


Unit 4

Transaction Management



Objective

- ❑ What is mean by transaction?
 - ❑ ACID properties
 - ❑ Why Concurrency control is needed
 - ❑ Concept of schedule
 - ❑ Serializability
 - ❑ Concurrency control techniques
 - ❑ Deadlock
 - ❑ Different Crash Recovery methods
-



Transaction Management

- A **transaction** is one or more SQL statements that make up a unit of work performed against the database, and either all the statements in a transaction are committed as a unit or all the statements are rolled back as a unit.
 - This unit of work typically satisfies a user request and ensures data integrity.
-



Transaction Management

- ❑ TPS are the systems with large databases
 - ❑ Hundreds of concurrent users executing database transactions
 - ❑ Examples are airline reservations, banking, credit card processing, stock markets, supermarket and so on
 - ❑ These system requires high availability & fast response time
-



Transaction Management

- When you transfer money from one bank account to another, the request involves a transaction: updating values stored in the database for both accounts.
 - For a transaction to be completed and database changes to be made permanent, a transaction must be completed in its entirety.
-



Transaction Management

- ❑ A transaction is a sequence of logical DML operations
 - ❑ A transaction is said to be completed when we issue a commit.
 - ❑ Commit - saves the changes permanently
 - ❑ Changes made through a transaction can be undone by the use of rollback
 - ❑ Rollback after commit is useless
-



Properties of Transactions

- For a transaction to be called successful it must satisfy ACID properties
 - **ACID** (*atomicity, consistency, isolation, durability*) is a set of properties that guarantee database transactions are processed reliably
 - Jim Gray defined these properties of a reliable transaction system in the late 1970s
-



ACID Properties of Transactions

□ Atomicity

- All changes to data are performed as if they are a single operation. That is, all the changes are performed, or none of them are.
 - For example, in an application that transfers funds from one account to another, the atomicity property ensures that, if a debit is made successfully from one account, the corresponding credit is made to the other account.
-



ACID Properties of Transactions

□ Consistency

- Data is in a consistent state when a transaction starts and when it ends.
 - For example, in an application that transfers funds from one account to another, the consistency property ensures that the total value of funds in both the accounts is the same at the start and end of each transaction.
-



ACID Properties of Transactions

□ Isolation

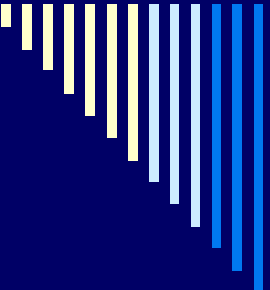
- The intermediate state of a transaction is invisible to other transactions. As a result, transactions that run concurrently appear to be serialized.
 - For example, in an application that transfers funds from one account to another, the isolation property ensures that another transaction sees the transferred funds in one account or the other, but not in both, nor in neither.
-



ACID Properties of Transactions

□ Durability

- After a transaction successfully completes, changes to data persist and are not undone, even in the event of a system failure.
 - For example, in an application that transfers funds from one account to another, the durability property ensures that the changes made to each account will not be reversed.
-



Single user Vs Multiuser systems

- ❑ A DBMS is a **single-user** if at most one user at a time can use system
 - ❑ In **multiuser**, many users can use the system
 - ❑ Hence access the database concurrently
 - ❑ Single user DBMS are mostly restricted to personal computer system
 - ❑ Most other DBMS are multiuser
-



Concurrency

- It is property of system in which several computations are executing simultaneously and interacting with each other
 - Concurrent processes often need access to shared data and shared resources
 - Concurrent execution of processes is interleaved
 - Interleaving also prevents a long process from delaying other process
-



Concurrency control

- ❑ Concurrency control deals with preventing concurrently running processes from improperly inserting, deleting, or updating the same data.
 - ❑ concurrency control ensures that correct results for concurrent operations are generated, while getting those results as quickly as possible.
 - ❑ Concurrency control in Database management systems ensures that database transactions are performed concurrently without violating the data integrity of the respective databases.
-



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 of concurrency control

- ❑ If concurrent transactions with interleaving operations are allowed in an uncontrolled manner, some unexpected result may occur.
 - ❑ Three basic problems caused by concurrency:
 - ❑ Lost update problem
 - ❑ Uncommitted dependency problem
 - ❑ Inconsistent analysis problem.
-



Lost Update Problem

- ❑ Successfully completed update is overridden by another user.
 - ❑ Example:
 - ❑ T1 withdraws £10 from an account with balx, initially £100.
 - ❑ T2 deposits £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 T2's update!!

