UTM
UNIVERSITI TEKNOLOGI MALAYSIA

SECD2523 DATABASE

TOPIC 7 | TRANSACTION MANAGEMENT

Content adapted from Connolly, T., Begg, C., 2015. Database Systems: A Practical Approach to Design, Implementation, and Management, Global Edition. Pearson Education.

LECTURE LEARNING OUTCOME

By the end of this lecture, students should be able to:

01	**Define a transaction and explain the importance of the ACID properties (Atomicity, Consistency, Isolation, Durability) in maintaining database integrity.**
02	**Describe the role of concurrency control and identify common issues like lost updates and inconsistent analysis in multi-user database systems.**
03	**Explain the concept of serializability and differentiate between serial and nonserial transaction schedules.**
04	**Analyze techniques for concurrency control, including locking and timestamping, and how they help manage simultaneous transactions.**

Transaction Support

Transaction

- Action, or series of actions, carried out by user or application, which reads or updates contents of database.
  - Logical unit of work on the database.
  - Application program is series of transactions with non-database processing in between.
  - Transforms database from one consistent state to another, although consistency may be violated during transaction.

Transaction Support

- Can have one of two outcomes:

Success

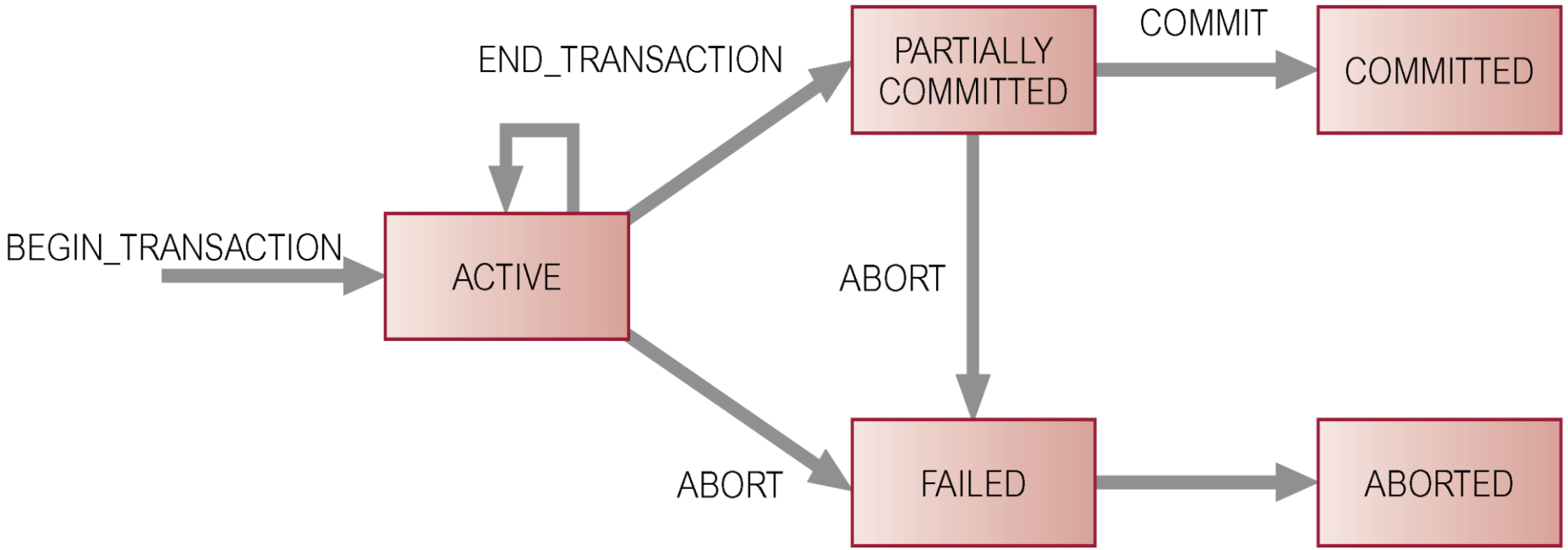
transaction **commits** and database reaches a new consistent state.

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 Transactions

Four basic **(ACID)** properties that define a transaction are:

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

ACID Properties of Transactions: A Simple Explanation

The ACID properties ensure that a database transaction is processed reliably.
Let's break it down with a relatable example from daily life: **withdrawing money from an ATM**.

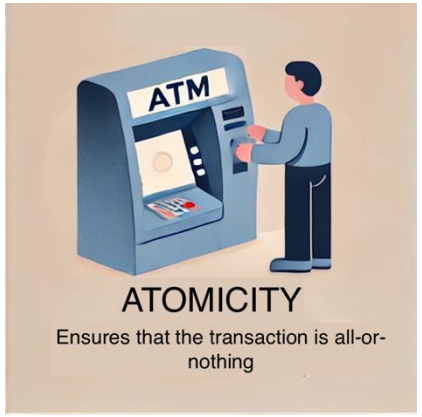
Imagine you have RM500 in your account and you want to withdraw RM100 from your bank account via an ATM:

Success:

- The ATM dispenses $100, and the amount is deducted from your account.

