LECTURE 6

TRANSACTION MANAGEMENT (TRANSACTION SUPPORT, CONCURRENCY CONTROL & DATABASE RECOVERY)

Muhammad Hamiz Mohd Radzi
Faiqah Hafidzah Halim

Content

- **Transaction Support**
  - *Properties of transaction*
  - *Database Architecture*
- **Concurrency Control**
  - *The need for concurrency control*
  - *Serializability and Recoverability*
  - *Locking Methods*
  - *Deadlock*
  - *Granularity of Data Items*
- **Database Recovery**

  The need for recovery
  Transactions and recovery
  Recovery Facilities
  Recovery Technique

Objectives

- At the end of this lesson, you should be able to:
  - *Explain transaction support, properties of transaction and the database architecture.*
  - *Explain concurrency control, lost update, uncommitted dependency and inconsistent analysis problems.*
  - *Explain the serializability schedule, serial and non-serial schedule, non-conflict serializability precedence graph, and its recoverability.*

- *Describe the locking, shared and exclusive lock, 2PL's shrinking and growing phase, concurrency problems' solution by using 2PL, cascading rollback, deadlock prevention and detection, and time-stamping concept.*
- *Describe the granularity data items and its hierarchy.*
- *Explain the database recovery and why it is needed.*
- *Explain the UNDO and REDO recovery mechanism*
- *Describe the log file and checkpoint backup mechanism*
- *Explain the deferred and immediate update, and shadow paging recovery techniques.*

Transaction Support

- *Transaction: **Action**, or **series of actions**, carried out by user or application, which **reads or updates** contents of database.*
- *Each operation on database can be considered as transactions.*
- *During transactions, database is transform from consistent state to another state, although the consistency might be violated.*

Example of Transaction

```
read(staffNo = x, salary)
salary = salary * 1.1
write(staffNo = x, new_salary)
```

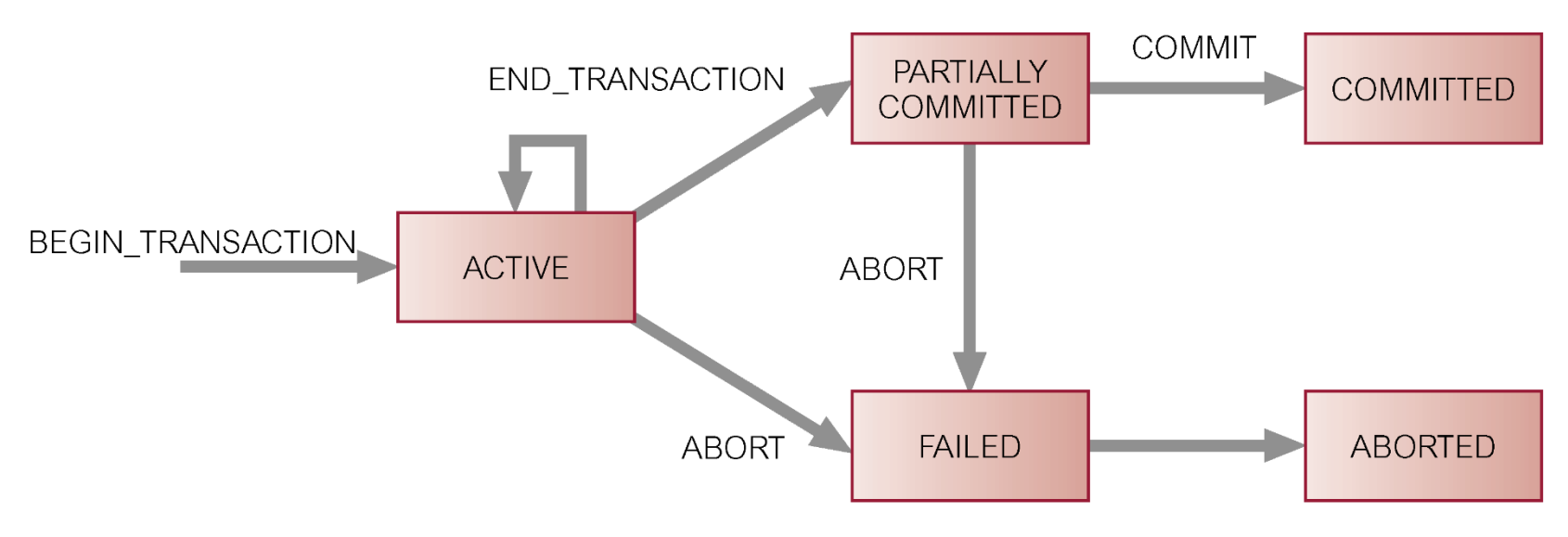
(a)

```
delete(staffNo = x)
for all PropertyForRent records, pno
begin
    read(propertyNo = pno, staffNo)
    if (staffNo = x) then
    begin
        staffNo = newStaffNo
        write(propertyNo = pno, staffNo)
    end
end
```

