# COMP163

Database Management Systems

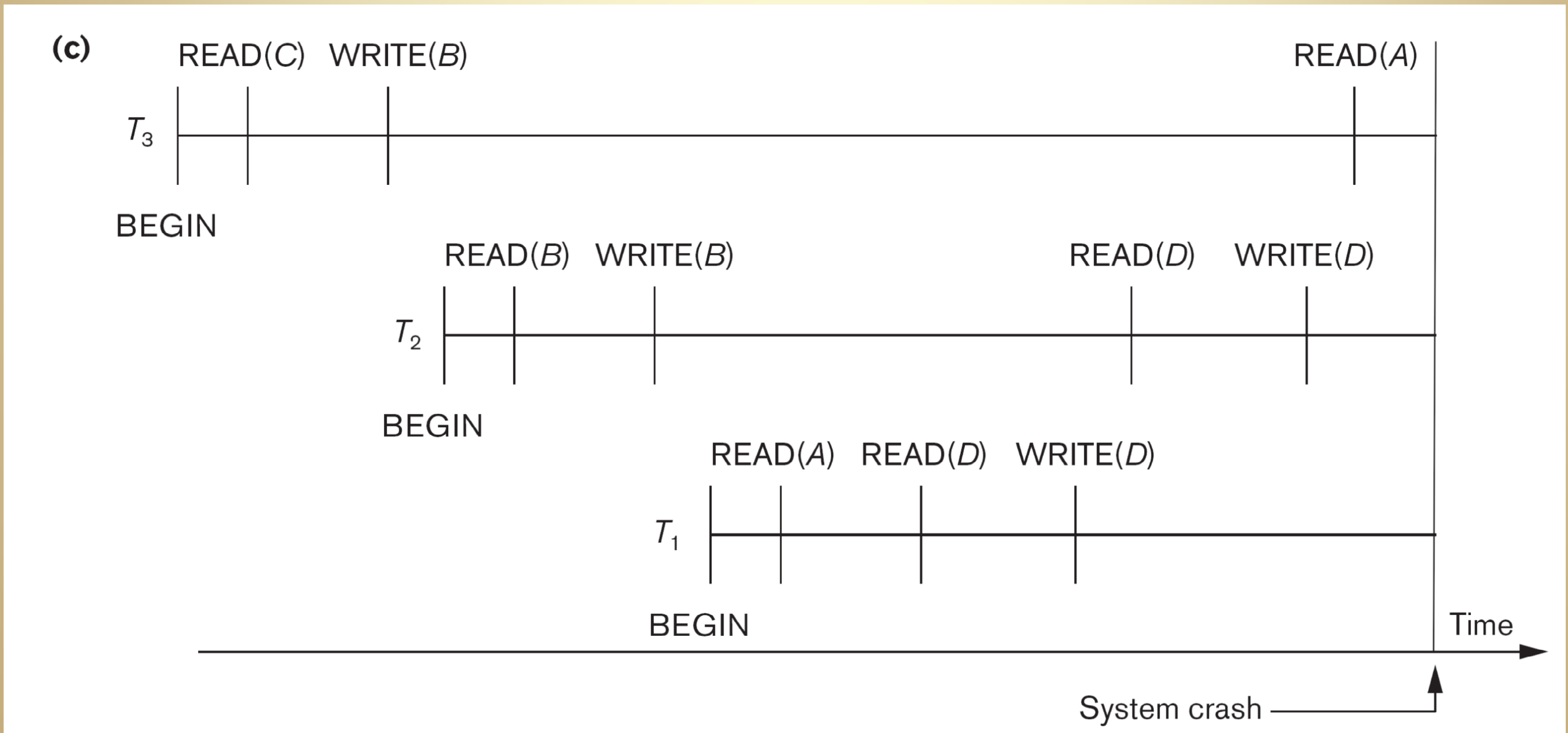
**Database Recovery Techniques**
**Chapter 23**

# Discussion Notes

- We started the discussion with notes on recover in Oracle and SQL Server
  - SQL Server: http://msdn.microsoft.com/en-us/library/ms189275.aspx
  - Oracle: http://docs.oracle.com/cd/B10501_01/server.920/a96519/strategy.htm