Failure:

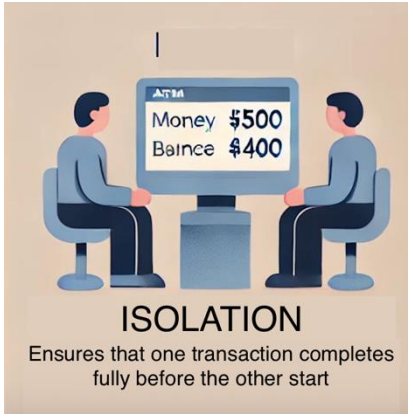
- If the ATM fails mid-way (e.g., power outage), neither the money is dispensed nor is your account debited.

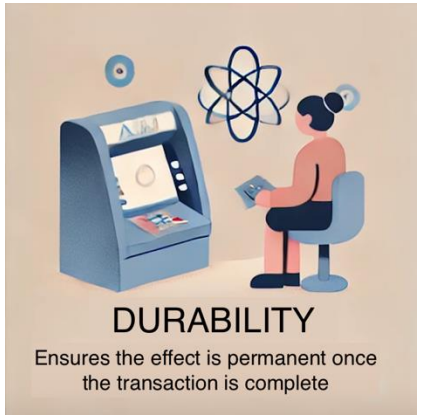


After withdrawing $100, your balance should be RM400 - not RM399 or RM401.

If in a situation where two persons try to withdraw RM300 from the same account at the same time:

- Without isolation, both of you might be able to withdraw RM300 each, even if the account balance is insufficient.
- With isolation, one transaction will go first, leaving RM200 in the account. The second transaction will then fail due to insufficient funds.





Once the transaction is completed, its effect is permanent—even if there's a system crash afterward.

- After withdrawing RM100, your updated balance of RM400 will still be there even if the ATM system restarts.

EXERCISE #1

For each scenario, identify which ACID property is violated and explain why.

Scenario 1:

A customer withdraws $100 from an ATM, but due to a system crash, the money is deducted from their account but not dispensed.

Property Violated:

Explanation:

Scenario 2:

While generating a sales report, new sales transactions are being added to the database. The report shows inconsistent totals.

Property Violated:

Explanation:

EXERCISE #1

For each scenario, identify which ACID property is violated and explain why.

Scenario 3:

A transaction updates the price of a product, but due to a server crash, the updated price is lost.

Property Violated:

Explanation:

Concurrency Control

- Process of **managing simultaneous operations** on the database **without having them interfere with one another**.
- Prevents interference when two or more users are accessing database simultaneously and at least one is updating data.
- Although two transactions may be correct in themselves, interleaving of operations may produce an incorrect result.
- Three examples of potential problems caused by concurrency:

Lost update problem.

Uncommitted dependency problem.

Inconsistent analysis problem.

Concurrency Control

Lost Update

Overwrites updates made by another transaction.

Example: Two clerks updating inventory, and one's changes are lost.

Uncommitted Dependency

Reads uncommitted changes that might be rolled back.

Example: Reading a temporary bank balance that is later reverted.

Inconsistent Analysis

Reads inconsistent data due to concurrent updates.

Example: Generating a report while sales data is being added, leading to incomplete totals.

Lost Update Problem

- Successfully completed update is overridden by another user.
- The ideal situation would be:

Transaction 1 (T_1) withdrawing RM10 from an account with bal_x (RM100).

$$T_1 = 100 - 10 = 90$$

Transaction 2 (T_2) depositing RM100 into same account.

Serially, final balance would be 190.

$$T_2 = 90 + 100 = 190$$

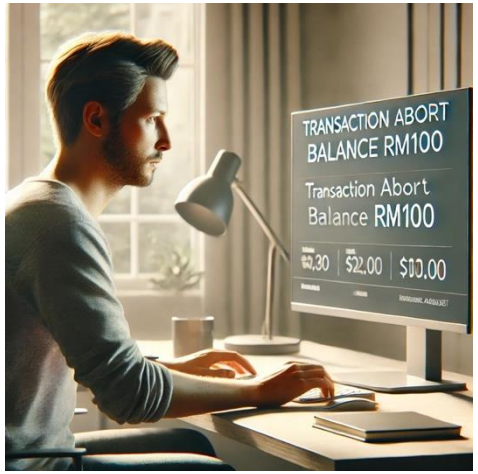
Lost Update Problem

- Lost updates happen when two transactions update the same data at the same time, and one update overwrites the other without considering it.
- This typically occurs because there is no mechanism in place to coordinate or lock the data during the updates.

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

Uncommitted Dependency Problem

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



Transaction 4 (T_4) updates bal_x to RM200 but it aborts, so bal_x should be back at original value of RM100.

$$bal_x = 100$$
$$T_4 = 100 + 100$$
$$= 200$$

⚠ T_4 ABORT

$$bal_x = 100$$

Another transaction (T_3) has read new value of bal_x (RM200) and uses value as basis of RM10 reduction, giving a new balance of RM190, instead of RM90.