(b)

- Can have one of **two outcomes**:
  - ***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.*
- **Committed** transaction cannot be aborted.
- Aborted transaction that is **rolled back** can be restarted later.

State Transition Diagram for Transaction

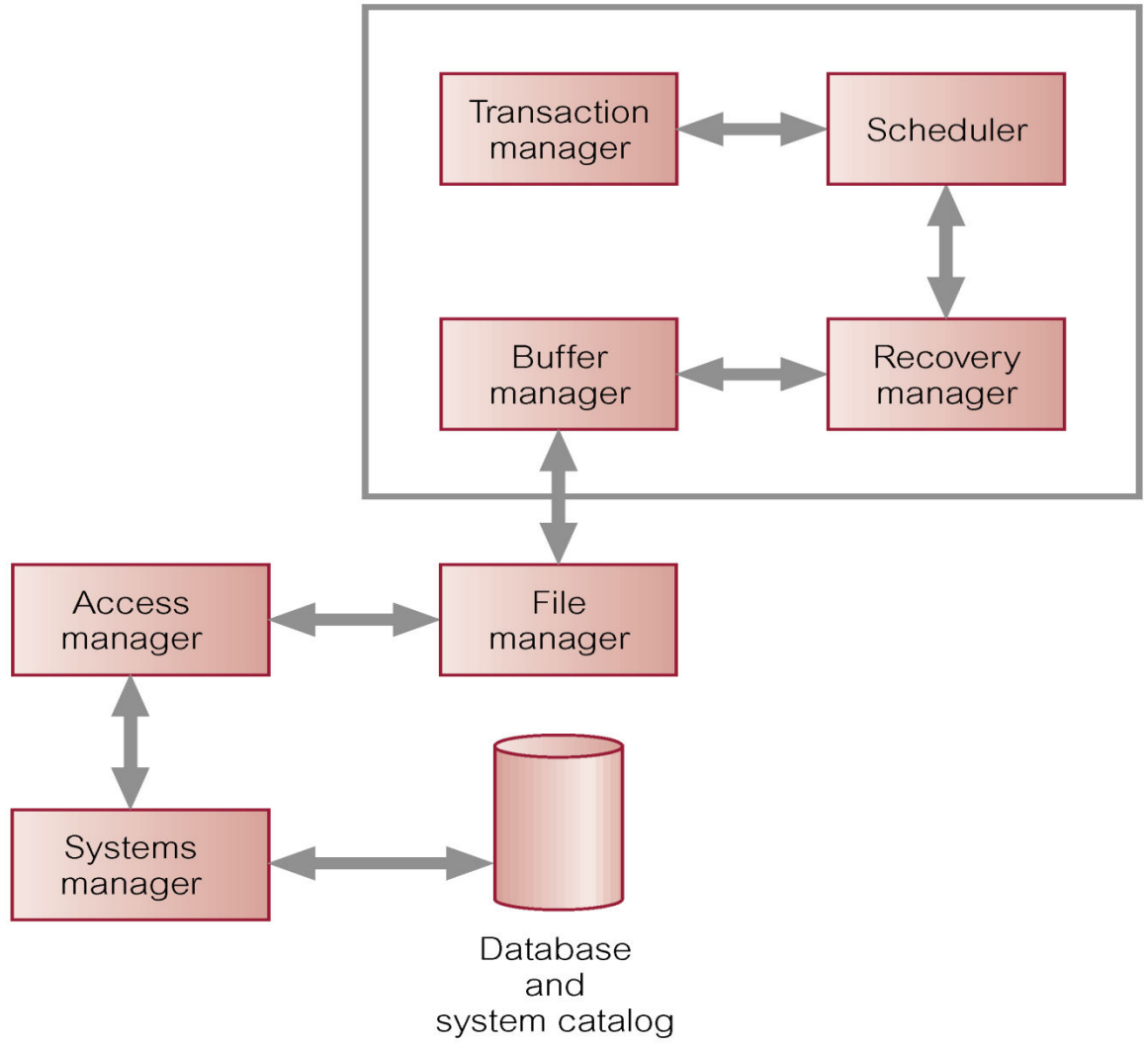


Properties of Transaction

- Four basic (ACID) properties of a transaction are:

Atomicity	‘All or nothing’ property.
Consistency	Must transform database from one consistent state to another.
Isolation	Partial effects of incomplete transactions should not be visible to other transactions.
Durability	Effects of a committed transaction are permanent and must not be lost because of later failure.

DBMS Transaction Subsystem



- **Transaction manager**: Coordinates transaction on behalf of application program. Communicates with scheduler
- **Scheduler**: Module responsible for implementing a particular strategy for concurrency control. Sometimes referred to as the **lock manager** if the concurrency protocol is locking-based.
- **Recovery manager**: Ensure database is restored to the state it was in before the start of the transaction and therefore a consistent state.
- **Buffer manager**: Responsible for the efficient transfer of data between disk storage and main memory

Concurrency Control

- Defined as process of managing simultaneous operations on the database without having them interfere with one another.
- Transaction can interleave with each other, but it cannot interfere.
- If 2 users want to update the same bank account at the same time, there will be some incorrect balance at the end of the transactions.
- Although two transactions may be correct in themselves, interleaving of operations may produce an incorrect result.

Need for Concurrency Problem

- Concurrency control is needed because of, **everybody want everything to settle fast with a correct result**.

- If there is no concurrency control, the **problems** that might occurs are:

  - *Lost Update Problem*
  - *Uncommitted Dependency Problem*
  - *Inconsistent Analysis 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.

Uncommitted Dependency Problem

- 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

- 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.

Inconsistent Analysis Problem

- Occurs when transaction reads several values but second transaction updates some of them during execution of first.

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

- Sometimes referred to as *dirty read* or *unrepeatable read*.
- 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).

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.

Schedule	**Sequence of reads/writes by set of concurrent transactions.**
Serial Schedule	**Schedule where operations of each transaction are executed consecutively without any interleaved operations from other transactions.**
Non Serial Schedule	**Schedule where operations from set of concurrent transactions are interleaved.**

- No guarantee that results of all serial executions of a given set of transactions will be identical.
- 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*.

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.**

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)

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

(b)

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

(c)

- Equivalent schedule:

  (a) Non serial Schedule S1;

  (b) Non serial schedule S2;

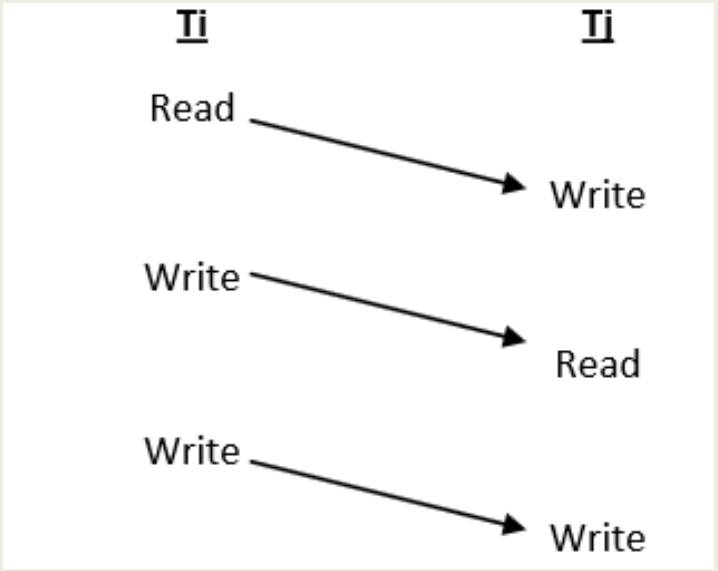
  (c) Serial schedule S3, equivalent to S1 and S2

