Transaction Management

Concurrency control

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**, **deadlock prevention** and **detection**, and **time-stamping** concept.

Transaction

- **Action**, or **series of actions**, carried out by user or application, which reads or updates contents of database.

A transaction is a **logical unit of work** on the database. May consists of:
- Entire program
- A part of a program
- A single statement (INSERT or UPDATE)
- May involve any number of operations on the database

A transaction must be entirely **completed** or **aborted**
- ✓ no **intermediate state** are acceptable
- ✓ can involve any number of reads of writes to a database.

Transaction

- Each operation on database can be considered as transactions.
- Application program is series of transactions with non-database processing in between.
- During transactions, database is transformed from consistent state to another.

Example Transaction

- Staff (<u>staffNo</u>, fname, lname, position, sex, DOB, salary, branchNo)
- PropertyForRent (<u>propertyNo</u>, street, city, postcode, type, rooms, rent, ownerNo, staffNo, branchNo)

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

- db operation: read & wire
- Non-db operation: salary = salary * 1.1

(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)

a) To update the salary of a particular staff
b) To delete the member of staff with given staff no

Transaction Example

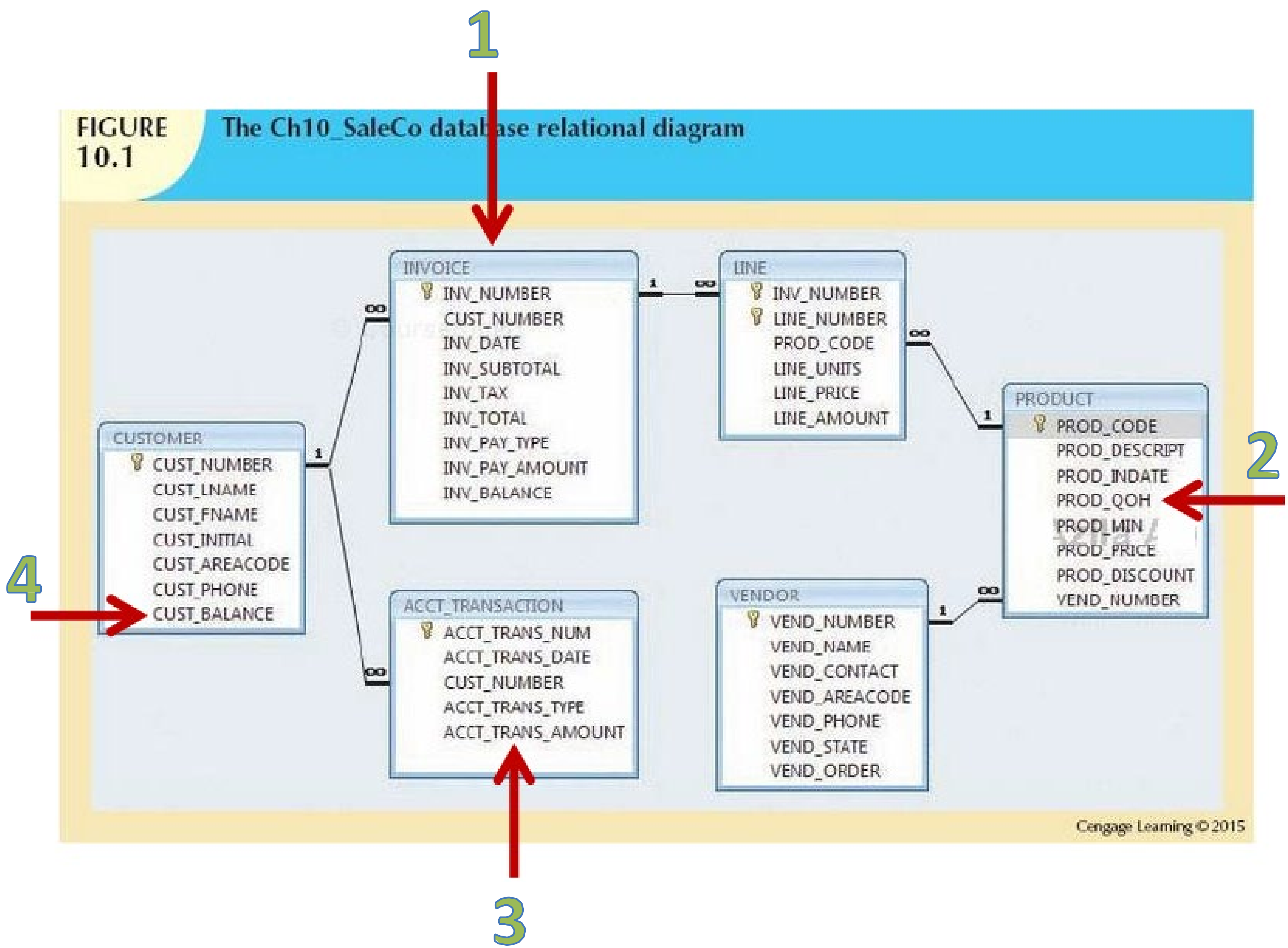
Scenario:

Sell product to a customer

Customer may charge purchase to his/her account

Action:

1. Write a new customer invoice
2. Reduce the quantity on hand in the product's inventory
3. Update the account transactions
4. Update the customer balance



1
2
3
4
FIGURE 10.1
The Ch10_SaleCo database relational diagram
INVOICE
INV_NUMBER
CUST_NUMBER
INV_DATE
INV_SUBTOTAL
INV_TAX
INV_TOTAL
INV_PAY_TYPE
INV_PAY_AMOUNT
INV_BALANCE
LINE
INV_NUMBER
LINE_NUMBER
PROD_CODE
LINE_UNITS
LINE_PRICE
LINE_AMOUNT
CUSTOMER
CUST_NUMBER
CUST_LNAME
CUST_FNAME
CUST_INITIAL
CUST_AREACODE
CUST_PHONE
CUST_BALANCE
PRODUCT
PROD_CODE
PROD_DESCRIPT
PROD_INDATE
PROD_QOH
PROD_MIN
PROD_PRICE
PROD_DISCOUNT
VEND_NUMBER
ACCT_TRANSACTION
ACCT_TRANS_NUM
ACCT_TRANS_DATE
CUST_NUMBER
ACCT_TRANS_TYPE
ACCT_TRANS_AMOUNT
VENDOR
VEND_NUMBER
VEND_NAME
VEND_CONTACT
VEND_AREACODE
VEND_PHONE
VEND_STATE
VEND_ORDER
Cengage Learning © 2015

Evaluating Transaction Result

Examine CUSTOMER table to determine current balance for customer 10016

```
SELECT CUST_NUMBER, CUST_BALANCE
FROM CUSTOMER
WHERE CUST_NUMBER = 10016
```

- Query does not make any changes to database
- Because the transaction did not alter the database, the database remains in consistent state after the access

