Transaction Management

Dr. Xu Yang

Transaction Management

- **Transaction Support**
- **Concurrency Control**
- **Recovery Control**

Part I. Transaction Support

- Transaction Concept
- Transactions Properties (ACID)
- Transaction States

What is a Transaction

Transaction

Action, or series of actions, carried out by user or application, which accesses or changes contents of database.

A transaction is a sequence of operations that must be executed completely, taking a *consistent (& correct)* database state into another *consistent (& correct)* database state, although consistency may be violated during transaction.

Application program is series of transactions with non-database processing in between.

begin Transaction execution of Transaction end Transaction

Example Transaction

Update the salary of a particular member of staff given the staff number, x.

```
Read(staffNo=x, salary)
Salary=salary*1.1
Write(staffNo=x,salary)
```

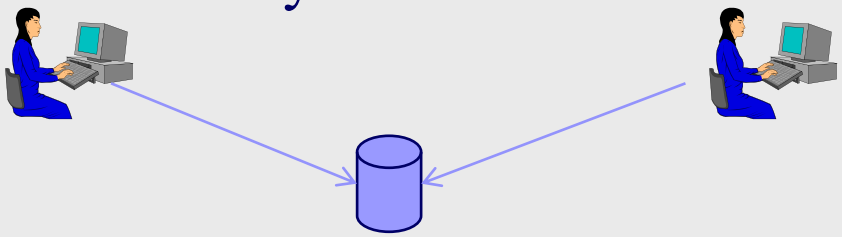
A transaction consisting of two database operations (read and write), and a non-database operation (salary=salary*1.1)

read(*X*), which transfers the data item *X* from the database to a variable also called *X*, in a buffer in main memory belonging to the transaction that executed the read operation.

write(X), which transfers the value in the variable X in the main-memory buffer of the transaction that executed the write to the data item X in the database.

Two main issues to deal with

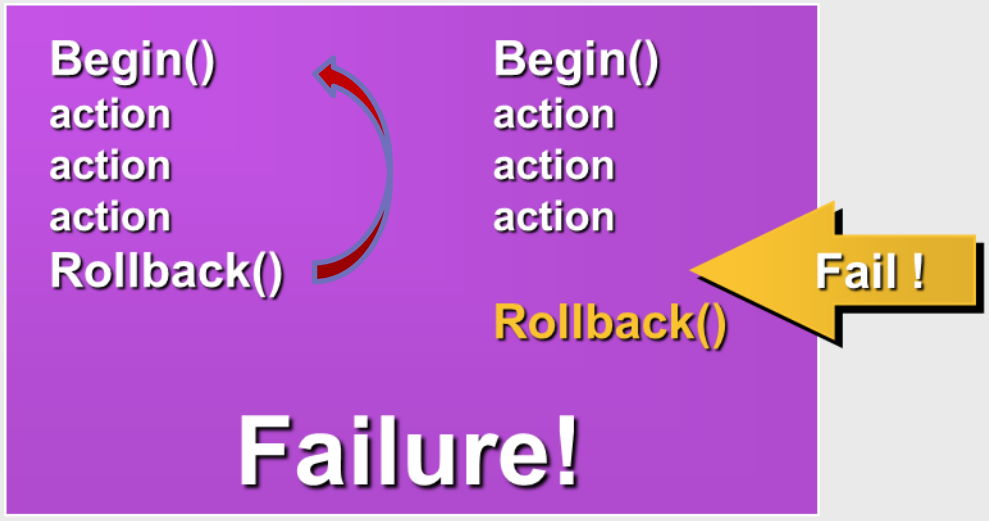
- **Concurrent execution of multiple transactions**
  - Most DBMS are multi-user systems.
  - The concurrent execution of transactions must be such that each transaction appears to execute in isolation.
  - Interleaved actions coming from different clients do not cause inconsistency in the data



- **Failures of various kinds, such as hardware failures and system crashes**

Transaction

- If the Transaction fails or aborts midway, then the Database is "rolled back" to its initial consistent state (when the Transaction began).



Transactions…

- The **COMMIT** operation signals successful end-of-transaction: It tells the transaction manager that
  - a logical unit of work has been successfully completed,
  - the database is (or should be) in a consistent state again,
  - and all of the updates made by that unit of work can now be "committed" or made permanent.
- The **ROLLBACK** operation signals unsuccessful end-of-transaction: It tells the transaction manager that
  - something has gone wrong,
  - the database might be in an inconsistent state,
  - and all of the updates made by the logical unit of work so far must be "rolled back" or undone.

8

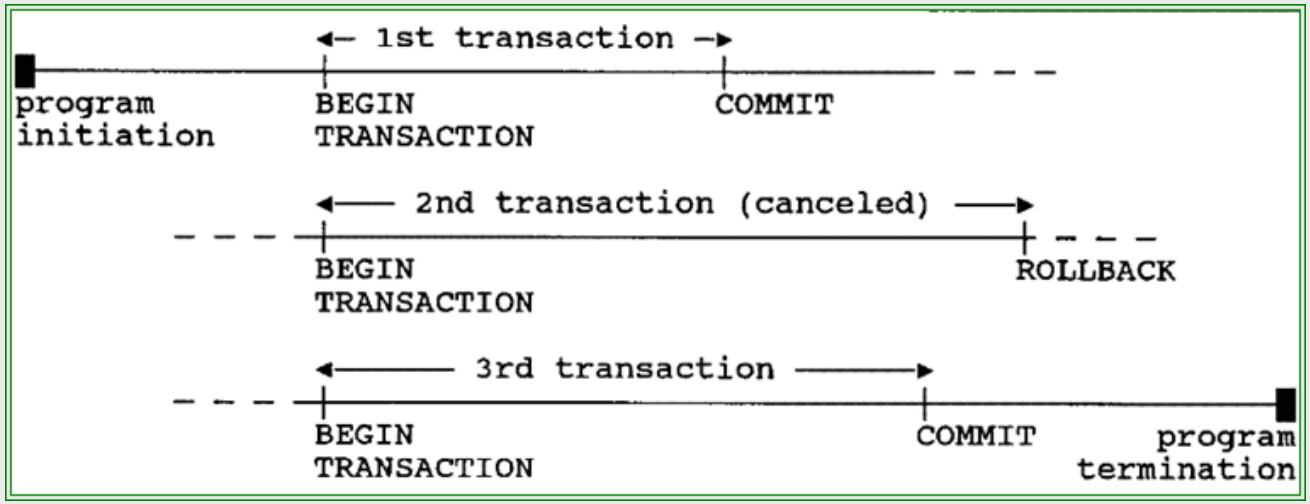
Transactions-commit point

- COMMIT establishes a commit point.
  - A commit point thus corresponds to the end of a logical unit of work, and hence to a point at which the database is or should be in a consistent state.
- ROLLBACK, by contrast, rolls the database back to the state it was in at BEGIN TRANSACTION, which effectively means back to the previous commit point.

9

Transactions-commit point

- When a commit point is established:
  - All updates made by the executing program since the previous commit point are committed; that is, they are made permanent.
  - All database positioning is lost and all tuple locks are released.



COMMIT and ROLLBACK terminate the transaction, not the program.

Two Outcomes of Transaction

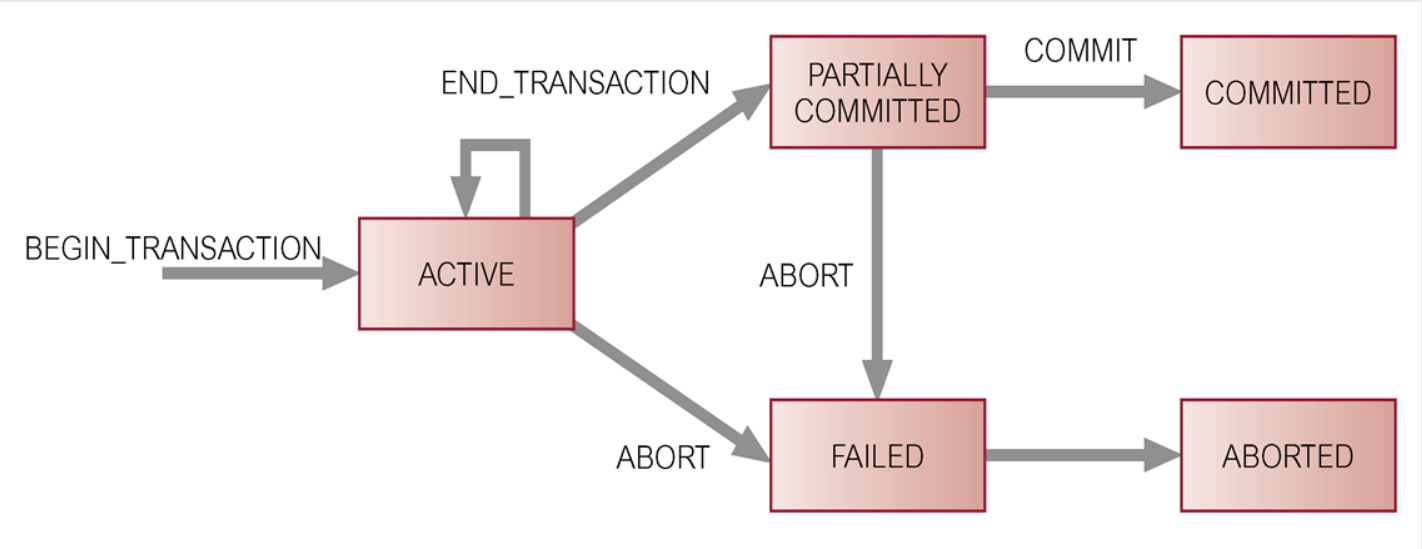
- **A transaction can have one of two outcomes: Success or Failure.**
  - **Success** - transaction *commits* and database reaches a new consistent state.
  - **Failure** - transaction *aborts*, and database must be restored to consistent state before it started. Such a transaction **is *rolled back*** or ***undone***.

Two Outcomes of Transaction

- **Committed transaction cannot be aborted.**
  - If we decide that the committed transaction was a mistake, we must perform another **compensating transaction** to reverse its effects.