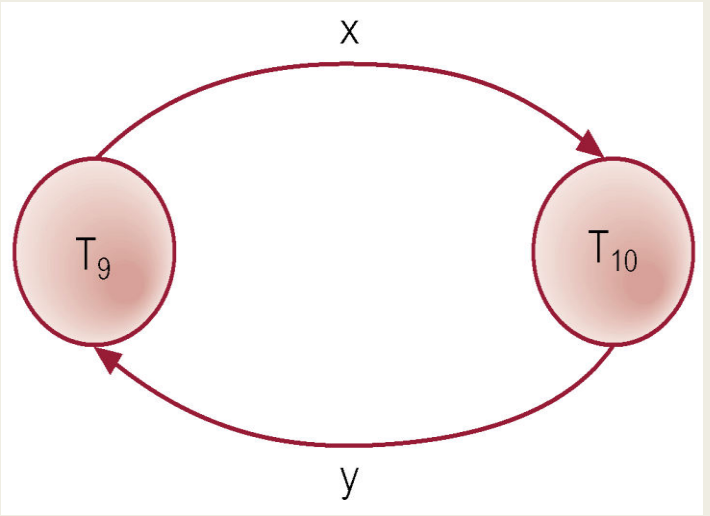
- Conflict serializable schedule orders any conflicting operations in same way as some serial execution.

Testing for Conflict Serializibility

- Under constrained write rule (transaction updates data item based on its old value, which is first read), use precedence graph to test for serializability.

- Create:
  - ***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 write by* T_i*.*





- If precedence graph contains cycle schedule is not conflict serializable.

Example Precedence Graph

$S = [R_1(Z), R_2(Y), W_2(Y), R_3(Y), R_1(X), W_1(X), W_1(Z), W_3(Y), R_2(X), R_1(Y), W_1(Y), W_2(X), R_3(W), W_3(W)]$

By using precedence graph, determine whether this transaction is serializable or not.

View Serializability

- Offers less stringent definition of schedule equivalence than conflict serializability.

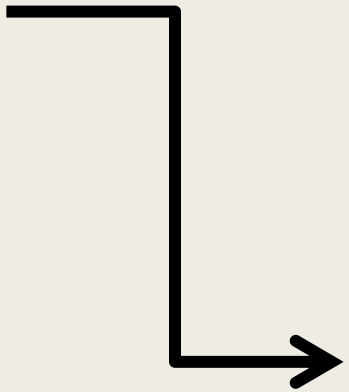
- Two schedules S_1 and S_2 are view equivalent if:
  - *For each data item x, if T_i reads initial value of x in S_1, T_i must also read initial value of x in S_2.*
  - *For each read on x by T_i in S_1, if value read by x is written by T_j, T_i must also read value of x produced by T_j in S_2.*
  - *For each data item x, if last write on x performed by T_i in S_1, same transaction must perform final write on x in S_2.*

Motivating Example

Determine whether S1 is View Serializable with S2

T1	T2	T3
		Read A
	Read A	
		Write A
Read A		
Write A		

S1



S2

T2	T3	T1
Read A		
	Read A	
	Write A	
		Read A
		Write A

- Rules 1: Initial Read
  - *Data item A:*

S1	S2
T3, T2	T2, T3

- Rules 2: Write-Read Conflict
  - *Data item A:*

S1	S2
T3 → T1	T3 → T1

- Rules 3: Last Write
  - *Data item A:*

S1	S2
T1	T1

- 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 (solutions can be verified in polynomial time – computer time).

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.

Concurrency Control Techniques

- Two basic concurrency control techniques:
  - *Locking,*
  - *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.

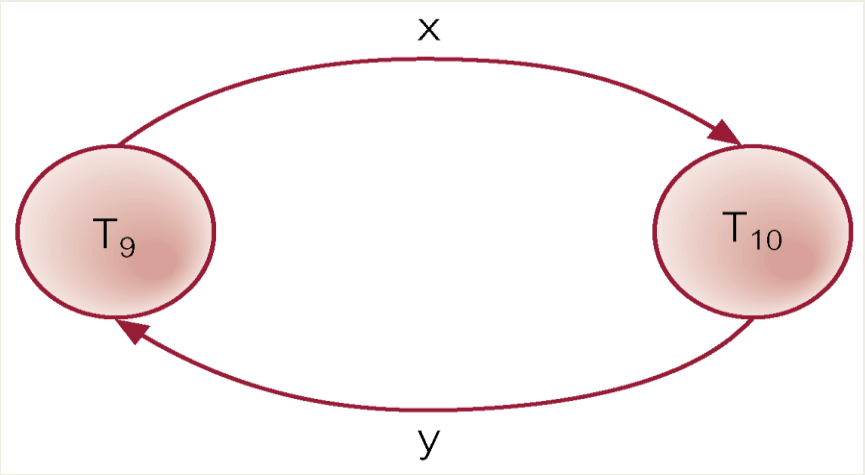
Locking

- Transaction uses locks to deny access to other transactions and to prevent incorrect updates.
- Most widely used approach to ensure serializability.
- Generally, a transaction must claim a *shared* (*read*) or *exclusive* (*write*) 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.