Airline Transaction Example

START TRANSACTION

Display greeting

Get reservation preferences from user

SELECT departure and return flight records

If reservation is acceptable, then

UPDATE seats remaining of departure flight record

UPDATE seats remaining of return flight record

INSERT reservation record

Print ticket if requested

End If

On Error: **ROLLBACK**

COMMIT

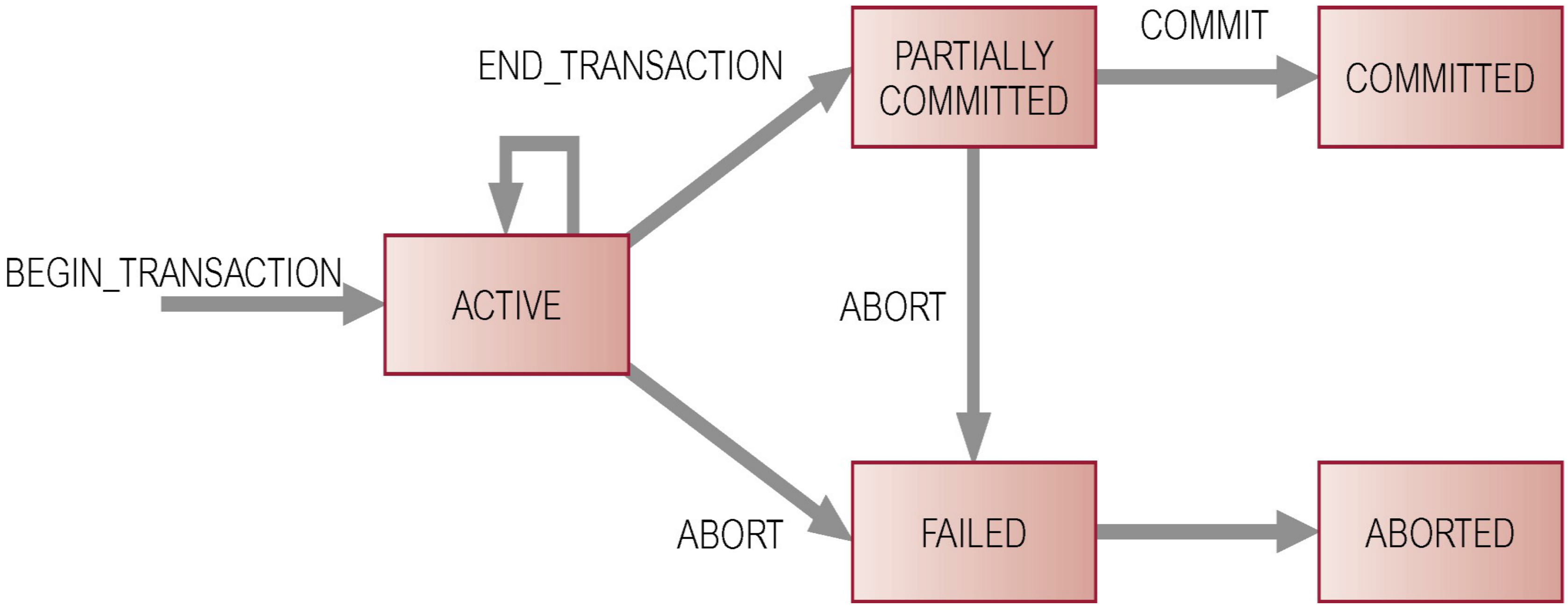
ATM Transaction Example

```
START TRANSACTION
    Display greeting
    Get account number, pin, type, and amount
    SELECT account number, type, and balance
    If balance is sufficient then
        UPDATE account by posting debit
        UPDATE account by posting credit
        INSERT history record
        Display message and dispense cash
        Print receipt if requested
    End If
    On Error: ROLLBACK
COMMIT
```

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



- PARTIALLY COMMITED: occurs after final statement has been executed.
  - ➢ If system fail (any data updated not safely recorded on secondary storage), the transaction would go to FAILED state & have to be aborted.
  - ➢ If the transaction successful, any updates can be recorded – transaction go to COMMITED state.
- FAILED: occurs if the transaction cannot be committed or aborted while in the ACTIVE state

Transaction Properties

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

Atomicity

- '**All or nothing**' property.
- All operations of a transaction must be **completed**, if not, the transaction is **aborted**

Consistency

- Must transform database from one **consistent state** to another.