- **Aborted transaction that is rolled back can be restarted later.**

Transaction States



- Active: the transaction is executing.
- Partially Committed: the transaction ends after execution of final statement.
- Committed: A transaction reaches its commit point when all operations accessing the database are completed..
- Failed: when the normal execution can no longer proceed.
- Aborted: after the transaction has been rolled back.

Properties of Transactions (ACID)

- **Atomicity**: all or nothing. Either all operations of the transaction are properly reflected in the database or none are.
- **Consistency:** state transformation. Must transform database from one consistent state to another.
- **Isolated:** no concurrency anomalies. Partial effects of incomplete transactions should not be visible to other transactions.
- **Durable:** committed transaction effects persist. Effects of a committed transaction are permanent and must not be lost because of later failure.

Why bother: atomicity

- insert record in file
  - Exactly once: keep trying
    'till acknowledged and server discards duplicate requests.

Transaction to transfer $50 from account A to account B:

1. **read**(A)
2. $A := A - 50$
3. **write**(A)
4. **read**(B)
5. $B := B + 50$
6. write(B)

Atomicity requirement
if the transaction fails after step 3 and before step 6, money will be "lost" leading to an inconsistent database state
Failure could be due to software or hardware

- Want ALL or NOTHING for group of actions

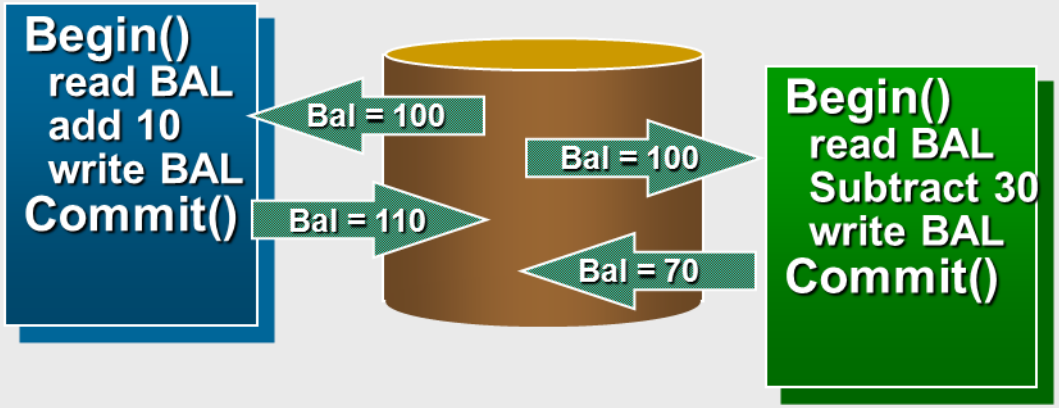
Why bother: consistency

- Begin-Commit brackets a set of operations
- You can violate consistency inside brackets
  - Debit but not credit (destroys money)
  - Delete old file before create new file in a copy

- Begin and commit are points of consistency

Why bother: isolation

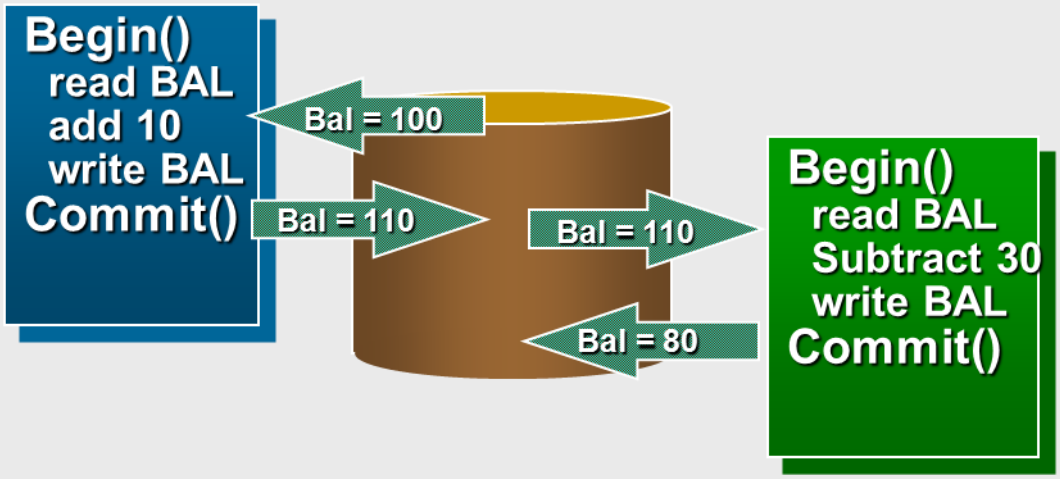
- Running programs concurrently on same data can create concurrency anomalies
  - The shared checking account example



  - if several transactions are executed concurrently, their operations may interleave in some undesirable way, resulting in an inconsistent state.

isolation

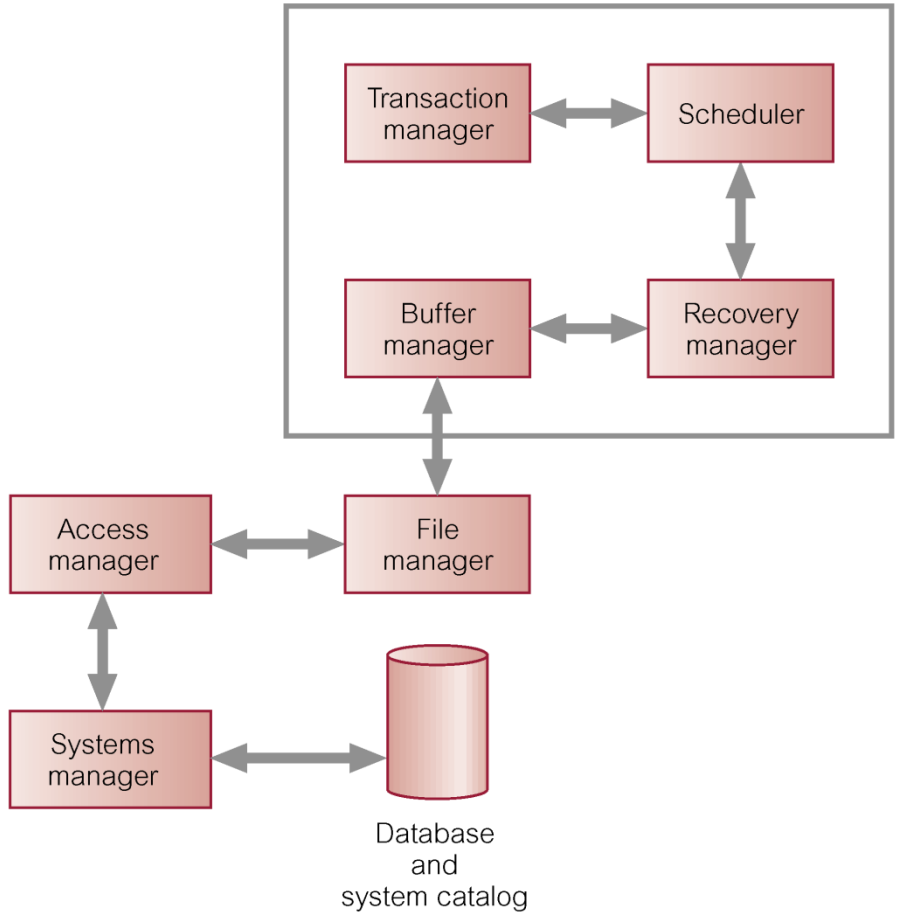
- It is as though programs run one at a time
  - No concurrency anomalies
- System automatically protects applications
  - Locking (DB2, Informix, Microsoft® SQL Server™, Sybase…)
  - Versioned databases (Oracle, Interbase…)



Why Bother: Durability

- Once a transaction commits, want **effects to survive failures**
- Fault tolerance
- Redo "lost" transactions in case of failure
- Resend unacknowledged messages

Transaction Execution



Transaction manager coordinates transactions on behalf of application programs.
Scheduler responsible for implementing a particular strategy for concurrency control.
Recovery manager is to ensure that the database is restored to the state it was in before the start of the transaction, and therefore a consistent state.
Buffer manager is responsible for the efficient transfer of data between disk storage and main memory.

Transaction Summary

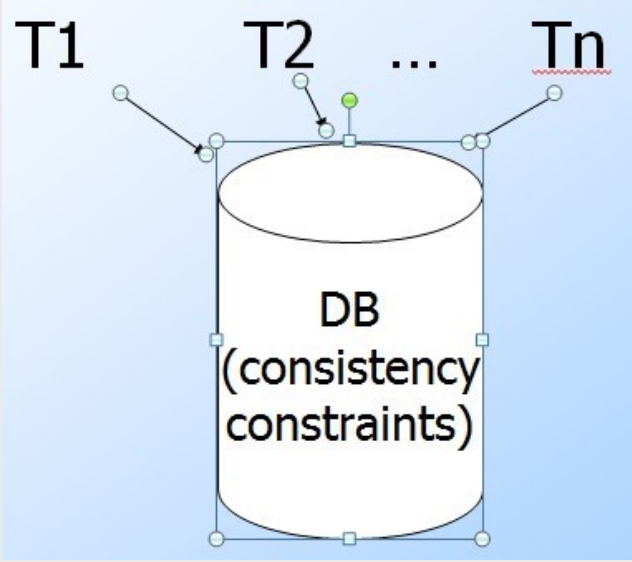
Transaction concept: read and write

Transaction properties (ACID): atomicity, consistency, isolated and durable

Commit and rollback

Part II. Concurrency control

- Serial Schedules and Serializability
- Locking and Two phase locking
- Deadlock and how it can be resolved
- Timestamping
- Optimistic Techniques



Concurrency : example

Suppose that the same program is executed concurrently by two applications aiming at reserving a seat in the same flight

The following temporal evolution is possible:

Application 1	Application 2
Find seat	
	Find seat
Book seat	
	Book seat

The result is that we have two reservations for the SAME seat.

Concurrency Control

Concurrency Control

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

- The DBMS deals with this problem by ensuring the socalled "isolation" property for the transactions

- This property for a transaction essentially means that it is executed like it was the only one in the system, i.e., without concurrent transactions.