This can be avoided by preventing T1 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.
 - ❑ Example:
 - ❑ T4 updates balx to £200 but it aborts, so balx should be back at original value of £100.
 - ❑ T3 has read new value of balx (£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 T3 from reading bal_x until after T4 commits or aborts.



Inconsistent Analysis Problem

- Occurs when transaction reads several values but second transaction updates some of them during execution of first.
 - Example:
 - T6 is totaling balances of account x (£100), account y (£50), and account z (£25).
 - Meantime, T5 has transferred £10 from balx to balz, so T6 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 T₆ from reading bal_x and bal_z until after T₅ completed updates.

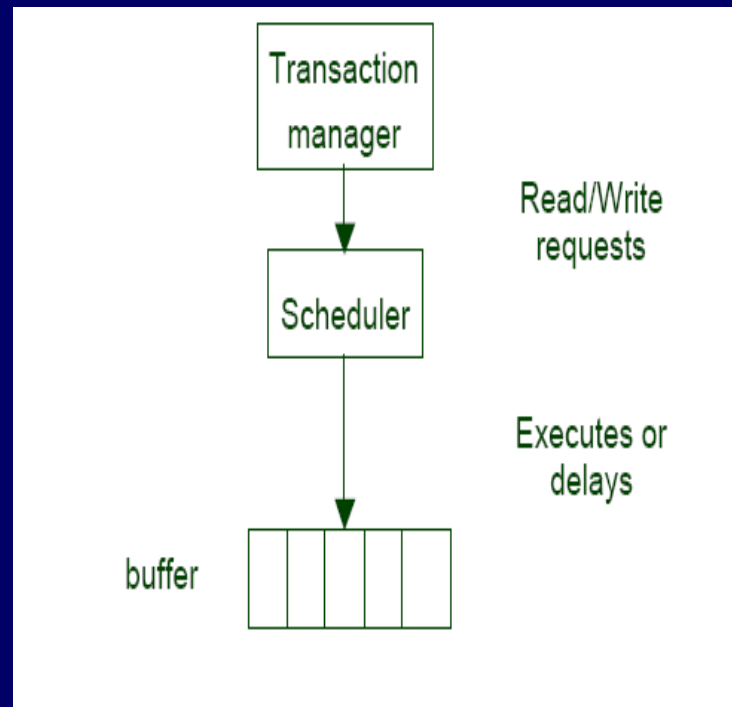


Serializability

- ❑ Objective of a concurrency control protocol is to schedule transactions in such a way as to avoid any interference.
 - ❑ Possible solution: Run all transactions serially.
 - ❑ This is often too restrictive as it limits degree of concurrency or parallelism in system.
 - ❑ Serializability identifies those executions of transactions guaranteed to ensure consistency.
-

The scheduler

- The scheduler component of a DBMS must ensure that the individual steps of different transactions preserve consistency.





Schedule

- Executions of transactions running in the system
 - Serial Schedule: a schedule where the operations of each transaction are executed consecutively without any interleaved operations from other transactions.
-



Serial Schedule

$D =$	$T1$	$T2$	$T3$
	$R(X)$		
	$W(X)$		
	$Com.$		
		$R(Y)$	
		$W(Y)$	
		$Com.$	
			$R(Z)$
			$W(Z)$
			$Com.$

$D = R1(X) W1(X) Com1 R2(Y) W2(Y) Com2 R3(Z) W3(Z) Com3$



Nonserial Schedule:

- Schedule where operations from a set of concurrent transactions are interleaved.
 - The 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 - some important rules

□ 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 completely 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)	