CORRECT

$$bal_x = 100$$
$$T_4 = 100 - 10$$
$$= 90$$

WRONG

$$bal_x = 200$$
$$T_4 = 200 - 10$$
$$= 190$$

Still reads RM200

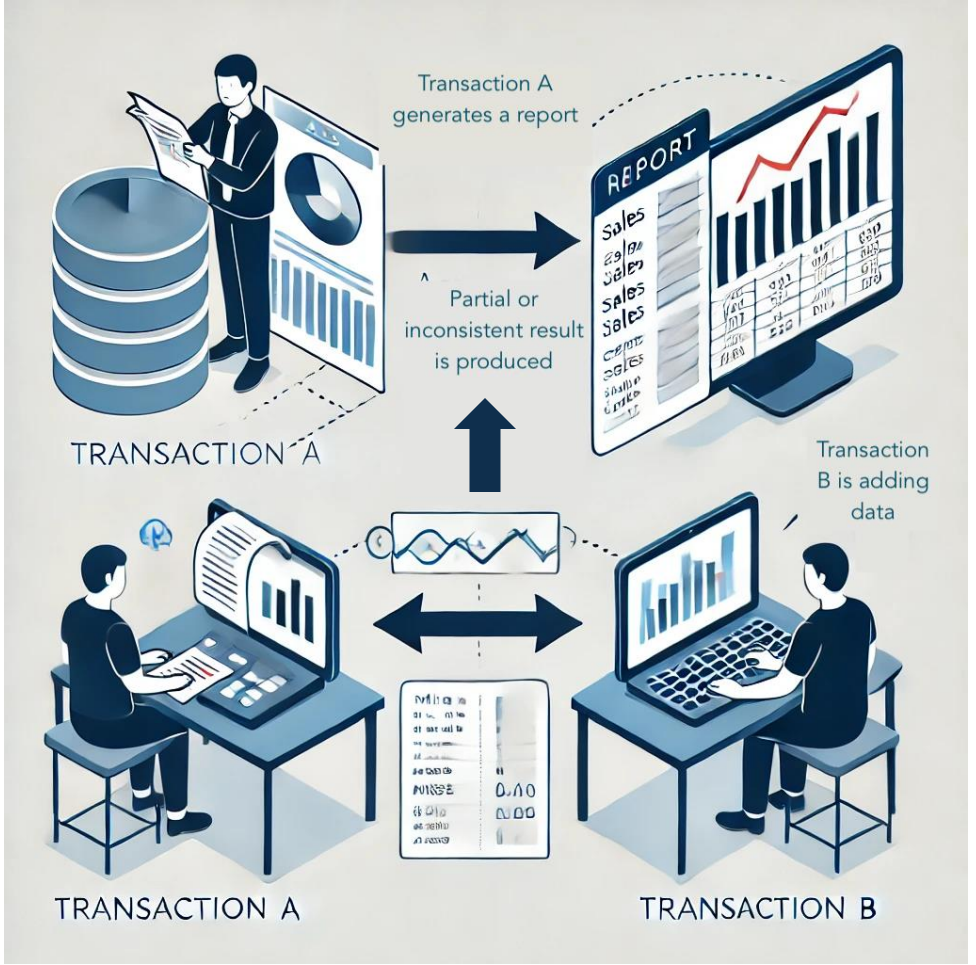
Uncommitted Dependency Problem

- Uncommitted dependency happens when one transaction uses data that another transaction has temporarily changed but not yet finalized (committed).
- If the first transaction is later canceled (rolled back), the data used by the second transaction becomes incorrect.

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

Inconsistent Analysis Problem

- Occurs when transaction reads several values but second transaction updates some of them during execution of first.
- Transaction A generates a report while Transaction B is adding data, leading to partial or inconsistent results.
- This happens because Transaction A reads some of the new data added by Transaction B but misses other updates that occur afterward.
- As a result, the final report created by Transaction A reflects an incomplete and inaccurate picture of the dataset.



Inconsistent Analysis Problem

Transaction 6 (T_6) is getting the total of account x (bal_x = RM100), account y (bal_y = RM50), and account z (bal_z = RM25).

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

The problem can be avoided by preventing T_6 from reading bal_x and bal_z until after T_5 completed updates.

EXERCISE #2

For each scenario, identify the concurrency problem and explain the issue.

Scenario 1:

A manager generates a report while another transaction is adding new sales data. The report includes some of the new data but misses others.

Problem:

Explanation:

Scenario 2:

Two employees update the stock of a product simultaneously: one adds 50 units, and the other removes 20 units. The database shows only one of these changes.

Problem:

Explanation:

EXERCISE #2

Scenario 3:

Transaction 1 temporarily updates a bank balance to RM300 but doesn't commit.
Transaction 2 reads this value to calculate interest. Later, Transaction 1 rolls back to RM500.

Problem: Uncommitted Dependency

Explanation: Transaction 2 used uncommitted data that was later invalidated.