Reason for Allowing Concurrency

Improved **throughput** of transactions and **system resource** utilization

Reduced waiting time of transaction.

Possible **Problems**

Three examples of potential problems caused by concurrency:

- Lost update problem
- Temporary update problem
- Incorrect summary problem

Lost update problem

Successfully completed update is overridden by another user.

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

T_1 withdrawing £10 from an account with bal_x, initially £100. T_2 depositing £100 into same account. Serially, final balance would be £190.

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

Uncommitted Dependency Problem （dirty read)

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

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

T_4 updates bal_x to £200 but it aborts, so bal_x should be back at original value of £100. T_3 has read new value of bal_x (£200) and uses value as basis of £10 reduction, giving a new balance of £190, instead of £90.

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

Inconsistency Analysis Problem

Occurs when transaction reads several values but second transaction updates some of them during execution of first. (dirty read or unrepeatable read)

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

T_6 is totaling balances of account x (£100), account y (£50), and account z (£25). Meantime, T_5 has transferred £10 from bal_x to bal_z, so T_6 now has wrong result (£10 too high).

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

Scheduling Transactions

A Schedule is a sequential order of the instructions (R / W / A / C) of *n* transactions such that the ordering of the instructions of each transaction is preserved.(execution sequence preserving the operation order of individual transaction)

- A transaction that successfully completes its execution will have a **commit** instructions as the last statement.
  - by default transaction assumed to execute commit instruction as its last step
- A transaction that fails to successfully complete its execution will have an **abort** instruction as the last statement.

Serial Schedule

Serial schedule: A schedule that does not interleave the actions of different transactions. (transactions executed consecutively)

T_7	T_8
begin_transaction	
read(bal_x)	
write(bal_x)	
read(bal_y)	
write(bal_y)	
commit	
	begin_transaction
	read(bal_x)
	write(bal_x)
	read(bal_y)
	write(bal_y)
	commit

A *serial* schedule in which T_7 is followed by T_8

Non-serial Schedules

Non-Serial schedule: A schedule where the operations from a set of concurrent transactions are interleaved.

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

A *non-serial* schedule in which T_7 and T_8 are interleaved.

Serializability

A schedule S is ***serializable*** if the outcome of its execution is the same as the outcome of at least one serial schedule constituted by the same transactions of S.

- In other words, a schedule S on T1,T2,…,Tn is serializable if there exists a serial schedule on T1,T2,…,Tn that is "equivalent" to S

What does "equivalent" mean?

- Two schedules S1 and S2 are equivalent if, for each initial state and for each database, the execution of S1 produces the same outcome on the database as the execution of S2

A serial schedule

T_1	T_2	A	B
		25	25
begin			
READ(A,t)			
t := t+100			
WRITE(A,t)		125	
READ(B,t)			
t := t+100			
WRITE(B,t)			125
commit			
	begin		
	READ(A,s)		
	s := s*2		
	WRITE(A,s)	250	
	READ(B,s)		
	s := s*2		
	WRITE(B,s)		250
	commit		

A serializable schedule

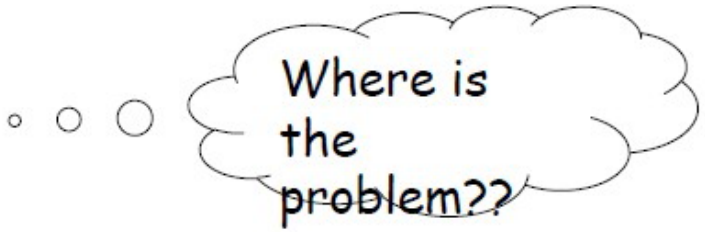
T_1	T_2	A	B
		25	25
begin			
	begin		
READ(A,t)			
t := t+100			
WRITE(A,t)		125	
	READ(A,s)		
	s := s*2		
	WRITE(A,s)	250	
READ(B,t)			
t := t+100			
WRITE(B,t)			125
commit			
	READ(B,s)		
	s := s*2		
	WRITE(B,s)		250
	commit		

The final values of A and B are the same as the serial schedule T1, T2, no matter what the initial values of A and B.

We can indeed show that, if initially A=B=c (c is a costant), then at the end of the execution of the schedule we have: A=B=2(c+100)

A non-serializable schedule

T_1	T_2	A	B
		25	25
begin	begin		
READ(A,t)			
t := t+100			
WRITE(A,t)		125	
	READ(A,s)		
	s := s*2		
	WRITE(A,s)	250	
	READ(B,s)		
	s := s*2		
	WRITE(B,s)		50
	commit		
READ(B,t)			
t := t+100			
WRITE(B,t)			150
commit			



Objective of Serializability

- Objective of a concurrency control protocol is to schedule transactions in such a way as to avoid any interference.
- Could run transactions serially, but this limits degree of concurrency or parallelism in system.
- Serializability identifies those executions of transactions guaranteed to ensure consistency.
- Objective of serializability is to find nonserial schedules that allow transactions to execute concurrently without interfering with one another.

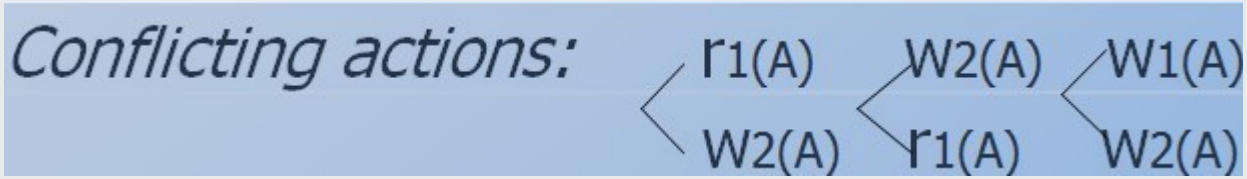
Read-Write Conflict Rules

- **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.



Two actions are conflicting in a schedule if they belong to different transactions, they operate on the same element, and one of them is a write.

Read-Write Conflict Rules

- **Two consecutive non-conflicting actions belonging to different transactions can be swapped without changing the effects of the schedule.** Indeed,
  - Two consecutive reads of the same elements in different transactions can be swapped
  - One read of X in T1 and a consecutive read of Y in T2 (with Y≠X) can be swapped
- **The swap of two consecutive actions of the same transaction can change the effect of the transaction**
- **Two conflicting consecutive actions cannot be swapped, because:**
  - Swapping two write operations w1(A) w2(A) on the same elements may result in a different final value for A
  - Swapping two consecutive operations such as r1(A) w2(A) may cause T1 read different values of A (before and after the write of T2, respectively)

Conflict Serializability

- S_1, S_2 are ***conflict equivalent*** schedules
  - if S_1 can be transformed into S_2 by a series of swaps on non-conflicting actions.
- A schedule is ***conflict serializable*** if it is conflict equivalent to some serial schedule.

- ***Constrained Write Rule***

(transaction updates data item based on its old value, which is first read by the transaction)

Conflict-equivalence Example

S = r1(A) w1(A) r2(A) w2(A) r1(B) w1(B) r2(B) w2(B) is **conflict-equivalent** to:

S' = r1(A) w1(A) r1(B) w1(B) r2(A) w2(A) r2(B) w2(B)

because it can be transformed into S' through the following sequence of swaps:

r1(A) w1(A) r2(A) w2(A) r1(B) w1(B) r2(B) w2(B)

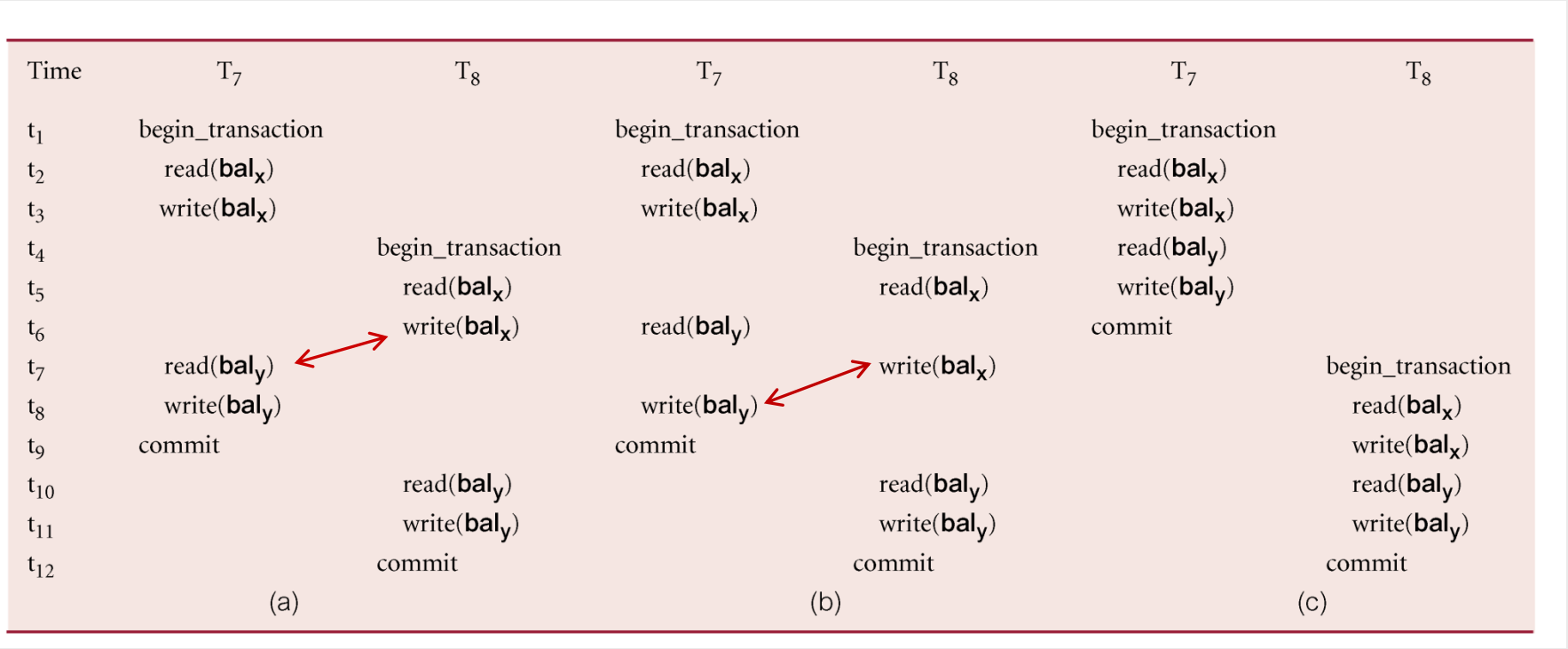
r1(A) w1(A) r2(A) r1(B) w2(A) w1(B) r2(B) w2(B)

r1(A) w1(A) r1(B) r2(A) w2(A) w1(B) r2(B) w2(B)

r1(A) w1(A) r1(B) r2(A) w1(B) w2(A) r2(B) w2(B)

r1(A) w1(A) r1(B) w1(B) r2(A) w2(A) r2(B) w2(B)

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)	

Schedule “a” can be transformed into Schedule “c”, a serial schedule where T_8 follows T_7, by series of swaps of non-conflicting instructions. Therefore Schedule a is conflict serializable.