Conflict Serializability

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



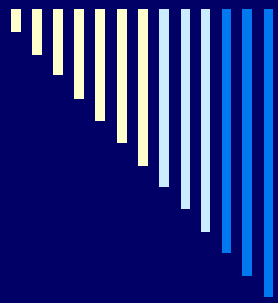
View Serializability

- Offers less stringent definition of schedule equivalence than conflict serializability.
 - Two schedules S1 and S2 are view equivalent if:
 - For each data item x , if T_i reads initial value of x in S1, T_i must also read initial value of x in S2.
 - For each read on x by T_i in S1, if value read by x is written by T_j , T_i must also read value of x produced by T_j in S2.
 - For each data item x , if last write on x performed by T_i in S1, same transaction must perform final write on x in S2.
-



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



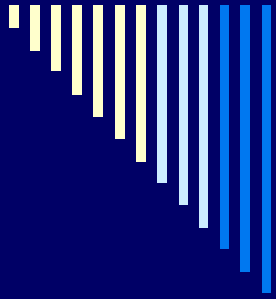
Recoverability

- ❑ 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 .
-



HOW CAN THE DBMS ENSURE SERIALIZABILITY?



Concurrency control mechanisms

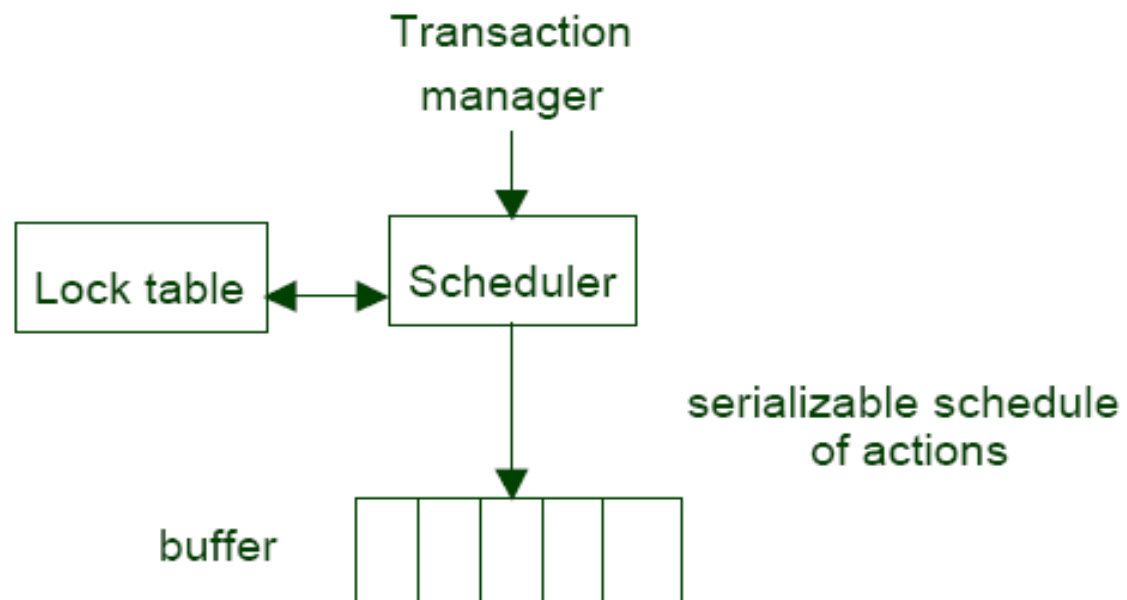
- Two basic concurrency control techniques:
 - Locking methods
 - Timestamping
 - Both are conservative approaches: delay transactions in case they conflict with other transactions.
 - Optimistic methods assume conflict is rare and only check for conflicts at commit.
-



Locking

- Transaction uses locks to deny access to other transactions and so prevent incorrect updates.
 - Generally, a transaction must claim a
 - read (shared), or
 - write (exclusive)
 - 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





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.
 - ❑ Some systems allow transaction to upgrade a shared lock to an exclusive lock, or vice-versa.
-



Example - Incorrect Locking Schedule

Time	T9	T10
T1	begin_transaction	
t2	read(balx)	
t3	balx = balx + 100	
T4	write(balx)	begin_transaction
T5		read(balx)
T6		balx = balx * 1.1
T7		write(balx)
t8		read(baly)
t9		baly = baly * 1.1
T10		write(baly)
t11	read(baly)	commit
t12	baly = baly - 100	
T13	write(baly)	
T14	commit	