- If transaction has shared lock (S) on item, can read but not update item.
- If transaction has exclusive lock (X) on item, can both read and update item.
- **Reads cannot conflict**, so more than one transaction can hold shared locks (S) simultaneously on same item.
- Exclusive lock (x) gives transaction **exclusive access** to that item.
- Some systems **allow transaction** to upgrade read lock to an exclusive lock, or downgrade exclusive lock to a shared lock.

Locking Method– Incorrect Locking Schedule

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	



Locking Methods– Incorrect Locking Schedule

- Example - Incorrect Locking Schedule:

For two transactions above, a valid schedule using these rules is:

$S = \{write\_lock(T_9, bal_x), read(T_9, bal_x), write(T_9, bal_x), unlock(T_9, bal_x), write\_lock(T_{10}, bal_x), read(T_{10}, bal_x), write(T_{10}, bal_x), unlock(T_{10}, bal_x), write\_lock(T_{10}, bal_y), read(T_{10}, bal_y), write(T_{10}, bal_y), unlock(T_{10}, bal_y), commit(T_{10}), write\_lock(T_9, bal_y), read(T_9, bal_y), write(T_9, bal_y), unlock(T_9, bal_y), commit(T_9)\}$

Locking Methods– Incorrect Locking Schedule

- If at start, $bal_x = 100$, $bal_y = 400$, result should be:
  - *$bal_x = 220$, $bal_y = 330$, if T_9 executes before T_{10}, or*
  - *$bal_x = 210$, $bal_y = 340$, if T_{10} executes before T_9.*
- **However, result gives $bal_x = 220$ and $bal_y = 340$.**
- S is **not a serializable** schedule.

Locking Methods – 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.

Two-Phase Locking (2PL)

- Transaction follows 2PL protocol if all locking operations precede first unlock operation in the transaction.
- Two phases for transaction:
  - ***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

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.

Cascading Rollback

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

Cascading Rollback

- Transactions conform to 2PL.
- T14 aborts.
- Since T15 is dependent on T14, T15 must also be rolled back. Since T16 is dependent on T15, it too must be rolled back.
- This is called cascading rollback.
- To prevent this with 2PL, leave release of all locks until end of transaction.

Deadlock

- An impasse that may result when two (or more) transactions are each waiting for locks held by the other to be released.
- 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.*

Deadlock - 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

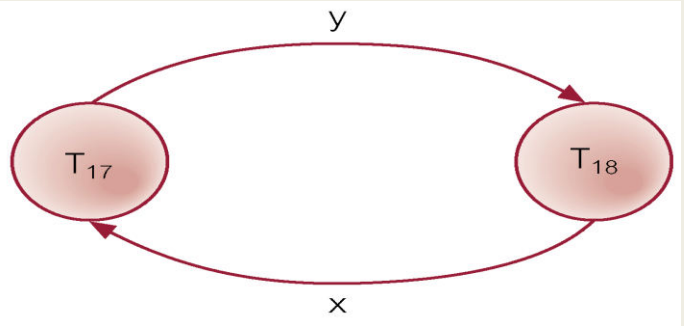
- only an older transaction can wait for younger one, otherwise transaction is aborted (dies) and restarted with same timestamp

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 -> 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.