Non-Conflict Serializable Schedule

- Example of a schedule that is not conflict serializable:

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

- We are unable to swap instructions in the above schedule to obtain either the serial schedule < T_3, T_4 >, or the serial schedule < T_4, T_3 >.

Precedence (Dependency) Graph

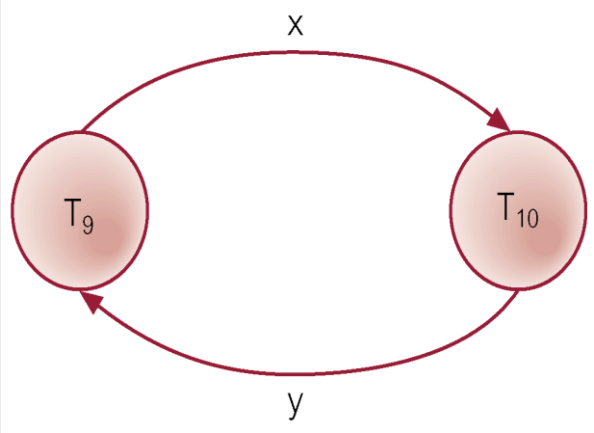
- **Theorem: Schedule is conflict serializable if and only if its dependency graph is *Dependency graph*:**
  - node for each transaction;
  - a directed edge $T_i \rightarrow T_j$, if T_j reads the value of an item written by T_I;
  - a directed edge $T_i \rightarrow T_j$, if T_j writes a value into an item after it has been read by T_i.
  - a directed edge $T_i \rightarrow T_j$, if T_j writes a value into an item after it has been written by T_i.

Example : Non-conflict serializable schedule

- T_9 is transferring £100 from one account with balance bal_x to another account with balance bal_y.
- T_{10} is increasing balance of these two accounts by 10%.

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	

Non Conflict Serializable



The cycle in the graph reveals the problem. The output of T9 depends on T10, and vice-versa.

Examples

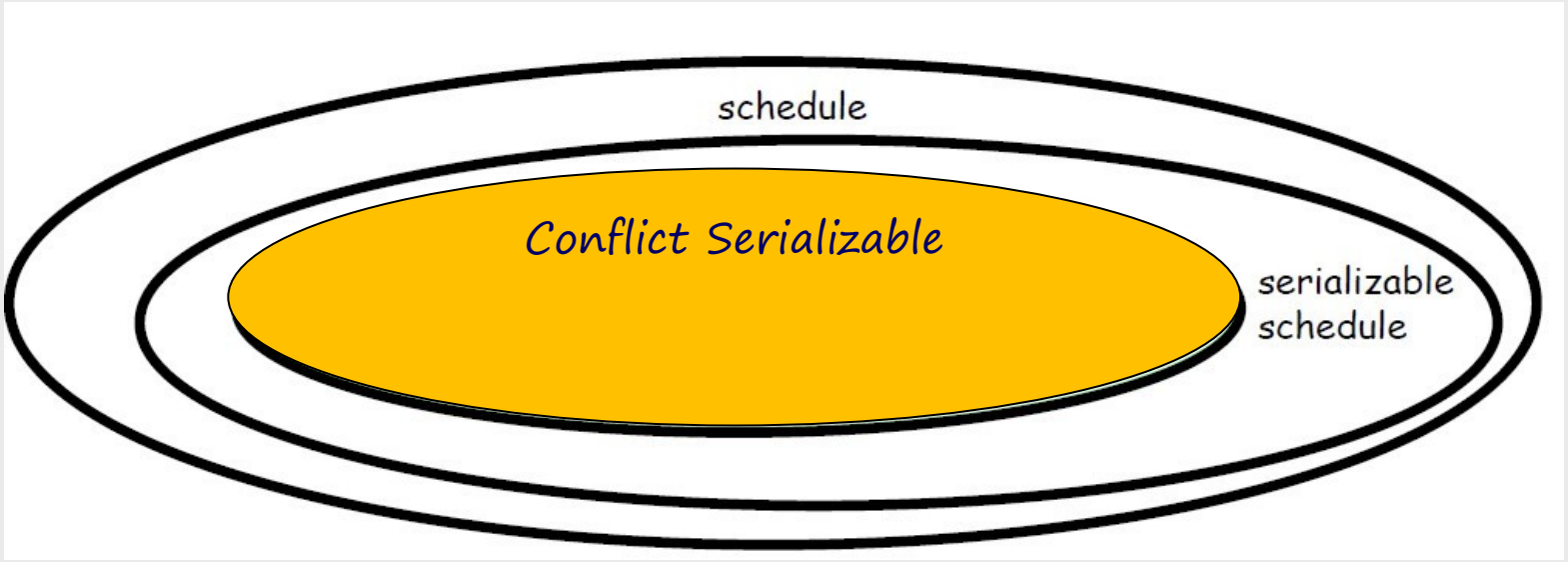
- Two schedules S1 and S2

  S1 = w1(A) r2(A) w2(B) r1(B)

  S2 = r2(A) w1(A) r1(B) w2(B)

have the same precedence graph, but they are not conflict equivalent, since to transform one of them into the other requires swapping two conflicting actions.

Relationship



Concurrency Control Techniques

- **Two basic concurrency control techniques:**
  - **Locking,** 
  - **Timestamping.** 
  - **Optimistic**
    - **Locking and timestamping** are conservative approaches: delay transactions in case they conflict with other transactions.
    - **Optimistic** methods are based on the premise that conflict is rare so they allow transactions to proceed unsynchronized and only check for conflicts at the end, when a transaction commits.
- conflict-serializability are not used in commercial systems

Locking Based Concurrency Control

Locking

A procedure used to control concurrent access to data. When one transaction is accessing the database, a lock may deny access to other transactions to prevent incorrect results.

- Lock requests are made to concurrency-control manager. Transaction can proceed only after request is granted.
- **Most widely used approach to ensure serializability.**

Can we lock the entire database ?

Two Modes of Locking

- If transaction has **shared lock** on item, can read but not update item.
  - Reads cannot conflict, so more than one transaction can hold shared locks simultaneously on same item.
- If transaction has **exclusive lock** on item, can both read and update item.
  - As long as a transaction holds the exclusive lock on the item, no other transactions can read or update that data item.
- Some systems allow shared locks to be upgraded to exclusive locks
- Similarly, sometimes exclusive locks can be downgraded to shared locks

50

Locking - Basic Rules

- Any transaction that needs to access a data item must first lock the item
  - For read only access, requesting a shared lock
  - For both read and write access, requesting an exclusive lock.

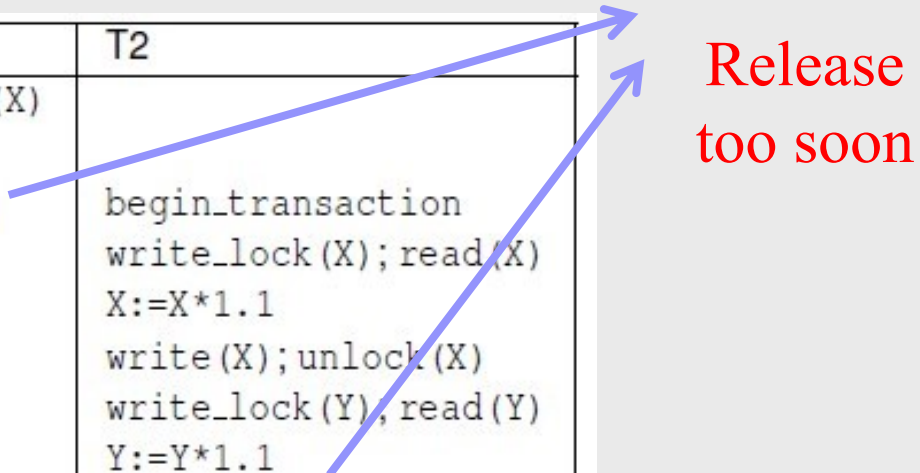
For one object		*Lock requested*	
		read	*write*
Lock already set	*none*	OK	OK
	read	OK	wait
	write	wait	wait

- When a transaction is committed or aborted, the server unlocks all objects it locked for the transaction.