Example - Incorrect Locking Schedule

- If at start, $balx = 100$, $baly = 400$, result should be:
 - $balx = 220$, $baly = 330$, if T9 executes before T10, or
 - $balx = 210$, $baly = 340$, if T10 executes before T9 .
 - However, result gives $balx = 220$ and $baly = 340$.
-



Example - Incorrect Locking Schedule

A valid schedule using the basic rules of locking is:

$S = \{ \text{write_lock}(T9, \text{balx}), \text{read}(T9, \text{balx}), \text{write}(T9, \text{balx}), \text{unlock}(T9, \text{balx}), \text{write_lock}(T10, \text{balx}), \text{read}(T10, \text{balx}), \text{write}(T10, \text{balx}), \text{unlock}(T10, \text{balx}), \text{write_lock}(T10, \text{baly}), \text{read}(T10, \text{baly}), \text{write}(T10, \text{baly}), \text{unlock}(T10, \text{baly}), \text{commit}(T10), \text{write_lock}(T9, \text{baly}), \text{read}(T9, \text{baly}), \text{write}(T9, \text{baly}), \text{unlock}(T9, \text{baly}), \text{commit}(T9) \}$



Example - Incorrect Locking Schedule

- Problems arise when 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)

- A transaction follows 2PL protocol if all locking operations precede first unlock operation in the transaction.
 - We can split the transaction in two phases:
 - 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



Locking methods: problems

- If every transaction in a schedule follows 2PL, schedule is serializable.
 - However, problems can occur with interpretation of when locks can be released.
 - Deadlock:
A bottleneck that may result when two (or more) transactions are each waiting for locks held by the other to be released.
-



Deadlock - an example

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 - possible solutions?

- 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).
 - Two general techniques for handling deadlock:
 - 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

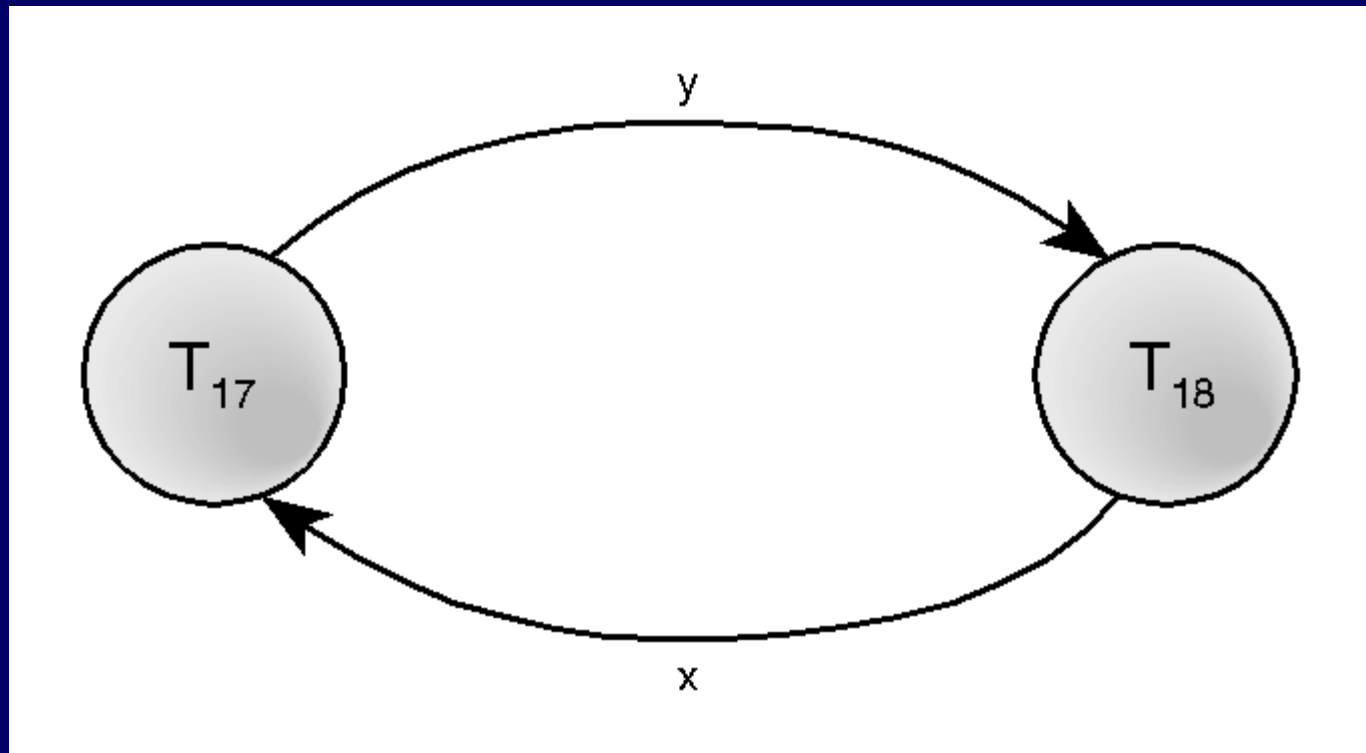
- DBMS looks ahead to see if transaction would cause deadlock and never allows deadlock to occur.
 - Could order transactions using transaction timestamps:
 - Wait-Die - only an older transaction can wait for younger one, otherwise transaction is aborted (dies) and restarted with same timestamp.
 - Wound-Wait - 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 $T_i \rightarrow T_j$, 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 earlier timestamps, get priority in the event of conflict.
 - Conflict is resolved by rolling back and restarting transaction.
 - No locks so no deadlock.
-