EXERCISE #3

You are managing a shared inventory system. Two employees, Alice and Bob, are updating the stock of a product (Product X) simultaneously:

Initial Stock of Product X: 100 units.

Alice's Task (Transaction T1): Add 20 units to the stock.

Bob's Task (Transaction T2): Subtract 10 units from the stock.

Both transactions are executed at the same time, and neither is aware of the other's changes. The following operations take place:

T1 (Alice): Reads the stock as **100**.

T2 (Bob): Reads the stock as **100**.

T1 (Alice): Writes the stock as **120** (100 + 20).

T2 (Bob): Writes the stock as **90** (100 - 10).

QUESTIONS

1. What is the final stock of Product X in the system after both transactions?
2. Why does this problem occur?

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.
- Serializability has two main concepts:

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

Serializability

Aspect	Schedule	Serial Schedule
Definition	A sequence of operations (read, write, commit, etc.) from one or more transactions, interleaved or not.	A specific type of schedule where transactions are executed one after the other without interleaving.
Execution	Allows operations from multiple transactions to be interleaved.	Executes each transaction completely before starting the next one.
Concurrency	Enables concurrent execution to improve performance.	Does not allow concurrency; transactions are processed sequentially.
Complexity	More complex to analyze for conflicts and consistency due to interleaving.	Simple to analyze since no interleaving occurs.
Consistency	May or may not maintain database consistency (depending on serializability and isolation levels).	Always maintains consistency since transactions don't interfere with one another.
Example	T1 and T2 are interleaved: - T1 Reads Balance. - T2 Reads Balance. - T1 Updates Balance. - T2 Updates Balance.	T1 and T2 execute without overlap: - T1 Reads Balance. - T1 Updates Balance. - T1 Writes Balance. (T1 finish) - T2 Reads Balance.

Serializability

S_1, S_2, and S_3 shows operations from two concurrent transactions T_7 and T_8, where the write operation on bal_x in T_8 does not conflict with the subsequent read operation on bal_y in T_7.

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) Schedule S_1

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) Schedule S_2

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) Schedule S_3

S_1 is a non-serial schedules

S_2 is a non-serial schedules

S_3 is a serial schedules

Serializability

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 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**.

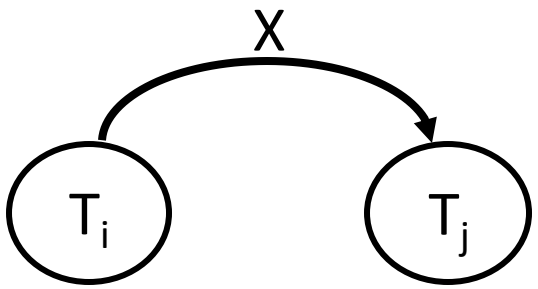
Serializability

Scenario	Description	Conflict?	Order Important?
Transaction 1 (T_1) READS data A. Transaction 2 (T_2) READS data A.	Two transactions read the same data item.	NO	NO
Transaction 1 (T_1) READS or WRITE data A. Transaction 2 (T_2) READS or WRITE data B.	Two transactions read or write different data items.	NO	NO
Transaction 1 (T_1) READS data A. Transaction 2 (T_2) WRITES data A.	One transaction writes to a data item while another reads or writes the same data item.	YES	YES

Conflict Serializability

- Conflict serializable schedule orders any conflicting operations in same way as some serial execution.
- Under **constrained write rule** (transaction updates data item based on its old value, which is first read), use **precedence graph** to test for serializability.
- For a schedule S, a precedence graph is a directed graph G = (N, E) that consists of a set of nodes N and a set of directed edges E, which is constructed as follows:

1. Create a node for each transaction.
2. Create a directed edge $T_i \rightarrow T_j$ (X), if:
   - T_j reads the value of an item written by T_i.
   - T_j writes a value into an item after it has been read by T_i.
   - T_j writes a value into an item after it has been written by T_i.



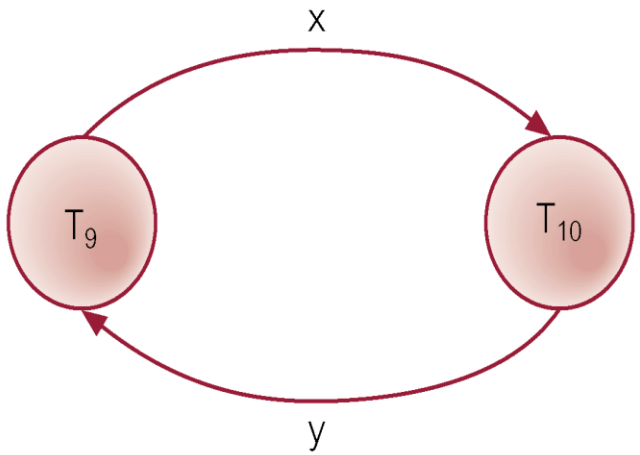
Non-Conflict Serializable Schedule

- A **Non-Conflict Serializable Schedule** is a schedule of transactions that does **not have any conflicting operations**.
- Because there are no conflicts, the schedule is automatically **serializable**, meaning its result is equivalent to a serial schedule, where transactions execute one after the other.
- A schedule is non-conflict serializable if:
  1. All operations in the transactions involve different data items, or
  2. The operations on the same data items are read-only (no writes are performed).