Example – Incorrect Locking Schedule

- If X=100 and Y=400, if T1 < T2 then X=220, Y=330, and if T2 < T1 then X=210, Y=340, but the schedule above gives: X=220 and Y=340

Time	T1	T2
$t_{1,2}$	write_lock(X); read(X)	
t_3	X:=X+100	
$t_{4,5}$	write(X); unlock(X)	begin_transaction
$t_{6,7}$		write_lock(X); read(X)
t_8		X:=X*1.1
$t_{9,10}$		write(X); unlock(X)
$t_{11,12}$		write_lock(Y); read(Y)
t_{13}		Y:=Y*1.1
$t_{14,15}$		write(Y); unlock(Y)
$t_{16,17}$	write_lock(Y); read(Y)	commit
t_{18}	Y:=Y-100	
$t_{19,20}$	write(Y); unlock(Y)	
t_{21}	commit	



Example shows that locking does not always guarantee serializability – lost update

Incorrect Locking Schedule

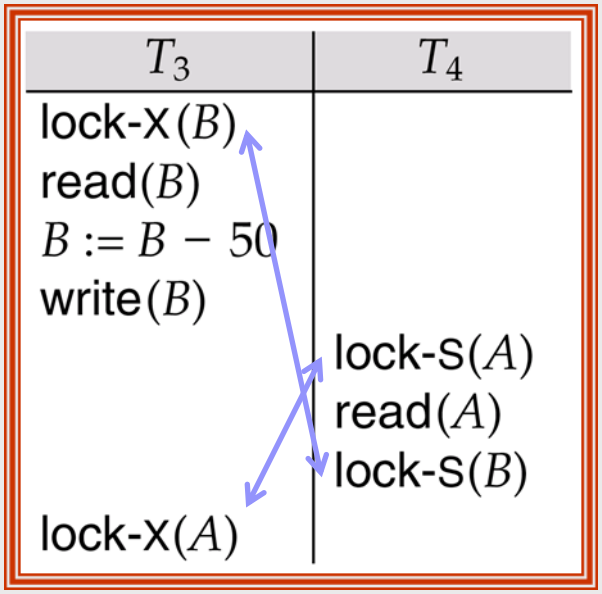
- Problem is that transactions release locks too soon, resulting in loss of total isolation and atomicity.

- To guarantee serializability, need an additional protocol concerning the positioning of lock and unlock operations in every transaction.

Dead Lock

Consider the partial schedule

Neither T_3 nor T_4 can make progress — executing **lock-S**(B) causes T_4 to wait for T_3 to release its lock on B, while executing **lock-X**(A) causes T_3 to wait for T_4 to release its lock on A.



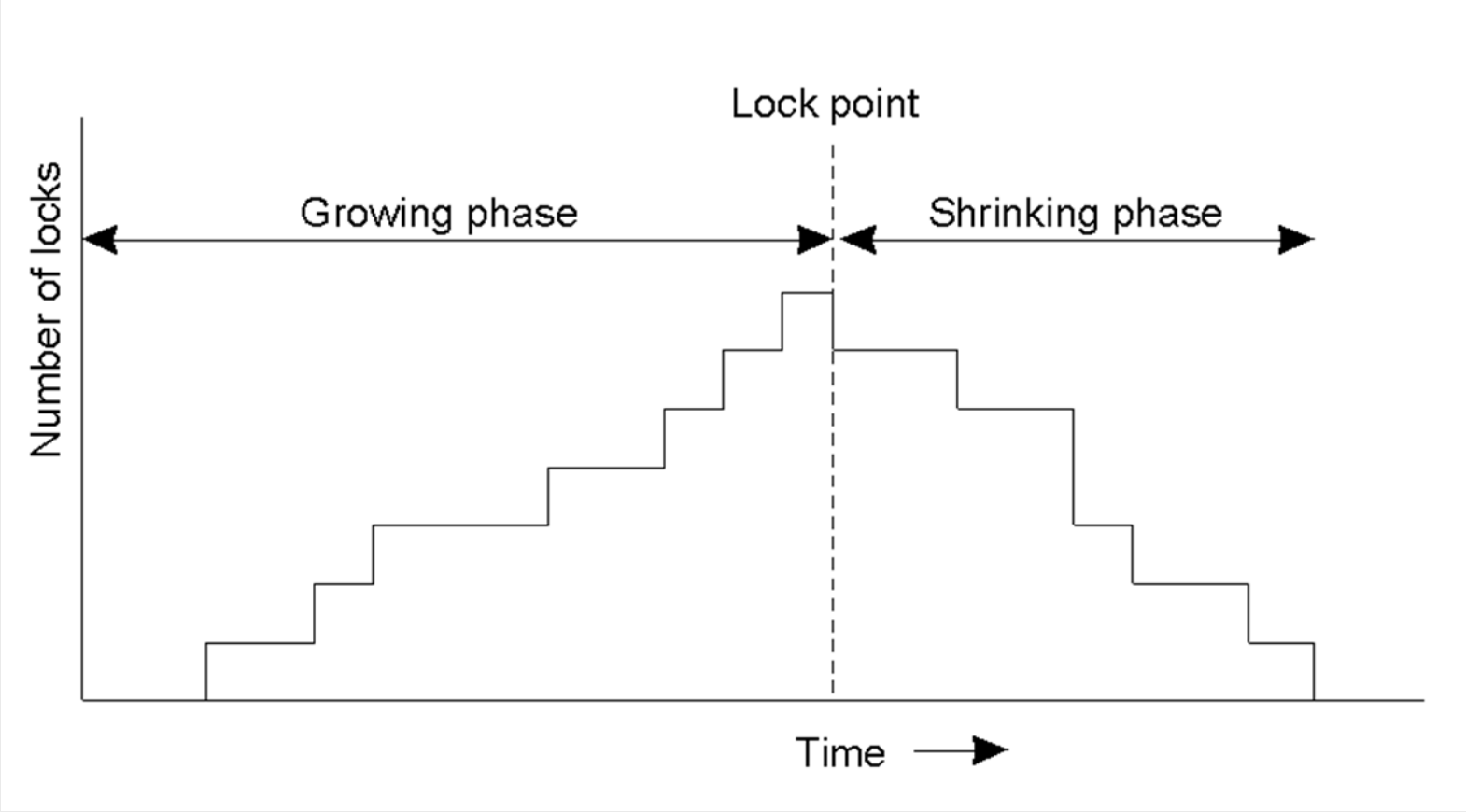
Such a situation is called a **deadlock.**

To handle a deadlock one of T_3 or T_4 must be **rolled back and its locks released**.

Two Phase Locking (2PL)

- Two-phase locking with lock conversions:
  - First Phase: (
    - can acquire a lock-S on item
    - can acquire a lock-X on item
    - can convert a lock-S to a lock-X (upgrade)
  - Second Phase:
    - can release a lock-S
    - can release a lock-X
    - can convert a lock-X to a lock-S (downgrade)
- **This protocol assures serializability.**
- Two-phase locking does *not* ensure freedom from deadlock.

Two Phase Locking



In two-phase locking, a transaction is not allowed to acquire any new locks after it has released a lock

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

To prevent the lost update problem occurring, T2 first requests an exclusive lock on balx. When T1 starts, it also requests an exclusive lock on balx. However, because the data item balx is currently exclusively locked by T2, the request is not immediately granted and T1 has to **wait** until the lock is released by T2. This occurs only once the commit of T2 has been completed.

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

To prevent uncommitted dependency problem occurs, T4 first requests an exclusive lock on balx. When the rollback is executed, the updates of transaction T4 are undone and the value of balx in the database is returned to its original value of £100. When T3 starts, it also requests an exclusive lock on balx. However, because the data item balx is currently exclusively locked by T4, the request is not immediately granted and T3 has to wait until the lock is released by T4. This occurs only once the rollback of T4 has been completed.

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

T5 must precede its reads by exclusive locks, and T6 must precede its reads with shared locks. Therefore, when T5 starts it requests and obtains an exclusive lock on balx. Now, when T6 tries to share lock balx the request is not immediately granted and T6 has to wait until the lock is released, which is when T5 commits.

Example of Cascading Rollback

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

a single transaction leads to a series of rollbacks, is called **cascading rollback.**

Cascading Rollback

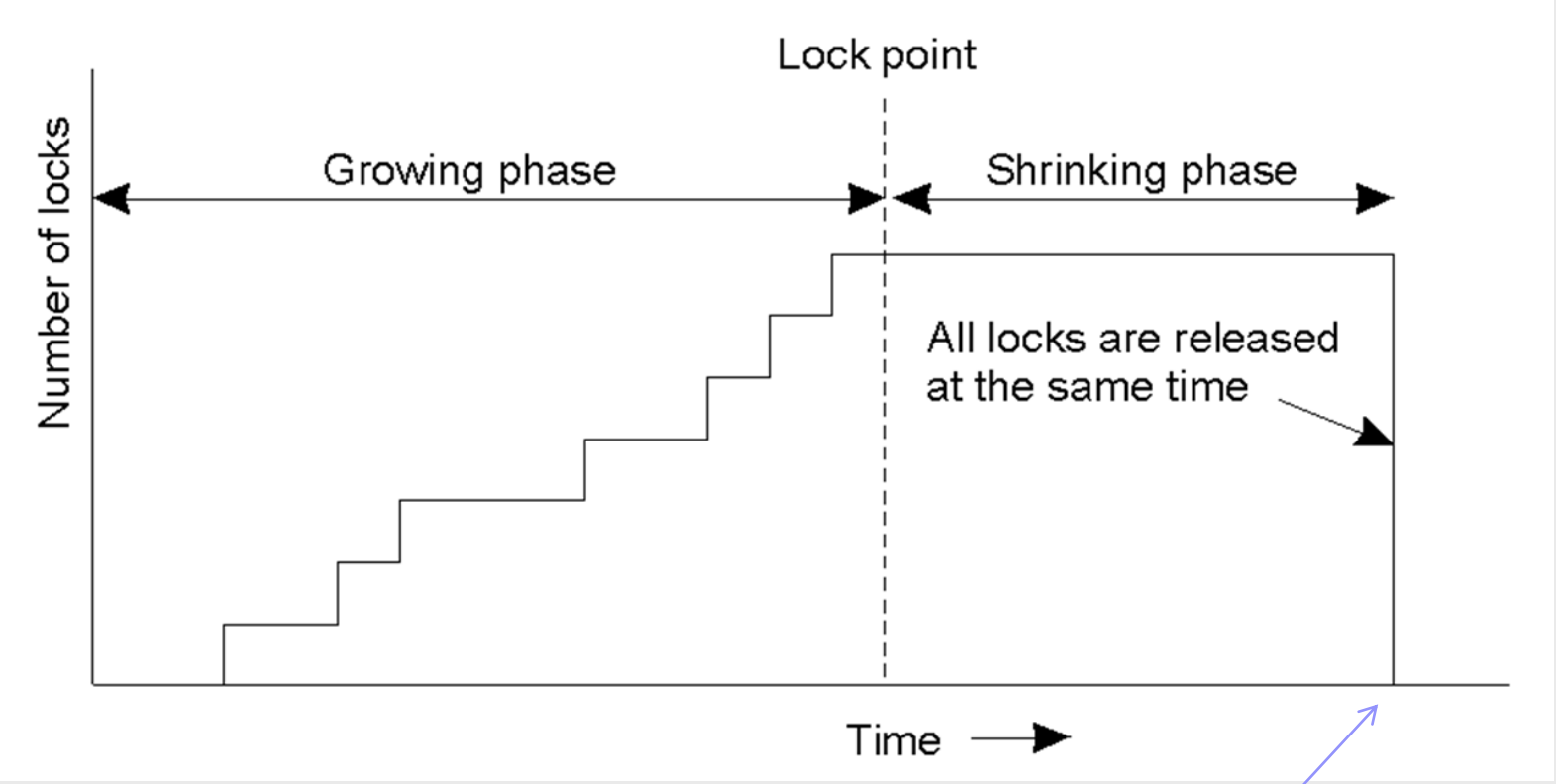
- **If *every* transaction in a schedule follows 2PL, schedule is serializable**.
- However, problems can occur with interpretation of when locks can be released.
  - Transactions conform to 2PL.
  - T_{14} aborts.
  - Since T_{15} is dependent on T_{14}, T_{15} must also be rolled back. Since T_{16} is dependent on T_{15}, it too must be rolled back.
  - This is called ***cascading rollback*.**

