database contents to a state to ensure consistency, atomicity, and durability

Storage Types and Operations

- Volatile storage: main memory, cache
  - Data in volatile storage does not survive system crashes
- Nonvolatile storage: disks, tapes
  - Data in nonvolatile storage survives system crashes
- Stable storage: data in stable storage is never lost
  - Two physical blocks is kept for each logical disk block
  - Write protocol: write succeeds only if both steps are done
    - Write the first physical block
    - Write the same information to the second physical block
  - Recovery
    - If both blocks are the same, and no error in checksum, no recovery is needed
    - If one block has error in checksum, use the other block to recover
    - If both blocks have no error but are not identical, use the content in the second block to replace that in the first block
- Data access operations
  - Read(X), write(X): read and write data in buffer, if the target data item is not in main memory, load it into main memory before the operation
    - Updates are not durable if they are not written onto disk
  - Input(B), output(B): transfer a physical block into main memory/write onto disk

Data Access Operations



Recovery and Log

- What if a failure happens in the middle of transaction?
  - No matter re-executing T or not the database is in an inconsistent state
  - Reason: no information is stored to ensure recovery
- Log: a widely used structure for recording database modifications
  - A sequence of log records stored on stable storage
  - Four types of log records
    - <T, start>
    - <T, X, V_{old}, V_{new}>
    - <T, commit>
    - <T, abort>

Transferring 500 from A to B

T	A	B
Start	1000	2000
A = A – 500	500	2000
B = B + 500	500	2500
Commit	500	2500

Deferred Database Modification

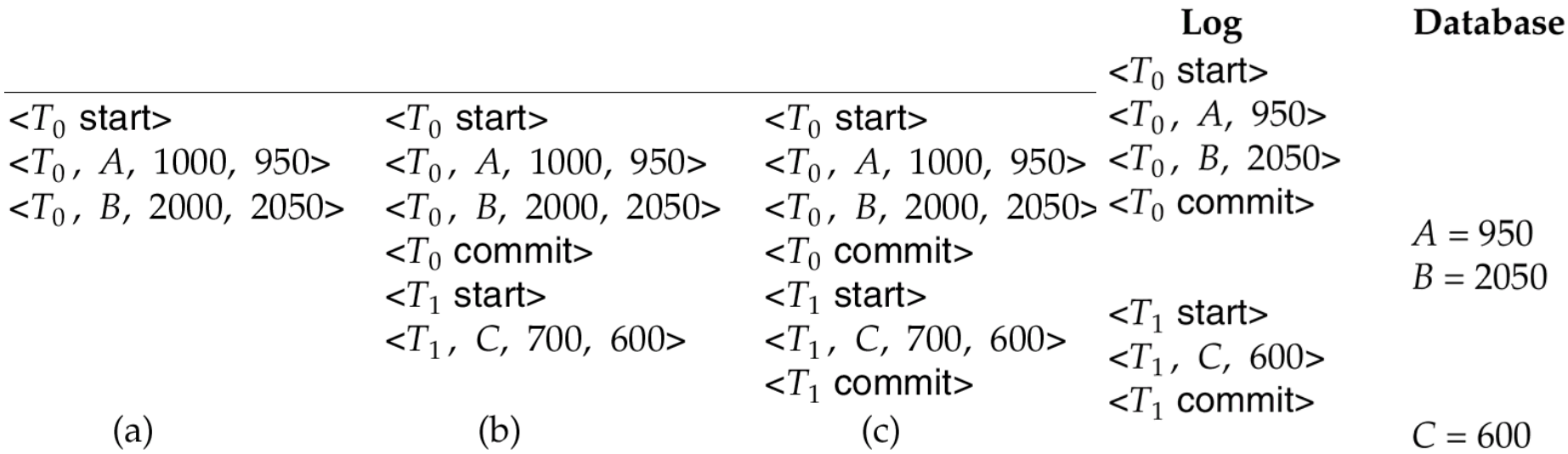
- Record all modifications in the log
- Defer the execution of all write operations until the transaction partially commits
  - Use the log in executing the deferred writes
- Protocol
  - Write <T, start> to the log when T starts
  - Write <T, X, V_{old}, V_{new}> when a write is requested in the transaction
  - When T finishes the last operation, write <T, commit> to the log
  - Update the database according to the log
    - Redo: set the values of all data items updated by transaction T to the new values
    - Redo is idempotent

T0	
Read(A);	<T_0 start>
A := A – 50;	<T_0, A, 950>
Write(A);	<T_0, B, 2050>
Read(B);	<T_0 commit>
B := B + 50;	<T_1 start>
Write(B)	<T_1, C, 600>
T1	<T_1 commit>
Read(C);	
C := C – 100;	
Write(C);	

