Advanced Database Management Systems

Concurrency Control – Chapter 18
Recovery – Chapter 19

2PL Example

T1	**T2**
	s2(Z)
s1(Y)	
	r2(Z)
r1 (Y)	
x1 (X)	
u1(Y)	
	x2(Y)
r1(X)	
	u2(Z)
w1(X)	
	r2(Y)
u1(X)	
	w2(Y)
	u2(Y)

Both transactions obey 2PL.
It is not possible to interleave them in a manner that results in a non-serializable schedule.

Basic 2PL

- Basic 2PL requires that no locks be requested after the first unlock

- Guarantees serializability
  - transactions that request operations that violate serializability are delayed while waiting on locks

- Reduces concurrency, since locks must be held until all needed locks have been acquired

- May cause deadlock

Conservative 2PL

- Conservative 2PL requires that
  all locks must be acquired at start of transaction

- Prevents deadlock, since all locks
  are acquired as a block
  - No transaction can be waiting on one lock
    while it holds another lock

- Further restricts concurrency,
  since transaction must request strongest lock
  that *might* be needed

Strict 2PL

- Strict 2PL requires that
  all locks must be held until end of transaction
- Deadlock is possible
- Guarantees strict schedules
- May require holding locks longer than necessary
- Most commonly used algorithm

Serializablity: Commutativity

- Two operations commute if,
  when executed in either order:
  - The values returned by both are the same and
  - The database is left in the same final state

- Two schedules are equivalent if one
  can be derived from the other
  by a series of simple interchanges
  of commutative operations

- A schedule is serializable
  if it is equivalent to a serial schedule

Serializablity: Commutativity

- S: $r_1(x)\ w_2(z)\ w_1(y)$
  is equivalent either serial schedules of *T1* and *T2*
  T1, T2: $r_1(x)\ w_1(y)\ w_2(z)$
  T2, T1: $w_2(z)\ r_1(x)\ w_1(y)$

  since operations of distinct transactions on different data items *commute*.

- S is a serializable schedule

Serializablity: Commutativity

- Schedule

  S: $r_1(z)\ r_2(q)\ w_2(z)\ r_1(q)\ w_1(y)$

  is equivalent to the serial schedule *T1, T2:*

  $r_1(z)\ r_1(q)\ w_1(y)\ r_2(q)\ w_2(z)$

  since read operations of distinct transactions on the same data item commute.
- *S* is *not* equivalent to *T2, T1*

  since read and write operations of distinct transactions on the same data item *do not* commute

Correctness of 2PL

- Intuition: Active transactions cannot have executed operations that do not commute, since locks required for non-commutative operations conflict

- A schedule produced by a 2PL is serializable, since operations of concurrent transactions can always be reordered to produce a serial schedule

2PL: Deadlock

T1	**T2**
s1(Y)	s2(X)
r1 (Y)	r2(X)
x1 (X)	x2(Y)
u1(Y)	u2(X)
r1(X)	r2(Y)
X:=X+Y	Y:=X+Y
w1(X)	w2(Y)
u1(X)	u2(Y)

T1 cannot proceed until T2 releases lock on X.
T2 cannot proceed until T1 releases lock on Y.
→ DEADLOCK

Conservative 2PL: ~~Deadlock~~

T1	**T2**
s1(Y), x1 (X)	s2(X), x2(Y)
r1 (Y)	r2(X)
u1(Y)	u2(X)
r1(X)	r2(Y)
X:=X+Y	Y:=X+Y
w1(X)	w2(Y)
u1(X)	u2(Y)

In this case, the only possible schedules are serial schedules.

Locks must be acquired as a unit at beginning of transaction.
Transaction cannot be holding locks while waiting on locks.
Deadlock is not possible.

Conservative 2PL: ~~Deadlock~~

T1	T2
s1(Y), x1 (Z)	
r1 (Y)	
u1(Y)	
r1(Z)	s2(X), x2(Y)
Z:=Z+Y	r2(X)
w1(Z)	u2(X)
u1(Z)	r2(Y)
	Y:=X+Y
	w2(Y)
	u2(Y)

T2 can proceed as soon as T1 releases lock on Y.

Concurrency is still possible under conservative 2PL.