Example: Non-Conflict Serializable Schedule

- T_9 is transferring RM100 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	



Precedence graph has a cycle and so is not serializable.

EXERCISE #4

For each of the following schedules, do the transaction table and draw the equivalent precedence graph:

1. read(T_1, bal_x), read(T_2, bal_x), write(T_1, bal_x), write(T_2, bal_x), commit(T_1), commit(T_2)
2. read(T_1, bal_x), read(T_2, bal_y), write(T_3, bal_x), read(T_2, bal_x), read(T_1, bal_y), commit(T_1), commit(T_2), commit(T_3).
3. read(T_1, bal_x), write(T_2, bal_x), write(T_1, bal_x), abort(T_2), commit(T_1)
4. write(T_1, bal_x), read(T_2, bal_x), write(T_1, bal_x), commit(T_2), abort(T_1)
5. read(T_1, bal_x), write(T_2, bal_x), write(T_1, bal_x), read(T_3, bal_x), commit(T_1), commit(T_2), commit(T_3)

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 so prevent incorrect updates.
- Most widely used approach to ensure serializability.
- Generally, a transaction must claim

Shared lock	If a transaction has a shared lock on a data item, it can read the item but not update it.
Exclusive lock	If a transaction has an exclusive lock on a data item, it can both read and update the item.

- Lock prevents another transaction from modifying item or even reading it, in the case of a write lock.

Locking - Basic Rules

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

Problems in Locking

STARVATION	A transaction that cannot perform any operations while other transactions continue to execute as they wish.
	This happens when a specific transaction is perpetually delayed or ignored due to the system prioritizing other transactions.
	For instance, in priority-based scheduling, a high-priority transaction may keep pre-empting a lower-priority transaction, leading to indefinite delays for the latter.
LIVELOCK	No transactions are able to complete their tasks, and they cannot acquire any new locks.
	In a livelock scenario, transactions keep releasing and re-requesting locks in response to each other's actions, but none make progress.
	This is similar to two people stepping aside repeatedly to let each other pass through a doorway, resulting in neither proceeding.
DEADLOCK	Two transactions are waiting for locks held by each other and cannot proceed with their operations.
	Deadlock occurs when there is a cyclic dependency between transactions, where each transaction holds a lock that the other requires.
	As a result, both transactions are stuck indefinitely unless an external mechanism intervenes, such as a deadlock detection and resolution system.

Problems in Locking: STARVATION

- Shared locks (S-locks) often take precedence over exclusive locks (X-locks) in locking protocols, which can lead to **starvation** for transactions requesting X-locks if many S-lock requests keep arriving.
- This is because the X-lock requester (T2) has to wait for all S-locks to be released before it can proceed.

	T1	T2	T3	T4
t1	begin transaction			
t2	lock (A)	begin transaction		
t3		request x-lock (A) - queue	begin transaction	
t4		.	s-lock (A)	
t5	unlock (A)	.		begin transaction
t6	commit	.		request s-lock (A)
t7	.	.	unlock (A)	
t8	.	.		s-lock (A)
t9	.	.	request s-lock (A)	
t10	.	.		unlock (A)
t11	.	.	s-lock (A)	.
t12	.	.	.	.
t13	.	.	.	.
t14	.	.	.	.

T2 cannot start until all s-lock is released, resulting in STARVATION

Problems in Locking: LIVELOCK

- Both transactions (T1 and T2) are trying to avoid blocking the other by releasing their locks, but they end up reacquiring the same locks they just released.
- Thus, both transactions are stuck in a cycle of “release and retry,” where neither transaction is able to acquire both A and B simultaneously, which is required to complete their operations.

	T1	T2
t1	begin transaction	
t2	x-lock (A)	begin transaction
t3		x-lock (B)
t4	request x-lock (B)	
t5	unlock (A)	
t6	x-lock (A)	
t7		request x-lock (A)
t8		unlock (B)
t9		x-lock (B)
t10	request x-lock (B)	.
t11	.	.
t12	.	.
t13	.	.
t14	.	.

Problems in Locking: DEADLOCK

- Neither T1 nor T2 can proceed because each is waiting for the other to release a resource.
- This results in a **deadlock**, where both transactions are stuck indefinitely.

	T1	T2
t1	begin transaction	
t2	x-lock (A)	begin transaction
t3		x-lock (B)
t4	request x-lock (B) - queue	
t5		request x-lock (A) - queue
t6	.	.
t7	.	.
t8	.	.
t9	.	.
t10	.	.

Problems in Locking

Problem with the use of locks:
Does not fully guarantee the isolation of transactions.