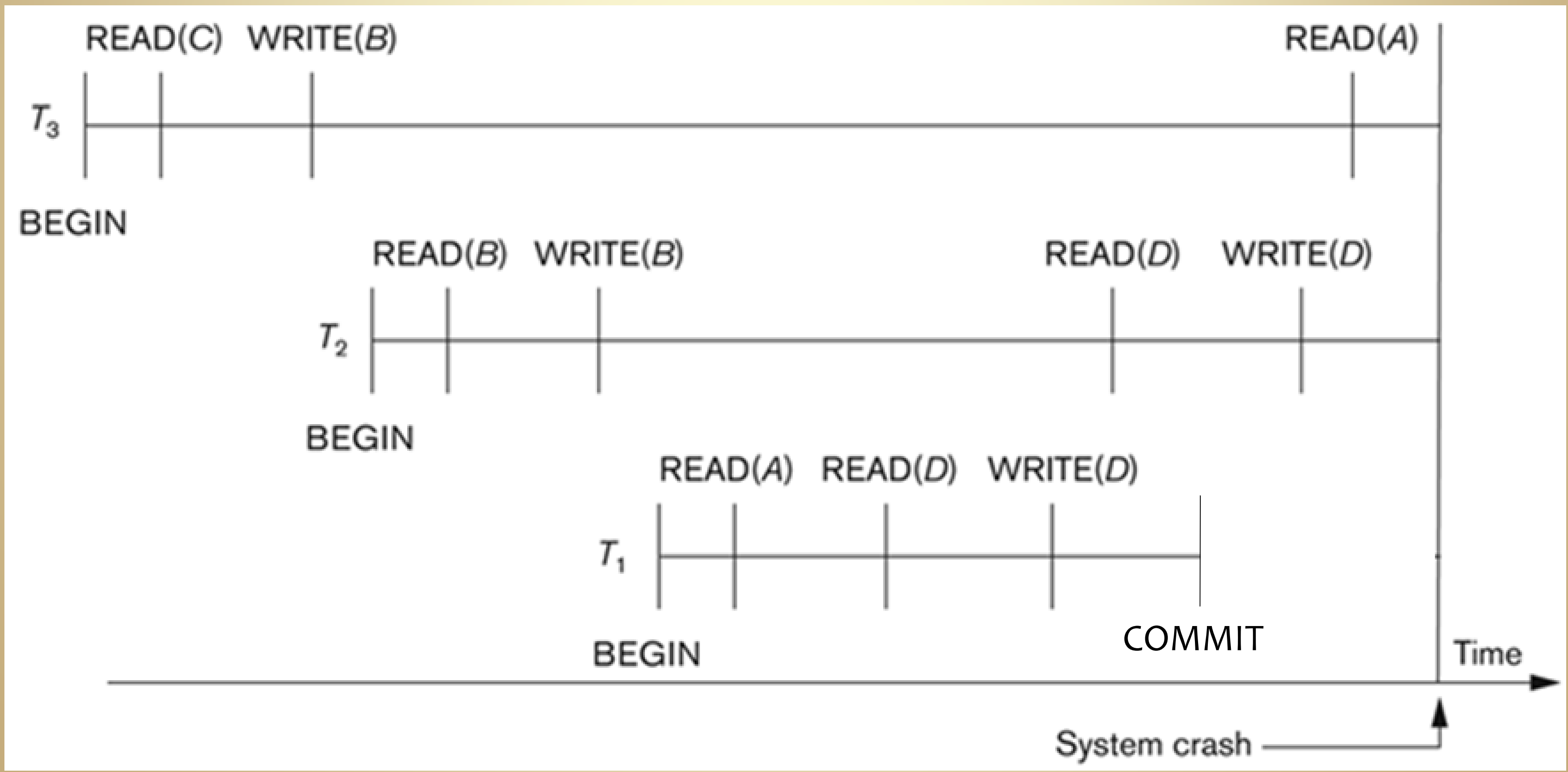
# Recovery Example

Three concurrent transactions and timeline before system crash



# Recovery Example

Three concurrent transactions and timeline before system crash



# Purpose of Database Recovery

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

- Preserve transaction properties
  - Atomicity 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, …

- **Media failure**:
  Disk head crash, power disruption, …

# Transaction Log

- Recovery from failures, requires
  - 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

# Deferred Update

- **Active Transactions:**
  - transaction updates stored in local workspace or in main memory cache
- **Partially Committed:**
  - updates are transferred to system log
- **Commit:**
  - updates written to disk
- This is a **NO-UNDO/REDO** algorithm
  - uncommitted transactions have not changed database
  - committed transactions are logged

# Immediate Update

- **Active Transactions**:
  - updates may reach disk before transaction commits
  - updates must be logged before it is written to disk
- **Transaction Failure:**
  - updates must be removed from the disk