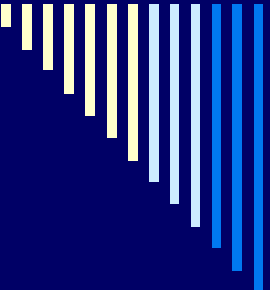
Timestamping

- 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
 - incrementing a logical counter every time a new transaction starts.
-



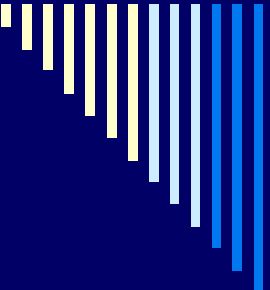
Timestamping - definition

- Timestamping: a concurrency control protocol that orders transactions in such a way that older transactions get priority in the event of a conflict.
 - 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: how does the protocol work?

- A transaction T with timestamp $ts(T)$ wants to read(x):
 - $ts(T) < write_timestamp(x)$
 - x already updated by younger (later) transaction.
 - Transaction T must be aborted and restarted with a new timestamp.
 - $ts(T) \geq write_timestamp(x)$
 - transaction can proceed
 - $read_timestamp = \max(ts(T), read_timestamp(x))$
-



Timestamping: how does the protocol work?

- A transaction T with timestamp $ts(T)$ wants to $write(x)$:
 - $ts(T) < read_timestamp(x)$
 - younger transaction has read the value x
 - rollback transaction T and restart using a later timestamp
 - $ts(T) < write_timestamp(x)$
 - x already written by younger transaction.
 - Write can safely be ignored - ignore obsolete write rule.
 - all other cases: operation accepted and executed.
-

Example

Time	Op	T ₁₉	T ₂₀	T ₂₁
t ₁		begin_transaction		
t ₂	read(bal_x)	read(bal_x)		
t ₃	bal_x = bal_x + 10	bal_x = bal_x + 10		
t ₄	write(bal_x)	write(bal_x)	begin_transaction	
t ₅	read(bal_y)		read(bal_y)	
t ₆	bal_y = bal_y + 20		bal_y = bal_y + 20	begin_transaction
t ₇	read(bal_y)			read(bal_y)
t ₈	write(bal_y)		write(bal_y) ⁺	
t ₉	bal_y = bal_y + 30			bal_y = bal_y + 30
t ₁₀	write(bal_y)			write(bal_y)
t ₁₁	bal_z = 100			bal_z = 100
t ₁₂	write(bal_z)			write(bal_z)
t ₁₃	bal_z = 50	bal_z = 50		commit
t ₁₄	write(bal_z)	write(bal_z) [‡]	begin_transaction	
t ₁₅	read(bal_y)	commit	read(bal_y)	
t ₁₆	bal_y = bal_y + 20		bal_y = bal_y + 20	
t ₁₇	write(bal_y)		write(bal_y)	
t ₁₈			commit	

⁺ At time t₈, the write by transaction T₂₀ violates the first timestamping write rule described above and therefore is aborted and restarted at time t₁₄.