Solution:
Use the two-phase locking (2PL) protocol, which emphasizes the positioning of lock and unlock operations in each 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.
- The use of 2PL can resolve the concurrency issues discusses before:
  1. lost updates
  2. uncommitted dependencies
  3. inconsistent analysis.

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

- By enforcing the **2PL protocol**, T1 is forced to wait until T2 finishes and releases its locks.
- This ensures that updates from both transactions are preserved, preventing the **lost update problem** and maintaining correct transaction results.

1. T_1 **Begins (t_2)**
   - T_1 starts its transaction and tries to acquire a write lock on bal_x at t_2.
   - However, T_2 has already acquired a write lock on bal_x at t_2.
   - Now T_1 has to WAIT for T_2 to finish the transaction
2. T_2 **Operations (t_3–t_6)**:
   - T_2 reads bal_x , modifies it ($bal_x = bal_x + 100$), and writes the new value (200) back.
   - Once T_2 completes its operation at t_6, it releases the lock (commit/unlock).
3. T_1 **Resumes (t_7–t_{10})**:
   - After T_2 releases the lock, T_1 acquires the write lock on bal_x.
   - T_1 performs its read ($bal_x = 200$), modifies it ($bal_x = bal_x - 10$), and writes the new value (190) back.
   - T_1 then commits and unlocks bal_x at t10.

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

1. **T_4 Operations (t_4-t_5):**
   - T_4 begins its transaction and requests a write lock on bal_x.
   - It acquires the lock because no other transaction is holding it at this point.
   - T_4 reads the value of bal_x (which is 100), adds 100 to it (resulting in 200), and writes the new value back to the database.
2. **T_4 Rollback (t_6):**
   - T_4 performs a rollback operation, meaning it aborts its transaction and restores the original value of bal_x to 100.
   - It then unlocks bal_x, allowing T_3 to proceed.

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

The use of locks ensures that only one transaction can write to bal_x at a time, avoiding concurrency issues as in this problem.

3. **T_3 waits (t_4-t_6):**
   - T_3 begins waiting because T_4 has also requested the same resource (bal_x). T_3 cannot proceed until T_4 completes its operations or releases its lock.

4. **T_3 Operations (t_7-t_{10}):**
   - Once T_4 releases the write lock on bal_x, T_3 proceeds to read the value of bal_x (now reverted to 100 after T_4's rollback).
   - T_3 deducts 10 from the value of bal_x (making it 90) and writes this new value to the database.
   - Finally, T_3 commits its transaction and releases the lock.

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

This ensures that no other transaction can modify the resources while other transaction is reading/writing them, maintaining consistency during the calculation.

1. **T_5 Updates (t_2-t_{11})**
   - T_5 begins by acquiring a **write lock** on bal_x and bal_z.
   - It modifies bal_x by subtracting 10 and bal_z by adding 10.
   - After updating these values, T_5 commits and releases its locks.
2. **T_6 Waits (t_4-t_{11})**
   - T_6 starts by acquiring a **read lock** on bal_x but is forced to wait because T_5 is holding a **write lock** on the same resource.
3. **T_6 Reads (t_{12}-t_{20})**
   - Once T_5 commits and releases its locks, T_6 reads bal_x, adds its value to sum, and continues to acquire locks and read bal_y and balz.
   - Finally, T_6 calculates the total sum using the consistent values of all three resources.

EXERCISE #5

1. Apply locking techniques to the transaction table below:

	T1	T2	T3
t_1		begin transaction	
t_2		read item (z)	
t_3		read item (y)	
t_4		write item (y)	begin transaction
t_5			read item (y)
t_6	begin transaction		read item (z)
t_7	read item (x)		
t_8	write item (x)		
t_9			write item (y)
t_{10}			write item (z)
t_{11}		read item (x)	commit
t_{12}	read item (y)		
t_{13}	write item (y)		
t_{14}	commit	write item (x)	
t_{15}		commit	

EXERCISE #5

2. Study the transaction table for T4 and T5.
   Let's say:
   - T4 wants to transfer RM 100 from one account with balance x (bal_x) to another account with balance y (bal_y).
   - T5 wants to increase both balances by 10%.

   Assume the initial values are x = 50 and y = 100.

   i. Is the balance for both account correct?
   ii. What is the concurrency problem that happens here?
   iii. Apply the 2PL solution to this problem. What is the correct values for bal_x and bal_y?

	T4	T5	bal_x	bal_y
t_1	begin transaction		50	100
t_2	read (bal_x)		50	100
t_3	$bal_x = bal_x + 100$		50	100
t_4	write (bal_x)	begin transaction	150	100
t_5		read (bal_x)	150	100
t_6		$bal_x = bal_x * 1.1$	150	100
t_7		write (bal_x)	165	100
t_8		read (bal_y)	165	100
t_9		$bal_y = bal_y * 1.1$	165	100
t_{10}		write (bal_y)	165	110
t_{11}	read (bal_y)	commit	165	110
t_{12}	$bal_y = bal_y - 100$		165	110
t_{13}	write (bal_y)		165	10
t_{14}	commit		165	10

Cascading Rollback

- If **every** transaction in a schedule follows 2 P L, schedule is serializable.
- However, problems can occur with interpretation of when locks can be released.
- A **cascading rollback** occurs when a single transaction failure causes multiple other dependent transactions to roll back as well.
- This happens because the dependent transactions have accessed uncommitted changes made by the failed transaction - potentially affecting a chain of transactions, which makes the system inefficient and error-prone.