Cascading rollbacks are undesirable since they potentially lead to the undoing of a significant amount of work.

To Avoid Cascading Roll-back

- Cascading roll-back is possible under two-phase locking.
  - To avoid this, follow a modified protocol called **strict two-phase locking**. Here a transaction must hold all its exclusive locks till it commits/aborts.
- **Rigorous two-phase locking** is even stricter: here *all* locks are held till commit/abort. In this protocol transactions can be serialized in the order in which they commit.

Rigorous two-phase locking



Strict two-phase locking: all exclusive locks are released at same time (transaction commit or abort)

Deadlock Management

- **Deadlock** occurs when two transactions T1 and T2 have the use of two elements A and B, and each of them asks for an exclusive lock on the element of the other one, and therefore no one can proceed.

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}	⋮	⋮

How to resolve deadlock problem?

Handling Deadlock in Two Phase Locking

- **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.

Dealing with Deadlock in two-phase locking

- **Timeouts**
  - Aborting transactions when time expires
- **Deadlock prevention**
  - Assign priorities based on timestamps
- **Deadlock detection**
  - Keep graph of locks held. Check for cycles periodically or each time an edge is added
  - Cycles can be eliminated by aborting transactions

Lock 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.

Resolution of Deadlock

Transaction T		Transaction U	
Operations	**Locks**	**Operations**	**Locks**
a.deposit(100);	write lock *A*		
		b.deposit(200)	write lock *B*
b.withdraw(100)			
•••	waits for *U*'s lock on *B*	*a.withdraw(200);*	waits for T's lock on *A*
	(timeout elapses)	•••	
T's lock on *A* becomes vulnerable, unlock *A*, abort T		•••	
		a.withdraw(200);	write locks *A*
			unlock *A*, *B*

Timestamps

- **To order transactions using transaction timestamps. Two algorithms:**
  - **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*).

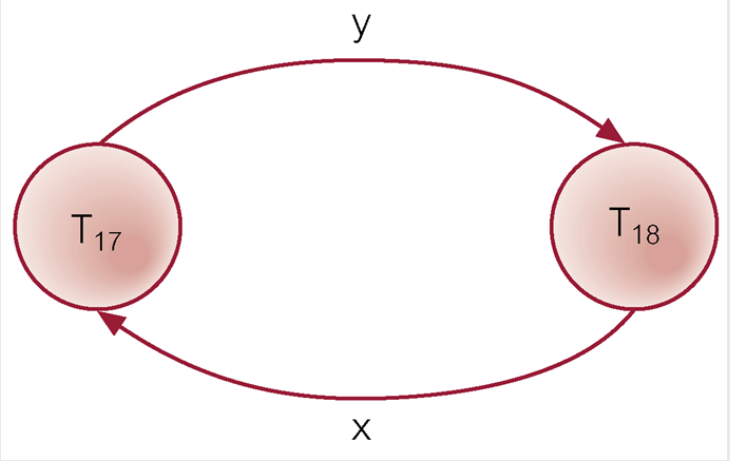
Example of Wait-Die and Would-Wait

- Three transactions T1, T2 and T3 have timestamps 5, 10, 15 (T1 is oldest, and T3 is youngest).
  - a) T1 request a data locked by T2, and
  - b) T3 request a data locked by T2.
- Wait-die:
  - a)T1 is older than T2, so T1 can wait.
  - b) T3 is younger than T2, so T3 can not wait, so T3 (younger one) roll back.
- Wound-wait:
  - a) T1 is older than T2, so T1 can not wait, then T2 (younger one) roll back.
  - b) T3 is younger than T2, so T3 can wait.

Deadlock Detection and Recovery

- **DBMS allows deadlock to occur but recognizes it and breaks it.**
- **Deadlock detection** is usually handled by construction of **wait-for graph (WFG)** showing transaction dependencies:
  - Create a node for each transaction.
  - Create edge T_i -> T_j, if T_i waiting to lock item locked by T_j.

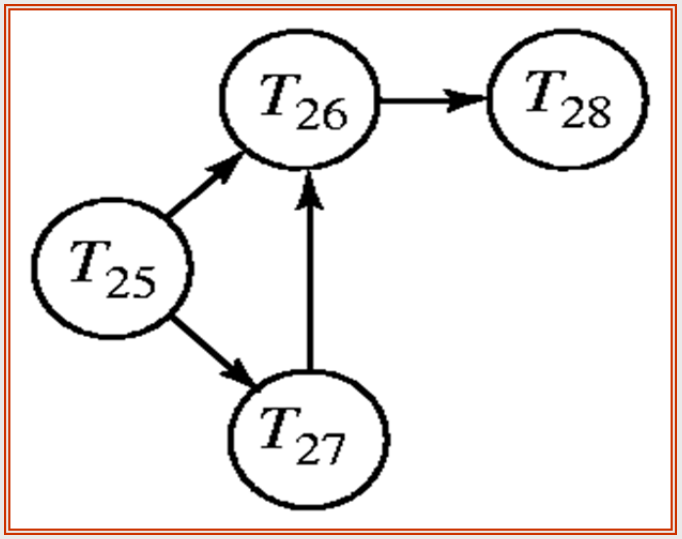
Deadlock exists if and only if WFG contains cycle.
WFG is created at regular intervals.



Example - Wait-For-Graph (WFG)

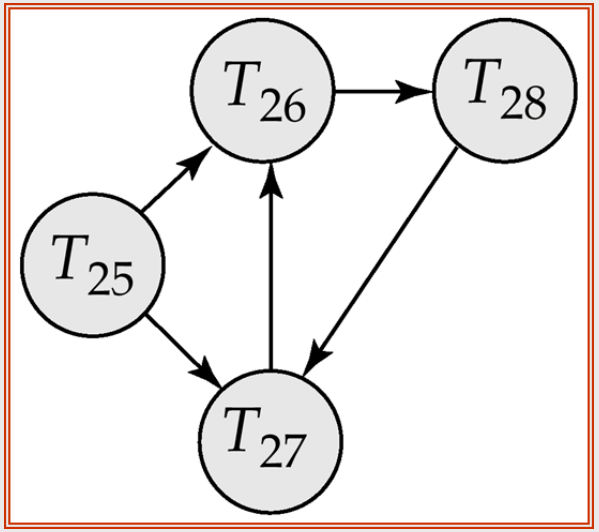
Transaction T25 is waiting for transactions T26and T27.

- Transaction T27 is waiting for transaction T26.
- Transaction T26 is waiting for transaction T28.



Wait-for graph without a cycle

If transaction T28 is waiting for transactions T27



Wait-for graph with a cycle

Recovery from Deadlock Detection

- **Choice of deadlock victim:** ⟹ ***any problem?***
  - abort transactions that incur the minimum costs
    - » – How long the transaction has been running?
    - » – How many data objects have been updated?
    - » – How many data objects the transaction is still to update?
- **How far to roll a transaction back;**
  - Total rollback: Abort the transaction and then restart it.
  - More effective to roll back transaction only as far as necessary to break deadlock.
- **Avoiding starvation.**
  - Starvation occurs when the same transaction is always chosen as the victim, and the transaction can never complete.

Concurrency based on timestamps

Timestamping

A concurrency control protocol that orders transactions in such a way that older transactions, transactions with smaller timestamps, get priority in the event of conflict.

- The protocol manages concurrent execution such that the time-stamps determine the serializability order.
- Conflict operations are resolved by timestamp ordering.
- No locks so no deadlock.
- Timestamps(though not widely utilized in real DBMSs, they have a theoretical interest).

Optimistic Techniques

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

- At commit, check is made to determine whether conflict has occurred.
- If there is a conflict, transaction must be rolled back and restarted.
- Potentially allows greater concurrency than traditional protocols.
- **Three Phases: Read, Validation, and Write**

Concurrency Control Summary

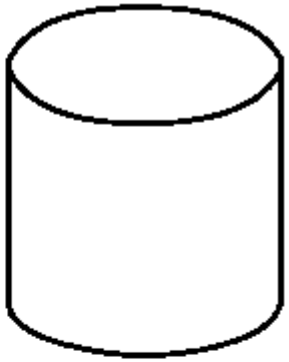
- Serial Schedules , Serializability, real-write conflict rules, Conflict Serializability, Precedence graph, recoverability.
- Locking, two phase locking
- Dead lock, wait for graph, cascading rollback
- Wait-Die and Would-Wait
- Optimistic Techniques

Part III. Database Recovery

- Database failure.
- Purpose of transaction log file.
- Purpose of checkpointing.
- How to recover following database failure.