Recovery from Deadlock Detection

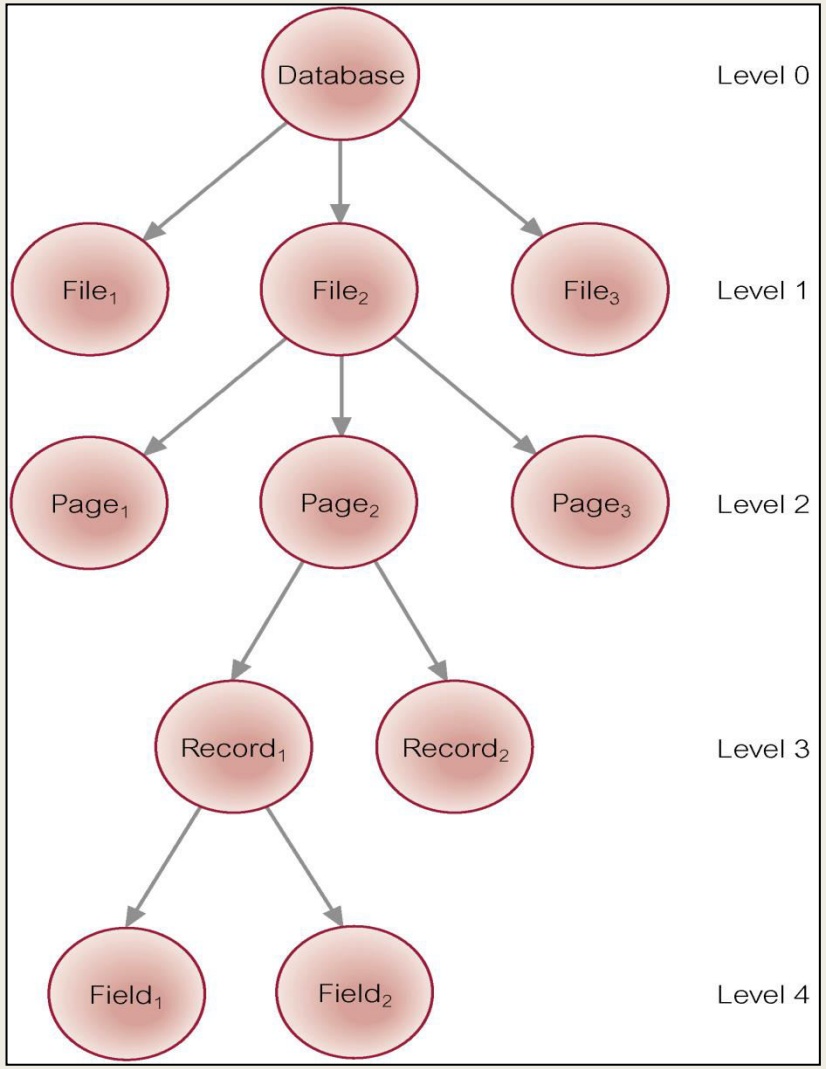
- Several issues:
  - *choice of deadlock victim;*
  - *how far to roll a transaction back;*
  - *avoiding starvation.*

Granularity of Data Items

- Granularity - size of data items chosen as unit of protection by concurrency control protocol.

- Ranging from coarse to fine:
  - *The entire database.*
  - *A file.*
  - *A page (or area or database spaced).*
  - *A record.*
  - *A field value of a record.*

- When node is locked, all its descendants are also locked.



Database Recovery

- No matter what, assume that your data will corrupt one day and you need a mechanism to reconstruct back the data lost.
- Database Recovery: Process of **restoring** database to a **correct state** in the **event of a failure.**
- Need for Recovery Control
  - *Two types of storage: volatile (main memory) and nonvolatile.*
  - *Volatile storage does not survive system crashes.*
  - *Stable storage represents information that has been replicated in several nonvolatile storage media with independent failure modes.*

Types of Failure

- System crashes, resulting in loss of main memory.
- Media failures, resulting in loss of parts of secondary storage.
- Application software errors.
- Natural physical disasters.
- Carelessness or unintentional destruction of data or facilities.
- Sabotage, or intentional corruption or destruction of data, hardware or software facilities

Recovery Facilities

Backing up the database

- *On a regularly scheduled basis, a company's databases must be backed up or copied.*
- *The copy version must be put away from original copy*

Maintaining a journal (Log File)- Keep track the changes to the data such as INSERTION, UPDATE and DELETION

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.