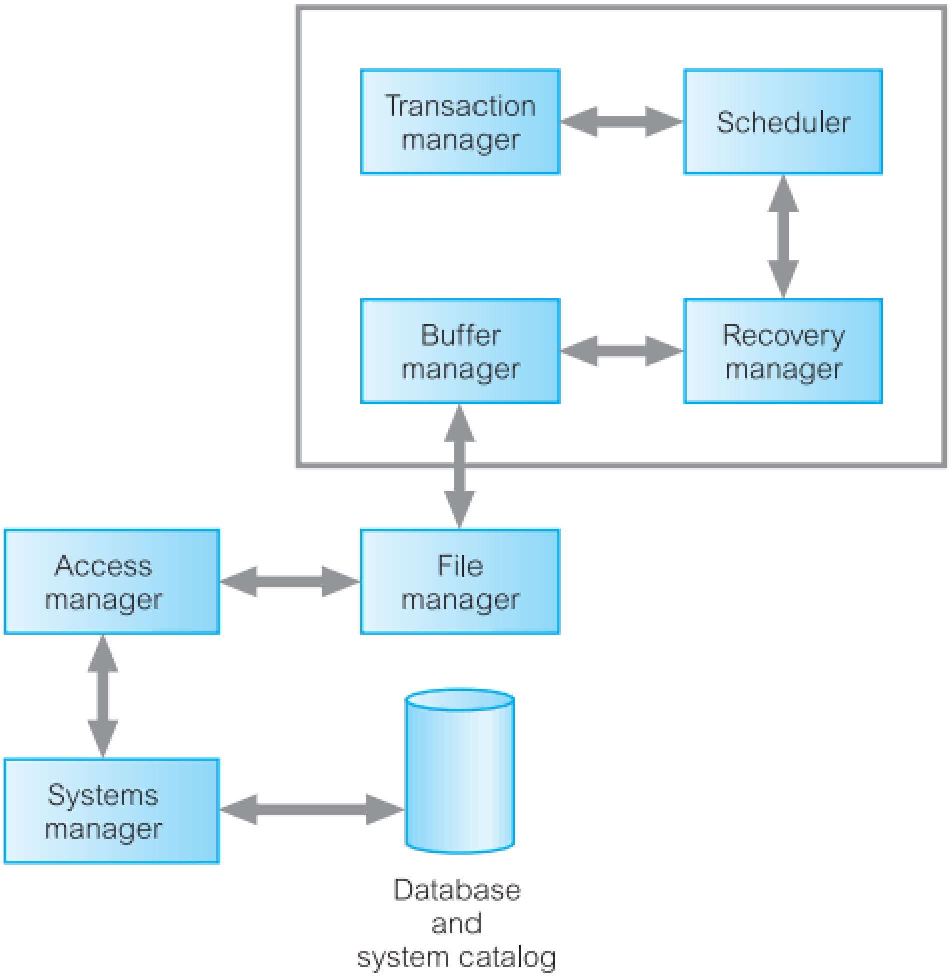
Isolation

- **Transactions** execute **independently** of **one another**. Partial effects of incomplete transactions should not be visible to other transactions.
- Data used during transaction cannot be used by second transaction until the first is completed.

Durability

- Effects of a committed transaction are **permanent** and must **not be lost because of later failure**.

DBMS Transaction Subsystem (Database architecture)



4 database modules that handles transactions, concurrency control, and recovery

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

- The process of **managing simultaneous operations** on the database without having them interfere with one another.

- What?

Coordination of the simultaneous transactions execution in a multiuser database system.

- Why? (Objective)

To ensure serializability of transactions in a multiuser database environment.

The Need for Concurrency Control

- The main objective of concurrency control is to allow many users perform different operations at the same time.
- It increases the throughput because of handling multiple transactions simultaneously.

- ✓ 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.

The Need for Concurrency Control

- If there is no concurrency control, the problems that might occur are:
  - ✓ **Lost Update Problem**
  - ✓ **Uncommitted Dependency Problem**
  - ✓ **Inconsistent Analysis Problem**

Lost Update Problem

- Occurs in two concurrent transactions when:
  - Same data element is updated
  - One of the updates is lost (successfully completed update is overridden by another user)

Time	T1	T2	Bal_x
t1		Begin_transaction	100
t2	Begin_transaction	Read(bal_x)	100
t3	read (bal_x)	$Bal_x = bal_x + 100$	100
t4	$Bal_x = Bal_x - 10$	Write (bal_x)	200
t5	Write (bal_x)	Commit	90
t6	Commit		90

Loss of T2's update is avoided by preventing T1 from reading bal_x until after update

Lost Update Problem

- Occurs when two concurrent transactions, T1 and T2, are updating the same data element and one of the update is lost (**overwritten** by the other transaction).