Deadlock: Detection/Resolution

- Let deadlocks happen, then resolve the problem
- **Wait-for graph**
  - scheduler maintains a *wait-for graph*
    - arc from Tx to Ty indicates Tx is waiting for a lock held by Ty
  - when a transaction is blocked, it is added to the graph
  - a cycle in the wait-for-graph indicates deadlock
  - one transaction involved in the cycle is selected (victim) and rolled-back
- **Timeout**
  - abort any transaction that has been waiting for some set amount of time
  - simple solution, but may be abort a transaction that could eventually proceed

Deadlock: Prevention

- **Locking policy**
  - Implement a CC policy that never allows deadlock to occur
  - Example: conservative 2PL
- **Waits-for cycle avoidance**
  - Use wait-for graph, but do not allow cycles to occur
  - Example: any transaction that would create a cycle is aborted
  - Other algorithms use timestamps to chose victim

Deadlock Victim Selection

- T1 tries to lock X, T2 holds lock on X
  - **wait-die:**
    if T1 is older than T2, T1 waits
    otherwise, T1 aborts
  - **wound-wait:**
    if T1 is older than T2, T2 aborts,
    otherwise, T1 waits
  - **no-waiting:**
    T1 aborts
  - **cautious waiting:**
    if T2 is waiting, T1 aborts,
    otherwise T1 waits

Starvation

- **Starvation**
  - A particular transaction consistently waits or gets restarted and never gets a chance to complete
  - Caused by deadlock victim selection policy
  - Inherent in all priority based scheduling mechanisms

  - Example: Wound-Wait
    a younger transaction may always be aborted
    by a long running older transaction,

Multiversion Protocols

Multiversion concurrency control techniques

- Maintain versions of a data item and allocate correct version to a read operation of a transaction
- read operation is never rejected.
- requires significantly more storage to maintain versions
- requires garbage collection to remove unneeded versions

Optimistic CC

- Serializability is tested at the time of commit

- **Read phase**:
  - transaction can read values of committed data items
  - writes are applied only to local copies (versions) of the data
- **Validation phase:**
  - Serializability is checked before transactions
    write their updates to the database
- **Write phase:**
  - if serializability check passed, write updates to database
    otherwise, abort (or restart)

Summary: 2PL

- prevents unwanted schedules by delaying conflicting operations
- + guarantees equivalence to some serial schedule
- + never requires transaction aborts due to conflict
- - reduces concurrency
- - may cause deadlock, livelock or starvation

Summary: Timestamp

- rejects transactions that request operations that are out of order
- order is determined by unique timestamps assigned to each transaction
- + guarantees equivalence to a particular serial schedule
- + cannot cause deadlock
- - may cause (cascading) aborts due to conflict
- - may cause starvation

Summary: Multiversion

- keep multiple versions of modified data items and selects the appropriate versions that each transaction sees
- + reads can proceed concurrently with conflicting writes
- + avoids cascading aborts due to conflict
- - requires additional storage space and maintenance
- - may cause deadlock
- - transaction commit may be delayed

Summary: Optimistic

- No checking is done while a transaction is executing
- All operations are performed
  on local copies of data items
- Validity of the transaction is checked at commit ,
  invalid transactions are aborted
- + maximal concurrency
- + no possibility of deadlock
- - may cause aborts due to conflict
  (conflict can be tested using precedence graphs)
- - determination of validity is delayed
  until latest possible time

Summary: Multistate

- keep multiple values for everything for each transaction
- merge resulting states by resolving conflicts
- + useful for applications with long transactions (i.e. computer-aided design tools)
- + some applications never merge states (temporal databases)
- - conflict resolution may require user intervention

Database Recovery

Purpose of Database Recovery

- Bring the database into the
  most recent consistent state
  that existed prior to a failure.

- Preserve transaction properties
  - Atomicity, Consistency, Isolation and Durability

- Example:
  - bank database crashes before a fund transfer transaction completes
  - either one or both accounts may have incorrect values
  - database must be restored to the state
    before the transaction modified any of the accounts

Types of Failure

The database may become unavailable due to

- **Transaction failure**: Transactions may fail because of incorrect input, deadlock, incorrect synchronization.
- **System failure**: System may fail because of addressing error, application error, operating system fault, RAM failure, etc.
- **Media failure**: Disk head crash, power disruption, etc.