Recovery Techniques

- 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

Deferred Update

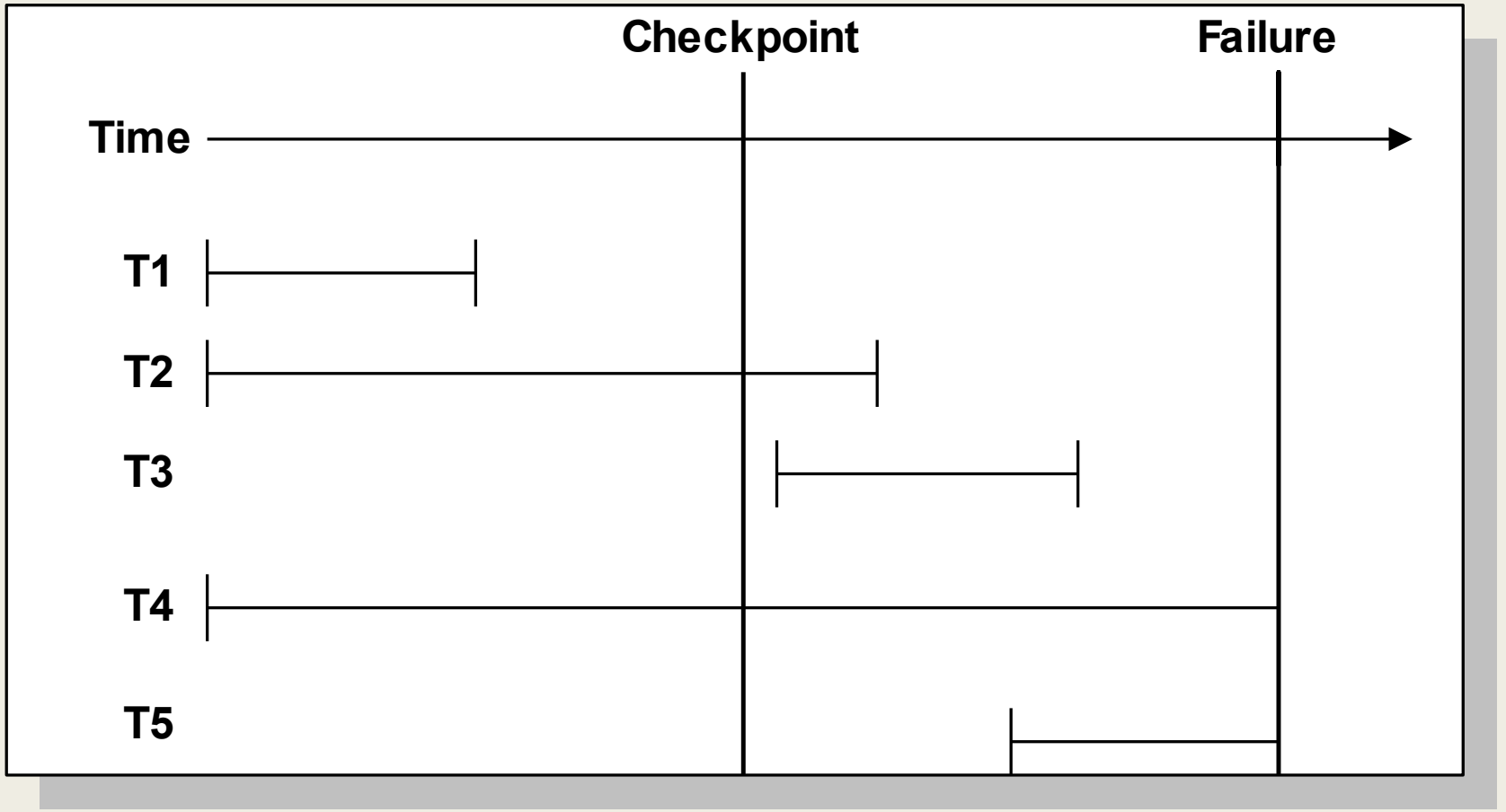
- Updates are not written to the database until after a transaction has reached its commit point.
- If transaction fails before commit, it will not have modified database and so no undoing of changes required.
- May be necessary to redo updates of committed transactions as their effect may not have reached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

Immediate Update

- Updates are applied to database as they occur.
- 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.