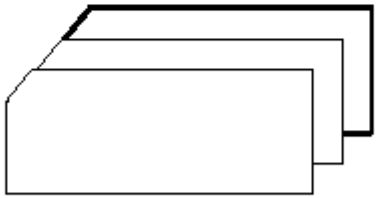
Potential problems

HARDWARE

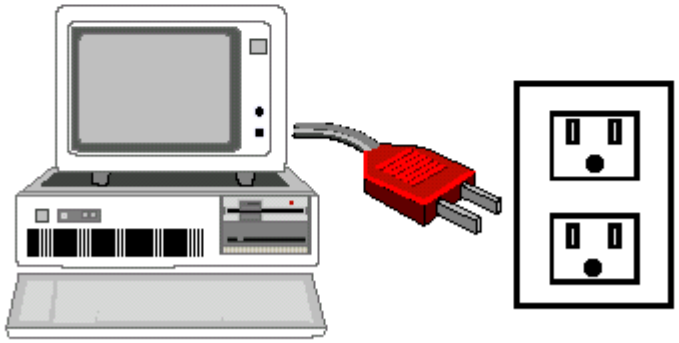


MEDIA

OPERATIONAL



SOFTWARE



POWER

Database Recovery

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

- **Example:**
  - If the system crashes before a fund transfer transaction completes its execution, then either one or both accounts may have incorrect value.
  - Thus, the database must be restored to the state before the transaction modified any of the accounts.

Types of Failures

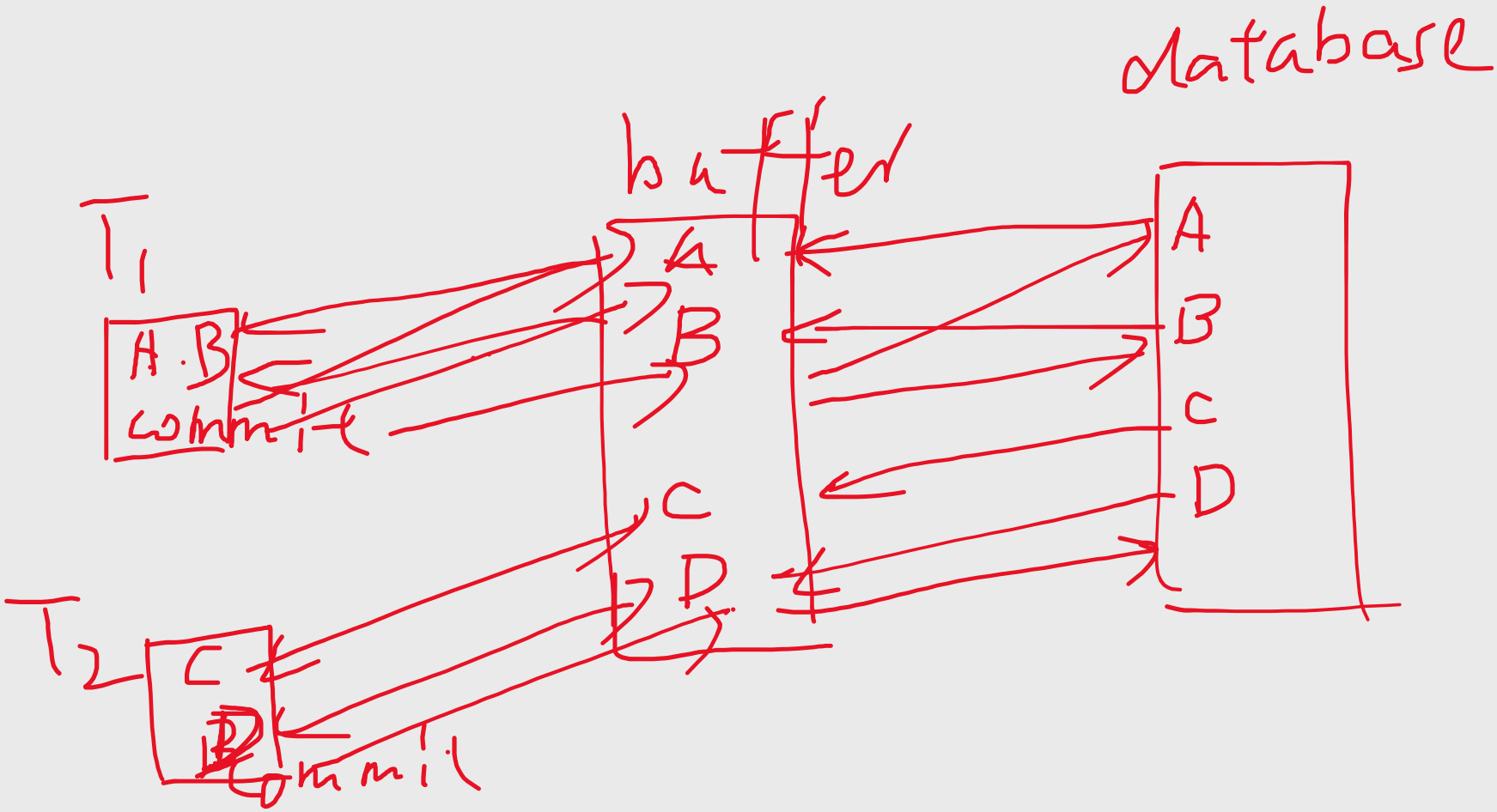
- Transaction failures: Transaction fails (such as logic errors or due to deadlock) for any reason to reach its planned termination.
- System failures: (e.g., power outage) Failure of processor, main memory… Affect all transactions currently in progress but do not physically damage the database.
- Media failures: Cause damage to the database (secondary storage), or to some portion of it, and affect at least those transactions currently using that portion.

Storage

- Two types of storage: **volatile (main memory, primary storage) and nonvolatile (secondary storage)**.
  - **Volatile storage**, generally too small to save the entire database, and does not survive system crashes.
  - Magnetic disks provide **online non-volatile** storage. Compared with main memory, disks are more reliable and much cheaper, but slower . The primary medium for the long term online storage of data is the magnetic disk.
  - Magnetic tape is an **offline non-volatile** storage medium, which is far more reliable than disk and fairly inexpensive, but slower, providing only sequential access. Tape storage is used primarily for backup and archival data.

The Buffer

- The **buffer** (also called buffer pool) is a non-persistent memory space used by the DBMS to transfer data from the secondary storage and back to the main memory.
  - Occupy an area in main memory from which data is transferred to and from secondary storage.
  - It is only once the buffers have been flushed to secondary storage that any update operations can be regarded as permanent.
- This memory area is
  - Shared by all transactions
  - Used by the system (e.g. by the recovery manager)

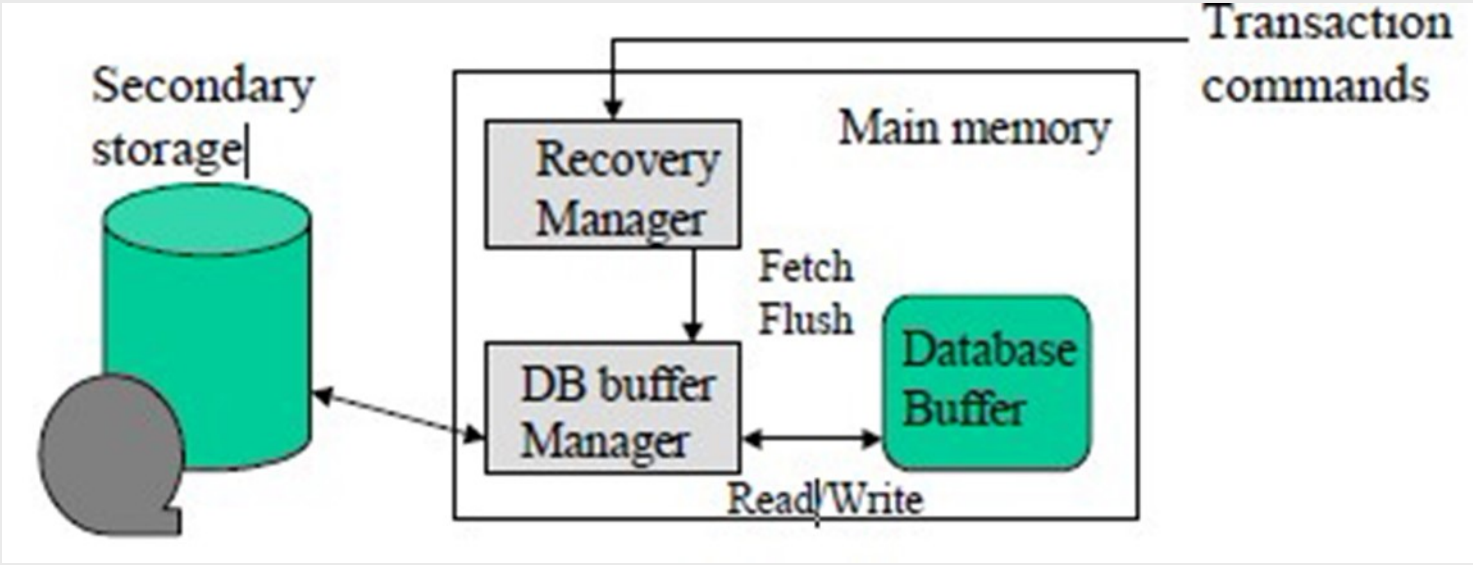

Database
buffer
T1
A B
commit
T2
C
D
commit
A
B
C
D
A
B
C
D

The Buffer

This flushing of the buffers to the database can be triggered

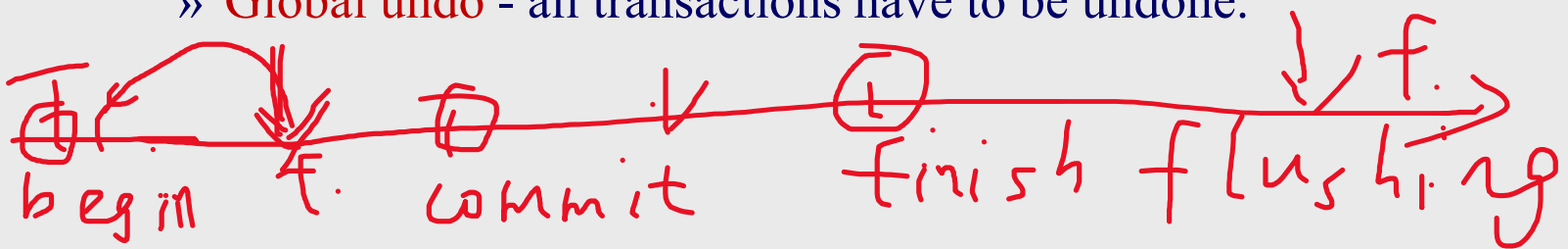
- by a specific command (for example, transaction **commit**)
- or automatically when the buffers become full.