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

How cascading rollback happens?

1. Transaction T_{14} obtains an exclusive lock on bal_x and then updates it using bal_y, which has been obtained with a shared lock, and writes the value of bal_x back to the database before releasing the lock on bal_x.
2. TransactionT_{15} then obtains an exclusive lock on bal_x, reads the value of bal_x from the database, updates it, and writes the new value back to the database before releasing the lock.
3. Finally, T_{16} share locks bal_x and reads it from the database.
4. By now, T_{14} has failed and has been rolled back. However, because T_{15} is dependent on T_{14} (it has read an item that has been updated by T_{14}), T_{15} must also be rolled back.
5. Similarly, T_{16} is dependent on T_{15}, so it too must be rolled back.

Timestamping

- The use of locks, combined with the two-phase locking protocol, guarantees serializability of schedules.
- The order of transactions in the equivalent serial schedule is based on the order in which the transactions lock the items they require.
- If a transaction needs an item that is already locked, it may be forced to wait until the item is released.
- A different approach that also guarantees serializability uses **transaction timestamps** to order transaction execution for an equivalent serial schedule.
- Transactions ordered globally so that older transactions, transactions with **smaller** timestamps, get priority in the event of conflict - conflict is resolved by rolling back and restarting transaction.
- No locks so no deadlock.

Timestamping

WHAT IS TIMESTAMP?

- A unique identifier created by DBMS that indicates relative starting time of a transaction.
- Can be generated by using system clock at time transaction started, or by incrementing a logical counter every time a new transaction starts.
- Read/write proceeds only if **last update on that data item** was carried out by an older transaction.
- Otherwise, transaction requesting read/write is restarted and given a new timestamp.
- Also timestamps for data items:
  - **read-timestamp** - timestamp of last transaction to read item;
  - **write-timestamp** - timestamp of last transaction to write item.

Timestamping Read (x)

Consider a transaction T with timestamp $t_s(T)$:

1. Transaction T asks to **read** an item (x) that has already been updated by a younger (later) transaction - **$t_s(T) <$ write_timestamp(x)**.
   - This means that an **earlier transaction is trying to read a value** of an item that **has been updated by a later transaction**.
   - The earlier transaction is too late to read the previous outdated value, and any other values it has acquired are likely to be inconsistent with the updated value of the data item.
2. In this situation, transaction T must be aborted and restarted with a new (later) timestamp.

Timestamping Write (x)

Consider a transaction T with timestamp $t_s(T)$:

WRITE (x) < READ (x)

1. Transaction T asks to **write** an item (x) whose value has already been **read** by a younger transaction - **$t_s(T) <$ read_timestamp(x)**.
   - This means that a later transaction is already using the current value of the item, and it would be an error to update it now.
   - This occurs when a transaction is late in doing a write and a younger transaction has already read the old value or written a new one.
2. In this case, the only solution is to roll back transaction T and restart it using a later timestamp.

WRITE (x) < WRITE (x)

1. Transaction T asks to **write** an item (x) whose value has already been **written** by a younger transaction - **$t_s(T) <$ write_timestamp(x)**.
   - This means that transaction T is attempting to write an obsolete value of data item x.
2. Transaction T should be rolled back and restarted using a later timestamp.
3. Otherwise, the write operation can proceed. We set write_timestamp(x) = $t_s(T)$.

Example: Basic Timestamp Ordering

Time	Op	T_{19}	T_{20}	T_{21}
t_1		begin_transaction		
t_2	read(bal_x)	read(bal_x)		
t_3	$bal_x = bal_x + 10$	$bal_x = bal_x + 10$		
t_4	write(bal_x)	write(bal_x)	begin_transaction	
t_5	read(bal_y)		read(bal_y)	
t_6	$bal_y = bal_y + 20$		$bal_y = bal_y + 20$	begin_transaction
t_7	read(bal_y)			read(bal_y)
t_8	write(bal_y)		write(bal_y)	
t_9	$bal_y = bal_y + 30$			$bal_y = bal_y + 30$
t_{10}	write(bal_y)			write(bal_y)
t_{11}	$bal_z = 100$			$bal_z = 100$
t_{12}	write(bal_z)			write(bal_z)
t_{13}	$bal_z = 50$	$bal_z = 50$		commit
t_{14}	write(bal_z)	write(bal_z)	begin_transaction	
t_{15}	read(bal_y)	commit	read(bal_y)	
t_{16}	$bal_y = bal_y + 20$		$bal_y = bal_y + 20$	
t_{17}	write(bal_y)		write(bal_y)	
t_{18}			commit	