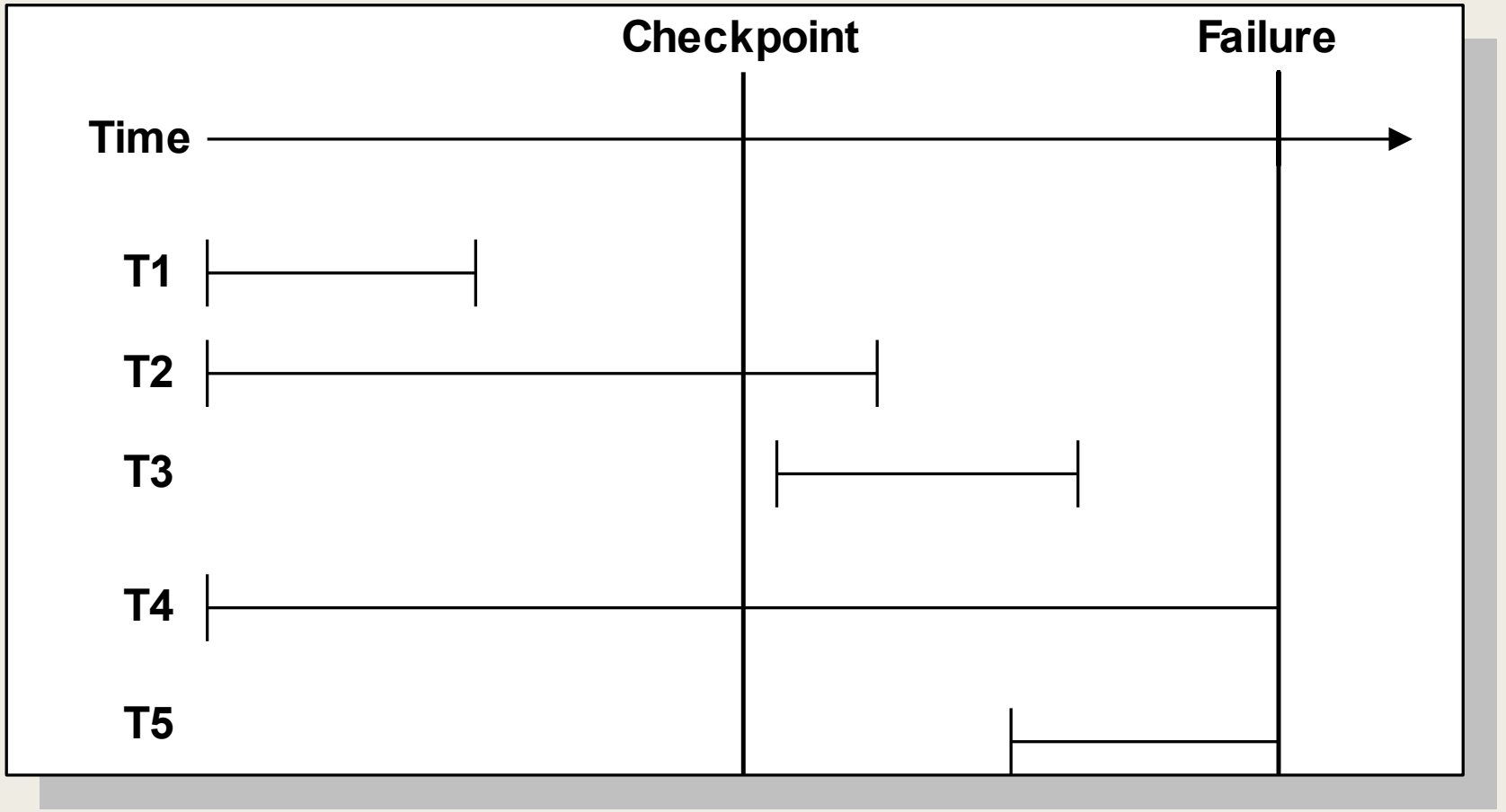
Shadow Paging

- 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.


Checkpoint
Failure
Time
T1
T2
T3
T4
T5

Immediate Update

Class	Description	Restart Work
T1	Finished before CP	None
T2	Started before CP; finished before failure	Redo forward from checkpoint
T3	Started after CP; finished before failure	Redo forward from checkpoint
T4	Started before CP; not yet finished	Undo backwards from most recent log record
T5	Started after CP; not yet finished	Undo backwards from most recent log record


Checkpoint
Failure
Time
T1
T2
T3
T4
T5

Deferred Update

Class	Description	Restart Work
T1	Finished before CP	None
T2	Started before CP; finished before failure	Redo forward from first log record
T3	Started after CP; finished before failure	Redo forward from first log record
T4	Started before CP; not yet finished	None
T5	Started after CP; not yet finished	None

REFERENCES

Database Systems: A Practical Approach to Design, Implementation, and Management, Thomas Connolly and Carolyn Begg, 5th Edition, 2010, Pearson.