The explicit writing of the buffers to secondary storage is known as **force-writing**.

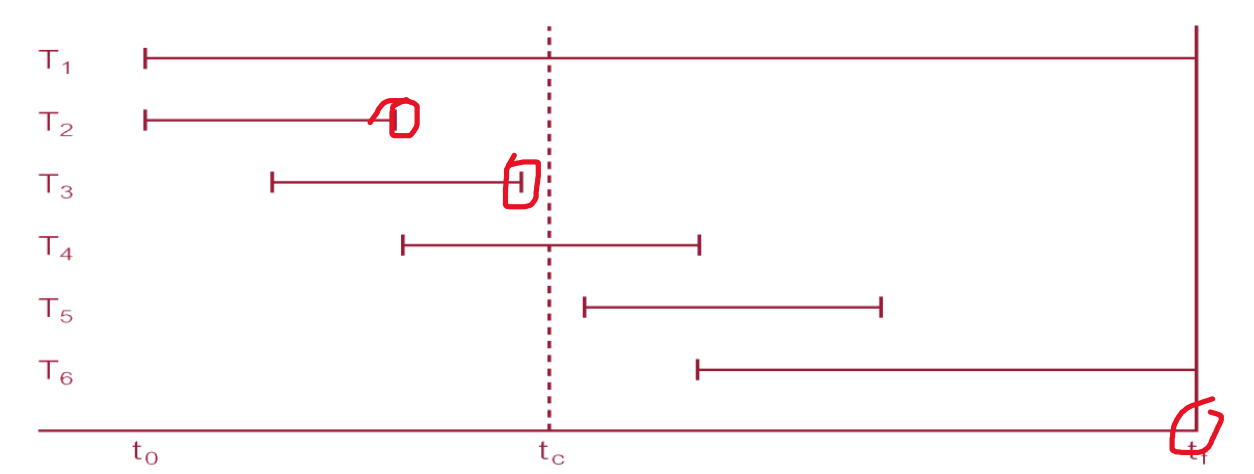


Transaction Recovery

- **Recovery manager responsible for atomicity and durability.**
  - Transactions represent basic unit of recovery
  - If failure occurs between commit and database buffers being flushed to secondary storage then, to ensure durability, recovery manager has to ***redo*** (*rollforward*) transaction's updates.
  - If transaction had **not committed** at failure time, recovery manager has to ***undo*** (*rollback*) any effects of that transaction for atomicity.
    - » Partial undo - only one transaction has to be undone.
    - » Global undo - all transactions have to be undone.



Example



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

Undo or Redo operations are **recorded in the log** as they happen.

Recovery Facilities

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

Transaction Log File

- To keep track of database transactions, DBMS maintains a special file called **a log** that contains information about all updates to database:
  - **Transaction records** (begin, insert, delete, update, commit, abort)**.**
  - **Checkpoint records.**

  - The log is a sequential file, recording all the update activities in the database.
  - Can be used for other purposes (for example, performance monitoring auditing).

Log File

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

Sample Log File

- **A segment of a log file that shows three concurrently executing transactions T1, T2, and T3.**

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

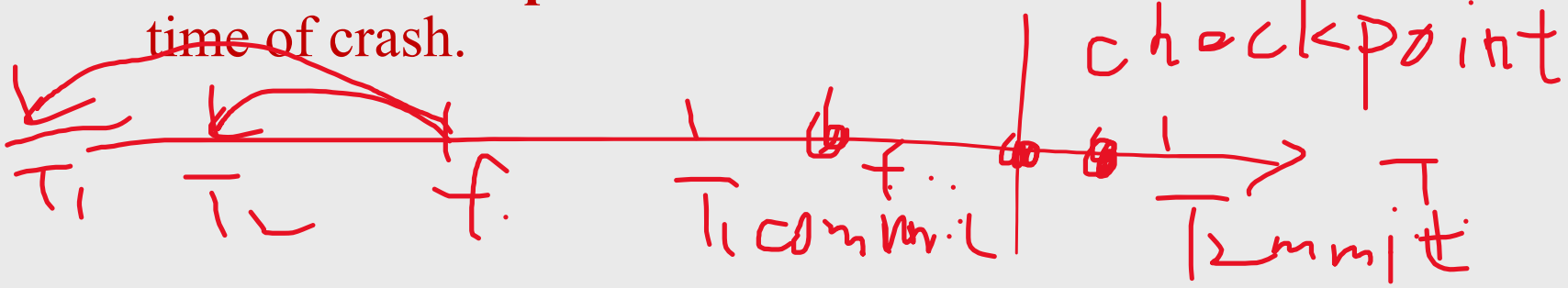
Potential bottleneck; critical in determining overall performance.

Checkpointing

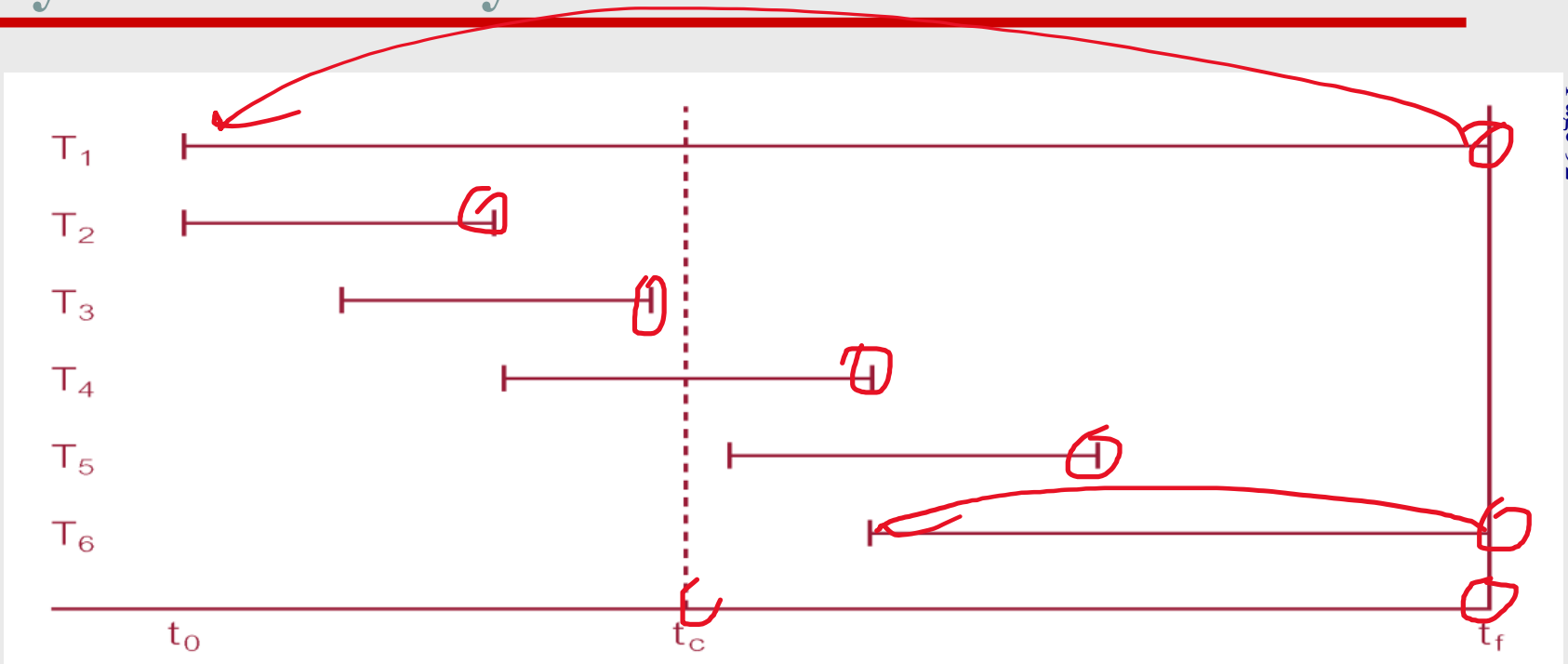
Checkpoint

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

- Time to time (randomly or under some criteria), the database automatically flushes its buffer to database disk to minimize the task of recovery. ("Force-writing")
- Checkpoint record is created containing identifiers of all active transactions at the time the checkpoint was taken.
- When failure occurs, **redo** all transactions that committed since the **checkpoint** and **undo** all transactions active at time of crash.



System Recovery



DBMS starts at time t_0, but fails at time t_f. Assume data for transactions T_2 and T_3 have been written to secondary storage. A checkpoint occurred at point t_c,

- Undo: T_1 and T_6 (Atomicity)
- Redo: T_4 and T_5 (Durability)

Checkpointing Example

- Generally, checkpointing is a relatively inexpensive operation, and it is often possible to take three or four checkpoints an hour. In this way, no more than 15–20 minutes of work will need to be recovered.

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

Two Cases of Recovery

- **If database has been damaged:**
  - Need to restore last backup copy of database and reapply updates of committed transactions using log file.

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

Main Recovery Techniques

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

Deferred Update Recovery Protocol

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

Immediate Update Recovery

- Updates are applied to database as they occur without waiting to reach the commit point.
- Essential that log records are written before write to database. ***Write-ahead log protocol***.
- In the event of failure
  - Need to redo updates of committed transactions following a failure.
  - May need to undo effects of transactions that had not committed at time of failure.

Immediate Update Recovery

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

Shadow Paging Recovery

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

Database Recovery Summary

- Log file
- Checkpoint
- Redo and undo
- Recovery techniques

End