Transaction Log

- Recovery from failures, may require
  - data values prior to modification: BFIM - BeFore Image
  - new value after modification: AFIM – AFter Image
- These values and other information are stored in a sequential file - a ***transaction log***
- Sample log data:

T ID	Back P	Next P	Operation	Data item	BFIM	AFIM
T1	0	1	Begin			
T1	1	4	Write	X	X = 100	X = 200
T2	0	8	Begin			
T1	2	5	W	Y	Y = 50	Y = 100
T1	4	7	R	M	M = 200	M = 200
T3	0	9	R	N	N = 400	N = 400
T1	5	nil	End			

Data Update Options

- **Immediate Update:**
  As soon as a data item is modified in cache,
  the disk copy is updated
- **Deferred Update:**
  Modified data items in the cache are written
  to disk either after a transaction ends its execution,
  or after a fixed number of transactions have
  completed their execution

Data Caching

- Modified data items are first stored into a cache, and later flushed (written) to the disk

- The flushing is controlled by **Dirty** and **Pin** bits (flags)
  - **Pin:** A pinned data item cannot be flushed from the cache
  - **Dirty (Modified):** A data item has been modified and must eventually be flushed to disk

Cache Flushing

- **In-Place Update:**
  Modified values in cache
  replace actual values on disk

- **Shadow update:**
  Modified version of a data item does not overwrite disk copy but is written at a separate disk location

Undo and Redo

- To maintain atomicity,
  a transaction’s operations
  may need to be redone or undone

- **Undo (roll-back):**
  - restore all BFIMs to disk (replace all AFIMs)
- **Redo (roll-forward):**
  - restore all AFIMs to disk

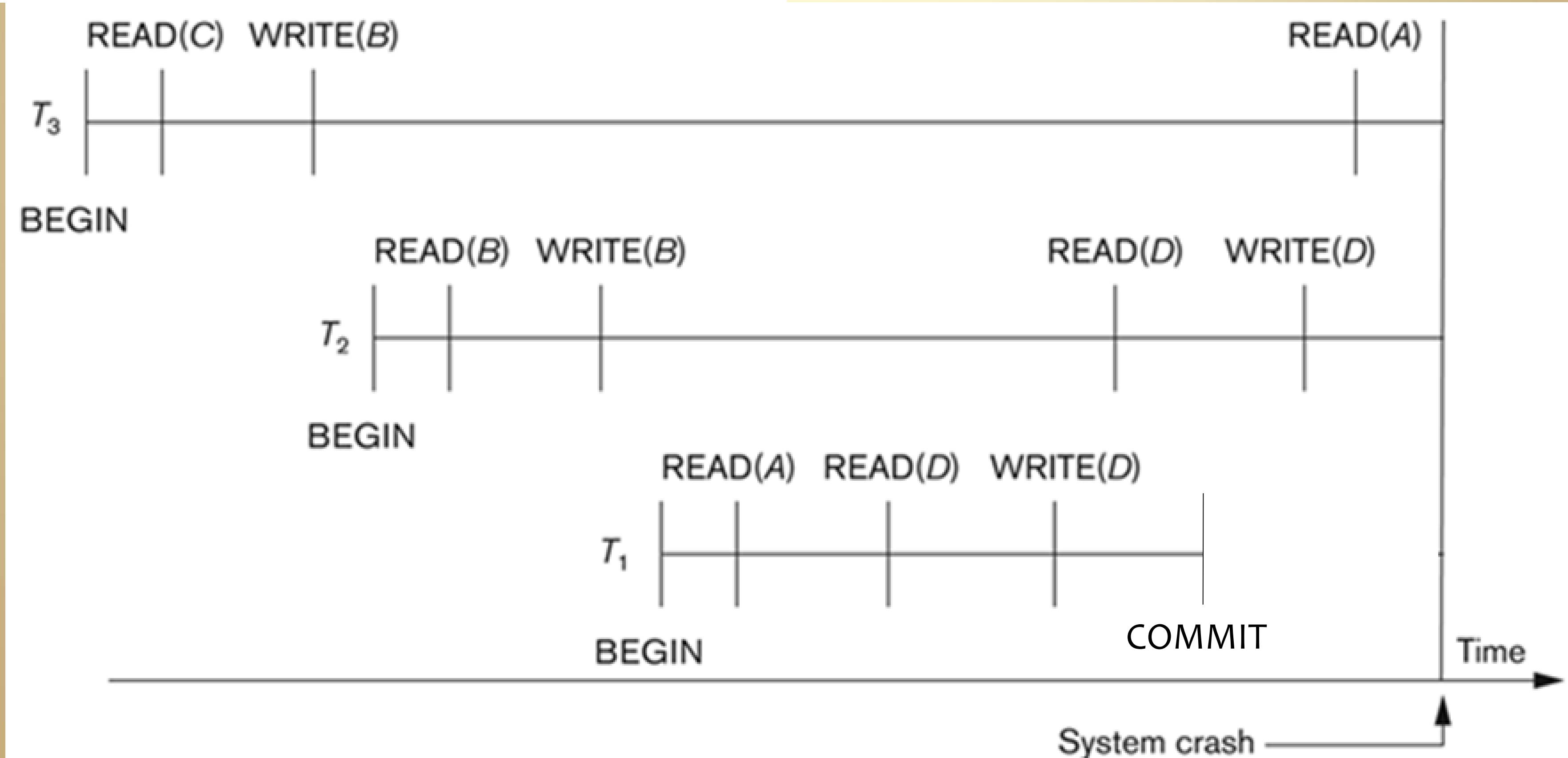
Roll-back Example

T_1
read_item(A)
read_item(D)
write_item(D)