Three transactions are executing concurrently:

1. Transaction T_{19} has a timestamp of $t_s(T_{19})$, T_{20} has a timestamp of $t_s(T_{20})$, and T_{21} has a timestamp of $t_s(T_{21})$.
2. At time t_8, the write by transaction T_{20} violates the first write rule and so T_{20} is aborted and restarted at time t_{14}.
3. Also, at time t_{14}, the write by transaction T_{19} can safely be ignored using the ignore obsolete write rule, as it would have been overwritten by the write of transaction T_{21} at time t_{12}.
4. After all transactions are complete (assuming initial bal_x and bal_y= 100):
   - bal_x: Updated by T_{19} to 110 (no further modifications).
   - bal_y: Updated by T_{21} to 150 (last modification).
   - bal_z: Updated by T_{21} to 100 (last modification).

EXERCISE #6

Refer to your answer in Exercise #5, Question 2 (iii):

1. Apply timestamping to your 2PL solution.
2. Explain your answer from (1)

Granularity of Data Items

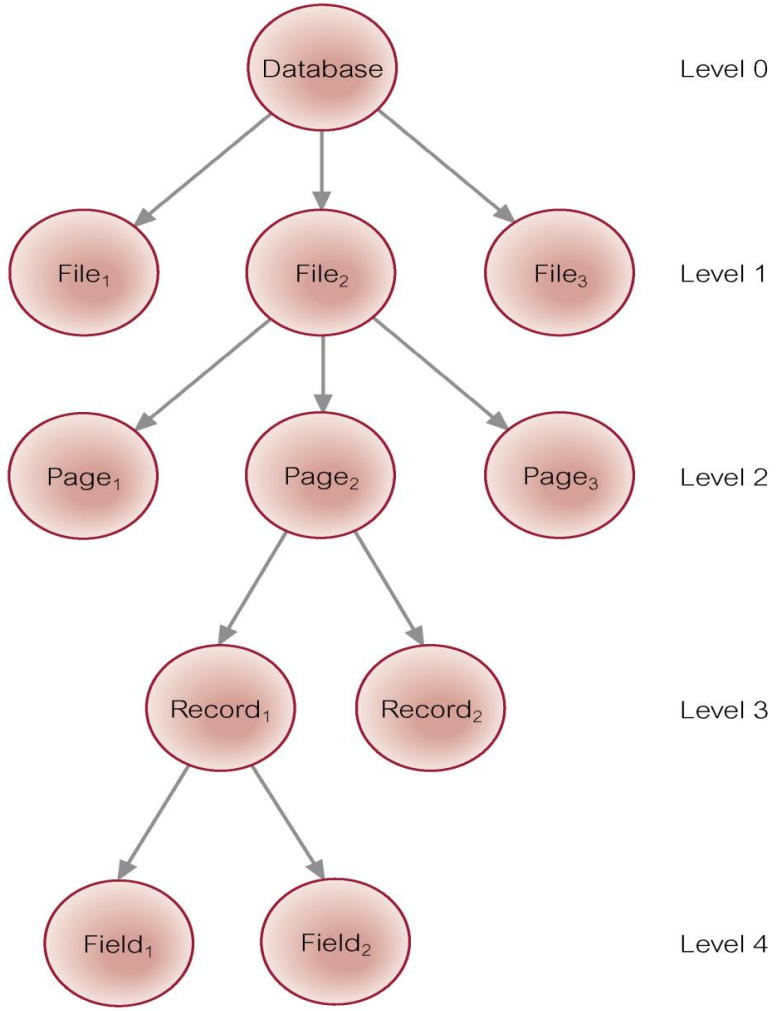
- All the concurrency control protocols that we have discussed assume that the database consists of a number of “data items,” without explicitly defining the term.
- Typically, a data item is chosen to be one of the following, ranging from coarse to fine, where fine granularity refers to small item sizes and coarse granularity refers to large item sizes:
  - The entire database.
  - A file.
  - A page (or area or database spaced).
  - A record.
  - A field value of a record.

Granularity of Data Items

- The larger the size of the data item, the lower the concurrency control.
- Conversely, the smaller the size of the data item, the higher the concurrency control because more locking information needs to be stored.
- Tradeoff of the granularity of data item:
  - Coarser: the lower the degree of concurrency;
  - Finer: more locking information that is needed to be stored.
- Best item size depends on the types of transactions.
  - If a transaction accesses specific fields from a single record, it is better to choose the granularity of the data item size as a **record**.
  - If a transaction uses many records from the same file, it is better to choose the granularity of the data item size as a **file**.

Hierarchy of Granularity

- Could represent granularity of locks in a hierarchical structure.
- Root node represents entire database, level 1s represent files, etc.
- When node is locked, all its descendants are also locked.
- The diagram shows the structure of granularity hierarchy.
  - If one of the nodes is locked, then the nodes below it that inherit it will automatically be locked as well.
- DBMS should check hierarchical path before granting lock.



Innovating Solutions
Menginovasi Penyelesaian