[‡] At time t₁₄, the write by transaction T₁₉ can safely be ignored using the ignore obsolete write rule, as it would have been overwritten by the write of transaction T₂₁ at time t₁₂.



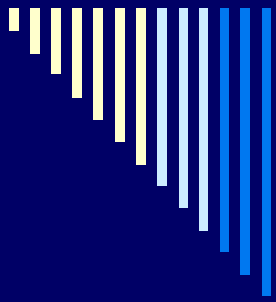
Rare Conflicts

- ❑ In some systems we can safely assume that conflicts are rare.
 - ❑ Do we really need locking or timestamping protocols?
 - ❑ No. More efficient techniques can be used to let transactions proceed without delays to ensure serializability.
-



Optimistic Techniques

- 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.
 - Three phases:
 - Read
 - Validation
 - Write.
-



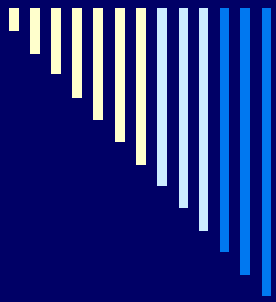
Optimistic Techniques - Read Phase

- 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.



Failure Classification

□ Transaction failure :

- **Logical errors:** transaction cannot complete due to some internal error condition
- **System errors:** the database system must terminate an active transaction due to an error condition (e.g., deadlock)

□ **System crash:** a power failure or other hardware or software failure causes the system to crash.

□ **Disk failure:** a head crash or similar disk failure destroys all or part of disk storage



Recovery Algorithms

- Recovery algorithms are techniques to ensure database consistency and transaction atomicity and durability despite failures
 - Recovery algorithms have two parts
 1. Actions taken during normal transaction processing to ensure enough information exists to recover from failures
 2. Actions taken after a failure to recover the database contents to a state that ensures atomicity, consistency and durability
-



Different Crash Recovery methods

- To ensure atomicity despite failures, we first output information describing the modifications to stable storage without modifying the database itself.
 - We study two approaches:
 - log-based recovery, and
 - shadow-paging
 - We assume (initially) that transactions run serially, that is, one after the other.
-



Shadow-Paging

- This scheme is useful if transactions execute serially
 - Idea: maintain *two* page tables during the lifetime of a transaction –the **current page table**, and the **shadow page table**
 - Store the shadow page table in nonvolatile storage, such that state of the database prior to transaction execution may be recovered.
 - Shadow page table is never modified during execution
-



Shadow-Paging

- To start with, both the page tables are identical. Only current page table is used for data item accesses during execution of the transaction.
 - Whenever any page is about to be written for the first time
 - A copy of this page is made onto an unused page.
 - The current page table is then made to point to the copy
 - The update is performed on the copy
-



Shadow Paging

- Advantages of shadow-paging over log-based schemes
 - no overhead of writing log records
 - recovery is trivial
-



Shadow Paging

❑ Disadvantages :

- Copying the entire page table is very expensive
 - ❑ Can be reduced by using a page table structured like a B⁺-tree
 - Commit overhead is high even with above extension
 - ❑ Need to flush every updated page, and page table
 - Data gets fragmented (related pages get separated on disk)
 - After every transaction completion, the database pages containing old versions of modified data need to be garbage collected
 - Hard to extend algorithm to allow transactions to run concurrently
 - ❑ Easier to extend log based schemes
-



Log-Based Recovery

- A **log** is kept on stable storage.
 - The log is a sequence of **log records**, and maintains a record of update activities on the database.
 - **Two approaches using logs:**
 - **Deferred database modification**
 - **Immediate database modification**
-



Deferred Database Modification

- The **deferred database modification** scheme records all modifications to the log, but defers all the **writes** to after partial commit. (final action of the tx. Is executed)
 - Crashes can occur while
 - the transaction is executing the original updates, or
 - while recovery action is being taken
-