T_2
read_item(B)
write_item(B)
read_item(D)
write_item(D)

T_3
read_item(C)
write_item(B)
read_item(A)
write_item(A)

Three concurrent transactions and timeline before system crash



Roll-back Example

Transaction log at time of crash

	A	B	C	D
	30	15	40	20
[start_transaction, T_3]				
[read_item, T_3, C]				
[write_item, T_3, B, 15, 12]		12		
[start_transaction, T_2]				
[read_item, T_2, B]				
[write_item, T_2, B, 12, 18]		18		
[start_transaction, T_1]				
[read_item, T_1, A]				
[read_item, T_1, D]				
[write_item, T_1, D, 20, 25]				25
[read_item, T_2, D]				
[write_item, T_2, D, 25, 26]				26
[read_item, T_3, A]				

← System crash

Roll-back Example

	A	B	C	D
	30	15	40	20
[start_transaction,T_3]				
[read_item,T_3,C]				
[write_item,T_3,B,15,12]		12		
[start_transaction,T_2]				
[read_item,T_2,B]				
[write_item,T_2,B,12,18]		18		
[start_transaction,T_1]				
[read_item,T_1,A]				
[read_item,T_1,D]				
[write_item,T_1,D,20,25]				25
[read_item,T_2,D]				
[commit_transaction, T_1]				
[write_item,T_2,D,25,26]				26
[read_item,T_3,A]				

← System crash

T3 is rolled-back, since it has not yet committed

T2 is also rolled-back, since it read values written by T3

T1 is has committed and is not dependent on other transaction, so it's updates should remain in database

Restored database state should be <30, 15, 40, 25>

Write-Ahead Logging

- The **Write-Ahead Logging (WAL)** protocol insures that log is consistent with database state at the time of a crash

- WAL states that
  - **For Undo**: Before a data item's AFIM is flushed to the database disk (overwriting the BFIM) its BFIM must be written to the log
  - **For Redo**: Before a transaction executes its commit operation, all its AFIMs must be written to the log
  - In both cases, the log must be saved in stable storage, before the flush or commit is processed.

Recover Schemes

- Steal = no pinning
  - can flush data items to recover buffer space
  - smaller buffer space requirements
- No-steal = pinning
  - cannot flush pinned data items before xact commits
  - may require larger buffer space
- Force
  - dirty data items must be flushed when xact commits
- No-force
  - dirty data items do not have to be flushed at commit (but do need to be flushed eventually)

Recover Schemes

- The force/no-force and steal/no-steal protocols used determine the recovery scheme:

Steal/No-Force → Undo/Redo
Steal/Force → Undo/No-redo
No-Steal/No-Force → No-undo/Redo
No-Steal/Force → No-undo/No-redo

Checkpointing

- From time to time (randomly or under some criteria) database flushes its buffer to database disk to minimize the task of recovery

- The following steps define a checkpoint operation:
  - Suspend execution of transactions temporarily.
  - Force write modified buffer data to disk.
  - Write a [checkpoint] record to the log, save the log to disk.
  - Resume normal transaction execution.