Lost update: a concurrency control problem in which one user's update overwrites another user's update

Lost Update Problem

- T1 withdrawing 10 from an account with balx, initially 100.
- T2 depositing 100 into same account.
- Serially, final balance would be 190.
- The addition of 100 units is “lost” during the process.

Uncommitted Dependency Problem

- Occurs when:
  - Two transactions are executed concurrently
  - First transaction is rolled back after the second transaction has already accessed uncommitted data

Time	T3	T4	Bal_x
t1		Begin_transaction	100
t2		read (bal_x)	100
t3		$Bal_x = bal_x + 100$	100
t4	Begin_transaction	Write (bal_x)	200
t5	read (bal_x)		200
t6	$Bal_x = Bal_x – 10$	Rollback	100
t7	Write (bal_x)		190
t8	Commit		190

T3 and T4, are executed concurrently and T4 is rolled back after T3 has already accessed the uncommitted data - thus violating the isolation property of transactions.

Uncommitted Dependency Problem

- T4 updates bal_x to 200 but it aborts, so bal_x should be back at original value of 100.
- T3 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.

Inconsistent Analysis Problem

- Occurs when a transaction accesses data before and after another transaction(s) finish working with such data.
  - e.g. T1 calculates some summary (aggregate) function over a set of data while another transaction (T2) was updating the same data.
  - Transaction might read some data **before** they are changed and other data **after** they are changed, thereby yielding inconsistent result

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

Problem is avoided by preventing T6 from reading bal_x and bal_z until after T5 completed updates

Inconsistent Analysis Problem

- Sometimes referred to as dirty read or unrepeatable read.
- T6 is totaling balances of account x (100), account y (50), and account z (25).
- In the meantime, T5 has transferred 10 from bal_x to bal_z, so T6 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** hence prevent the concurrency problem.

- ✓ Could run transactions serially, but this limits degree of concurrency or parallelism in system.
- ✓ The aim of multi-user DBMS is to maximize the degree of concurrency or parallelism in the system, so the transactions that can execute without interfering one another can run in parallel.
- ✓ Serializability identifies those executions of transactions that are guaranteed to ensure consistency.

Serializability

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
Nonserial Schedule	Schedule where operations from **set of concurrent transactions are interleaved**.

Serializability

- How do we ensure **database** are in a **consistent state**?
  - ✓ It is when we have a serial schedule/serial in nature

- **Serializability** is a concept that helps us to **check which schedules are serializable**.

- A **serial schedule is always a serializable schedule** → because in serial schedule, a transaction only starts when the other transaction finished execution.

- A serializable schedule is the one that always leaves the database in consistent state.

- If a set of transactions executes concurrently, we say that the (nonserial) schedule is correct if it **produces the same results** as some **serial execution**. This schedule is called **serializable**.

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** 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) Nonserial Schedule S1;

(b) Nonserial Schedule S2;

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

This type of serializability is known as conflict serializability.

A conflict serializable schedule orders any conflicting operations in same way as some serial execution.