Log	Database
<T_0 start>	
<T_0, A, 950>	
<T_0, B, 2050>	
<T_0 commit>	
	$A = 950$
	$B = 2050$
<T_1 start>	
<T_1, C, 600>	
<T_1 commit>	
	$C = 600$

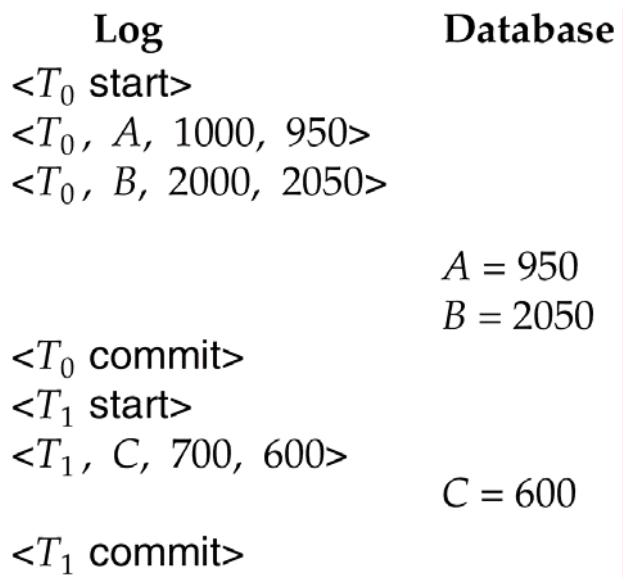
Recovery Using Log

- After a failure, a transaction T needs to be redone if and only if the log contains both <T, start> and <T, commit>
- Examples
  - Case 1: a crash occurs just after the log record of write(B)
  - Case 2: a crash occurs just after the log record of write(C)
  - Case 3: a crash occurs just after the log record <T1, commit>
  - Case 4: a crash occurs in the recovery procedure of case 3



Immediate Database Modification

- Undo(T): restore the values of all data items updated by transaction T to the old values
- Undo-redo in recovery
  - Transaction T needs to be undone if the log contains <T, start> but does not contain <T, commit>
  - Transaction T needs to be redone if the log contains <T, start> and <T, commit>
  - Case 1: a crash occurs just after write(B)
  - Case 2: a crash occurs just after write(C)
  - Case 3: a crash occurs just after <T1, commit>



Checkpoints

- Cost in recovery using a large log file
  - Searching a large log file to determine transactions to be redone and undone is time consuming
  - Many transactions need to be redone may have written their updates into the database
- Checkpoint: “synchronize” the database
  - Output onto stable storage all records currently residing in main memory
  - Output to disk all modified buffer blocks
  - Output onto stable storage a log record <checkpoint>
  - Transactions are not allowed to perform any update actions when a checkpoint is in progress
- Using checkpoints in recovery
  - A transaction T whose record <T, commit> appears before a checkpoint does not need to be redone – updated data already in database
  - Search the log backward to find all transactions T started but have not committed before the last checkpoint
    - Undo all transactions T which do not have a <T, commit> record in the log
    - Redo all transactions T which have a <T, commit> record in the log after the last checkpoint

Log-record Buffering

- Writing log files using buffers: log buffers may fail and thus be lost
- Log force
  - Transaction T enters the commit state after the <T, commit> log record has been output to stable storage
  - Before the <T, commit> log record can be output to stable storage, all log records pertaining to T must have been output to stable storage
  - (Write-ahead logging, WAL) Before a block of data in main memory can be output to the database, all log records pertaining to data in the block must have been output to stable storage

Rolling Back and Restart Recovery

- Scan the log backward, for each record <T, X, V_{old}, V_{new}> restore the data item X to its old value V_{old}, until record <T, start> is found
  - The transaction is rolled back correctly if strict two-phase locking is used
- Restart recovery: recovery from crash
  - Compute redo-list and undo-list by scanning the log backward, until the first <checkpoint> record
    - For each record found of the form <T, commit>, insert T into the redo-list
    - For each record found of the form <T, start> and T not in the redo-list, insert T into the undo-list
  - Recovery algorithm
    - Rescan the log backward, undo every log record belonging to a transaction T in the undo-list, until all <T, start> records are found for all transactions in the undo-list
    - Find the latest <checkpoint> record
    - Scan the log forward, redo every log record belong to a transaction T in the redo-list

Summary

- Using log for recovery
- Log structure
- Deferred database modification
- Immediate database modification
- Checkpoints and recovery
- Log-record buffering
- Rollback and restart recovery