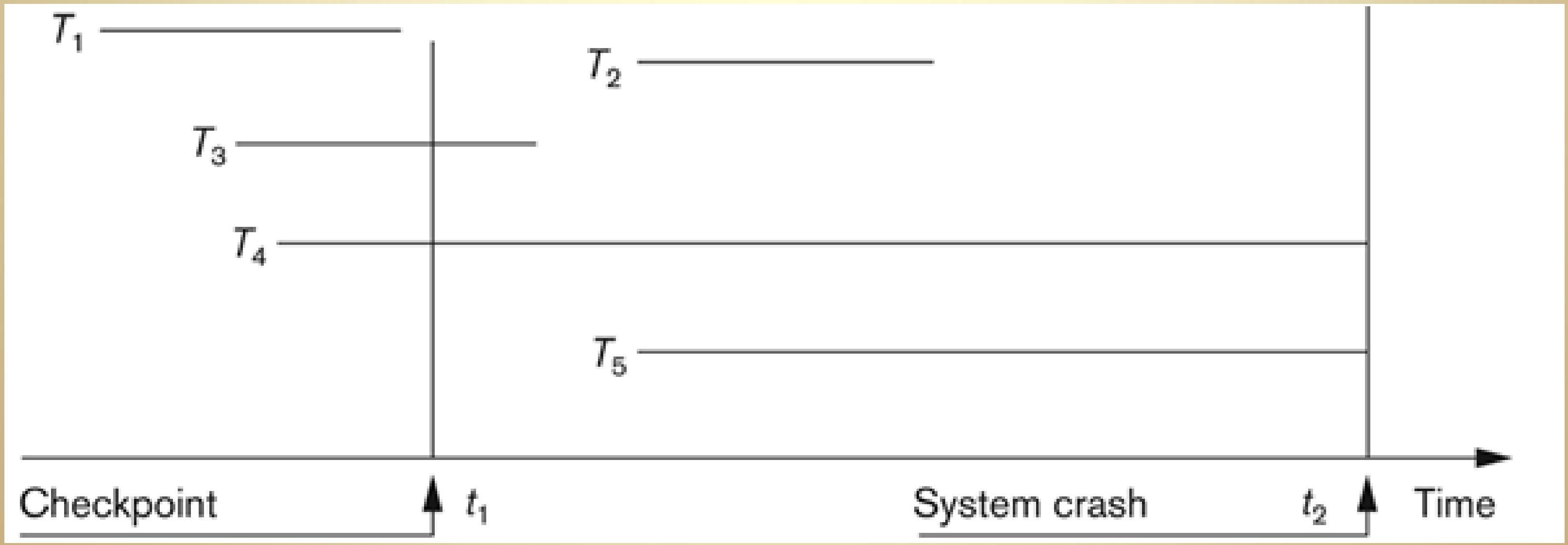
- During recovery redo or undo may be required for transactions appearing after [checkpoint] record.

Recovery: Deferred Update

- No-Undo/Redo
  - assume no-steal/force
  - during transaction, updates are only in cache and log
  - disk is updated at commit
- After reboot from a failure the log is used to redo all the transactions affected by this failure
  - No undo is required because no AFIM is flushed to the disk before a transaction commits

Recovery: Deferred Update

- T1 is already in checkpoint, no action required
- T2 and T3 are in log and must be redone
- T4 and T5 were not committed and can be ignored



Recovery: Immediate Update

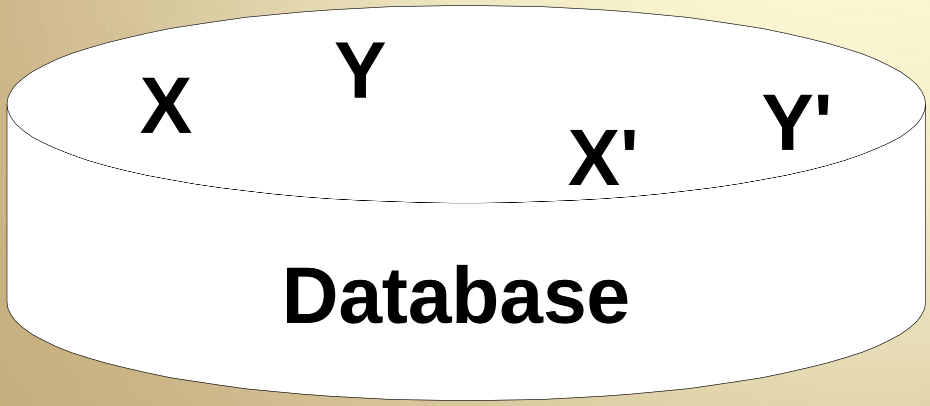
- **Undo/No-redo**
  - assume steal/force
  - WAL of BFIMs required
  - AFIMs of a transaction are flushed to the disk at commit
- **undo** all active transactions during recovery
- no transaction needs to be **redone**

Recovery: Immediate Update

- **Undo/Redo**
  - assume steal/no-force
  - WAL of BFIMs and AFIMs required
  - checkpointing used
- **undo** all active transactions during recovery
- **redo** all transactions that committed since last checkpoint

Recovery: Shadowing

- The AFIM does not overwrite its BFIM
  but is recorded at another place on the disk
- At any time a data item has AFIM and BFIM
  (shadow copy of the data item)
  at two different places on the disk
- NO-UNDO/NO-REDO
- Requires eventual merging of shadow to current database



X and Y: Shadow copies of data items
X' and Y': Current copies of data items