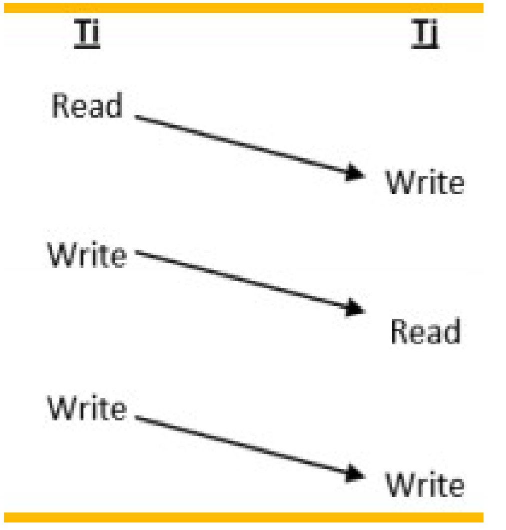
Conflict Serializable

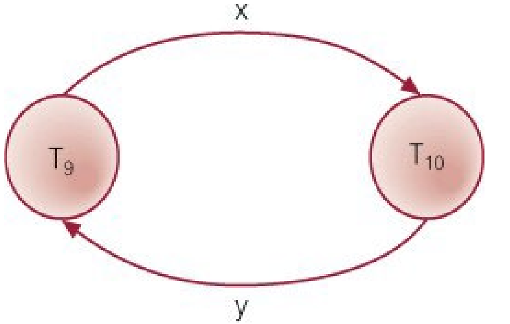
- A schedule is called conflict serializable if we can convert it into a serial schedule after swapping its non-conflicting operations.

Testing for Conflict Serializibility

- Use precedence graph to test for conflict serializability.

1. Create a node for each transaction (N)
2. Create directed edges (E)
   - ✓ a directed edge Ti → Tj, if Tj reads the value of an item written by Ti;
   - ✓ a directed edge Ti → Tj, if Tj writes a value into an item after it has been read by Ti.
   - ✓ a directed edge Ti → Tj, if Tj writes a value into an item after it has been written by Ti.





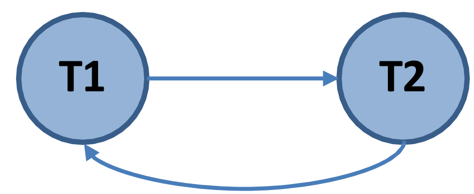
- If precedence graph contains cycle schedule is not conflict serializable.

Serializability

S1	
T1	T2
Read A	
Write A	
	Read A
	Write A

No cycle ➔ **serial graph**

S2	
T1	T2
Read A	
	Read A
Write A	
	Write A



Has cycle ➔ **parallel (nonserial) schedule**

Can we change from parallel to serial?

Why parallel is not ok? ➔ cannot ensure the database consistent state

Example on 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.

Testing for Conflict Serializability

https://youtu.be/odik_Bbg5Lk

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.

Locking

- If transaction has **shared lock (S)** on item, it **can read but not update** it.
- 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
- **Lock manager**: Responsible for assigning and policing the locks used by the transactions

Locking Conflicts

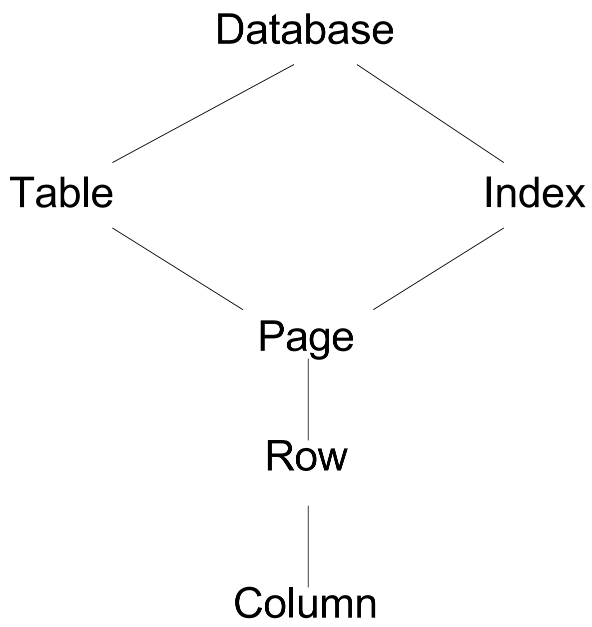
	USER 2 REQUESTS	
USER 1 HOLDS	S Lock	X Lock
S Lock	Lock Granted	User 2 Waits
X Lock	User 2 Waits	User 2 Waits

- A shared (S) lock must be obtained before reading a database item, whereas an exclusive (X) lock must be obtained before writing
- Any number of users can hold a S lock on the same item but only one user can hold X lock