- This is an **UNDO/REDO** algorithm

- If we require all updates reach disk before commit, then REDO is not necessary

# Data Caching

- Modified data items are first stored into a cache, and later flushed and 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

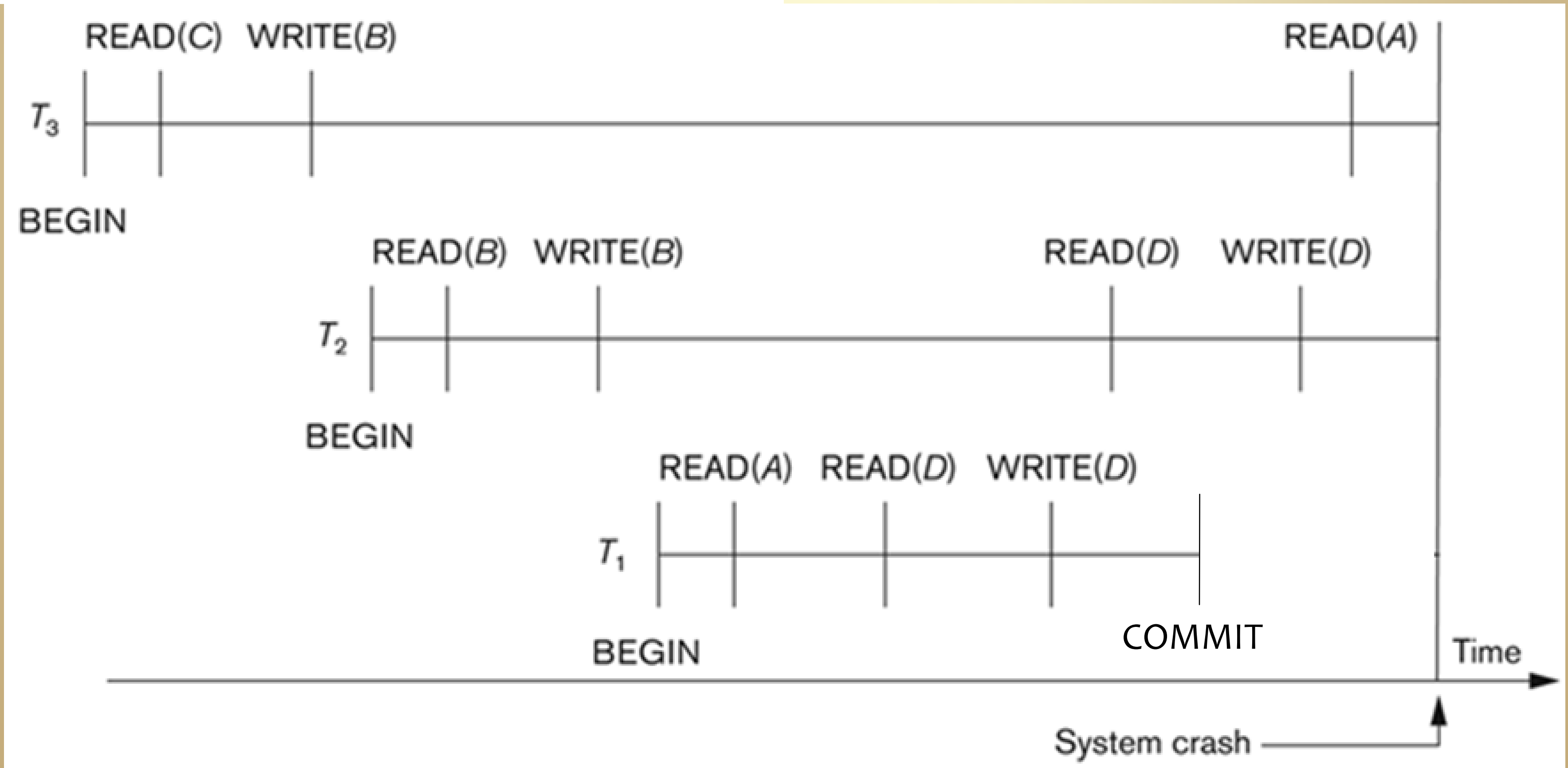
# Undo and Redo

- 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

| $T_1$ | $T_2$ | $T_3$ |
|---|---|---|
| read_item($A$) | read_item($B$) | read_item($C$) |
| read_item($D$) | write_item($B$) | write_item($B$) |
| write_item($D$) | read_item($D$) | read_item($A$) |
| | write_item($D$) | 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,$T_3$,$A$] | | | | |

[commit, $T_1$]

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.

# Cache Control Options

- 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)

# Recovery 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