Immediate Database Modification

- The immediate database modification scheme allows database updates of an uncommitted transaction to be made as the writes are issued
 - since undoing may be needed, update logs must have both old value and new value
- Update log record must be written *before* database item is written
 - We assume that the log record is output directly to stable storage
 - Can be extended to postpone log record output, so long as prior to execution of an output(B) operation for a data block B , all log records corresponding to items B must be flushed to stable storage



Immediate Database Modification

- ❑ Output of updated blocks can take place at any time before or after transaction commit
 - ❑ Order in which blocks are output can be different from the order in which they are written.
-



Checkpoints

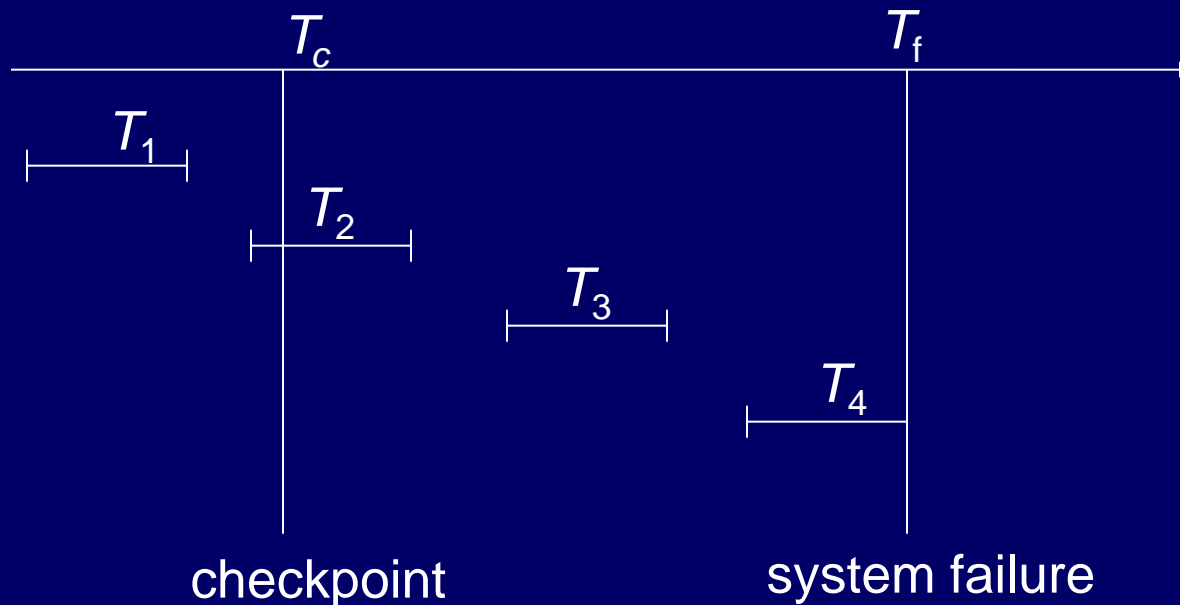
- Problems in recovery procedure as discussed earlier :
 1. searching the entire log is time-consuming
 2. we might unnecessarily redo transactions which have already
 3. output their updates to the database.
 - Streamline recovery procedure by periodically performing **checkpointing**
 1. Output all log records currently residing in main memory onto stable storage.
 2. Output all modified buffer blocks to the disk.
 3. Write a log record < **checkpoint** > onto stable storage.
-



Checkpoints

- During recovery we need to consider only the most recent transaction T_i that started before the checkpoint, and transactions that started after T_i .
 1. Scan backwards from end of log to find the most recent <checkpoint> record
 2. Continue scanning backwards till a record < T_i start> is found.
 3. Need only consider the part of log following above start record. Earlier part of log can be ignored during recovery, and can be erased whenever desired.
 4. For all transactions (starting from T_i or later) with no < T_i commit>, execute $\text{undo}(T_i)$. (Done only in case of immediate modification.)
 5. Scanning forward in the log, for all transactions starting from T_i or later with a < T_i commit>, execute $\text{redo}(T_i)$.
-

Example of Checkpoints



- T_1 can be ignored (updates already output to disk due to checkpoint)
- T_2 and T_3 redone.
- T_4 undone