Lock Granularity

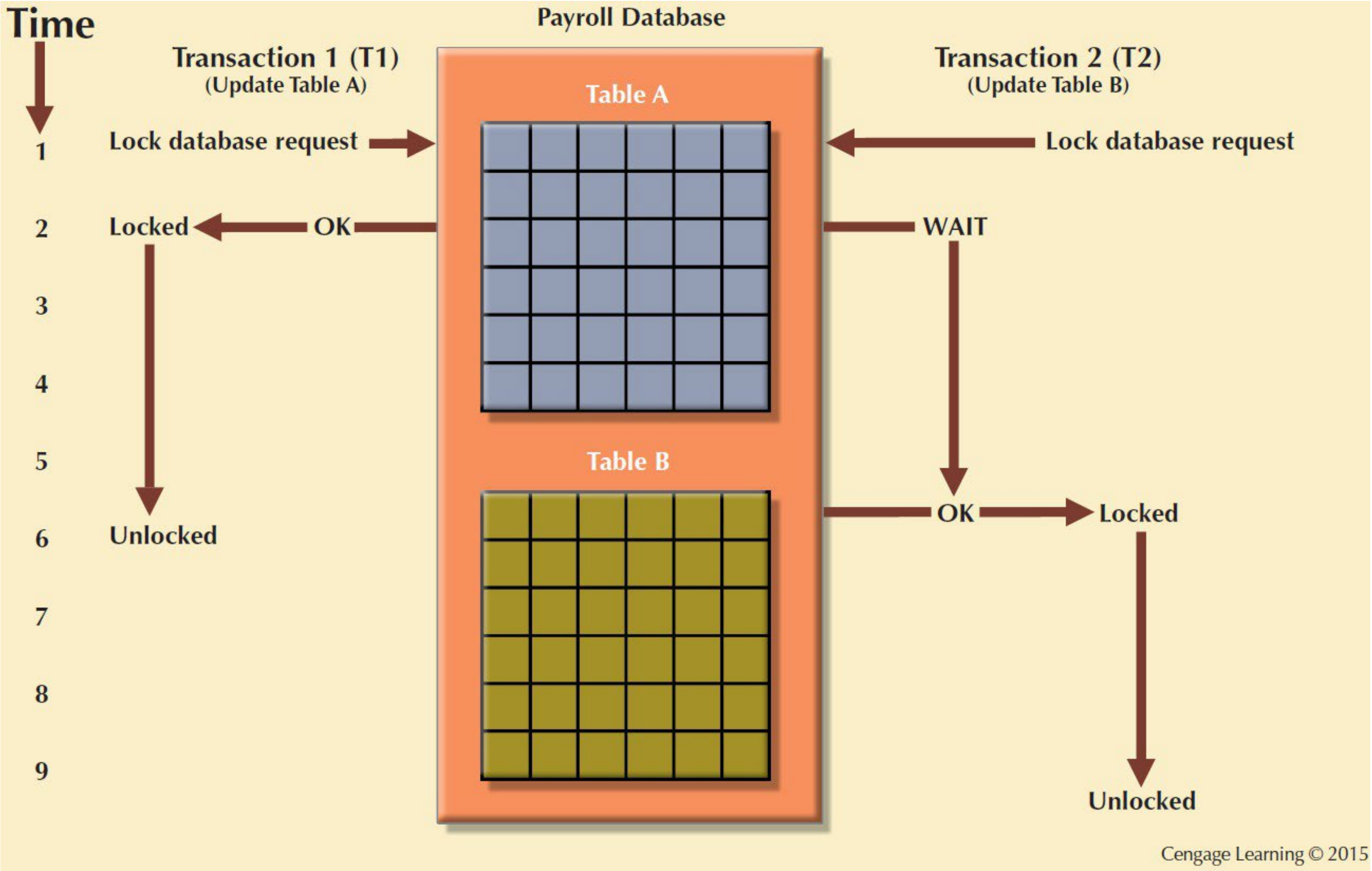
- Indicates the level of lock use
- Levels of locking
  - **Database-level lock**
  - **Table-level lock**
  - **Page-level lock**
    - **Page** or **diskpage**: Directly addressable section of a disk
  - **Row-level lock**
  - **Field-level lock**

Lock Granularity

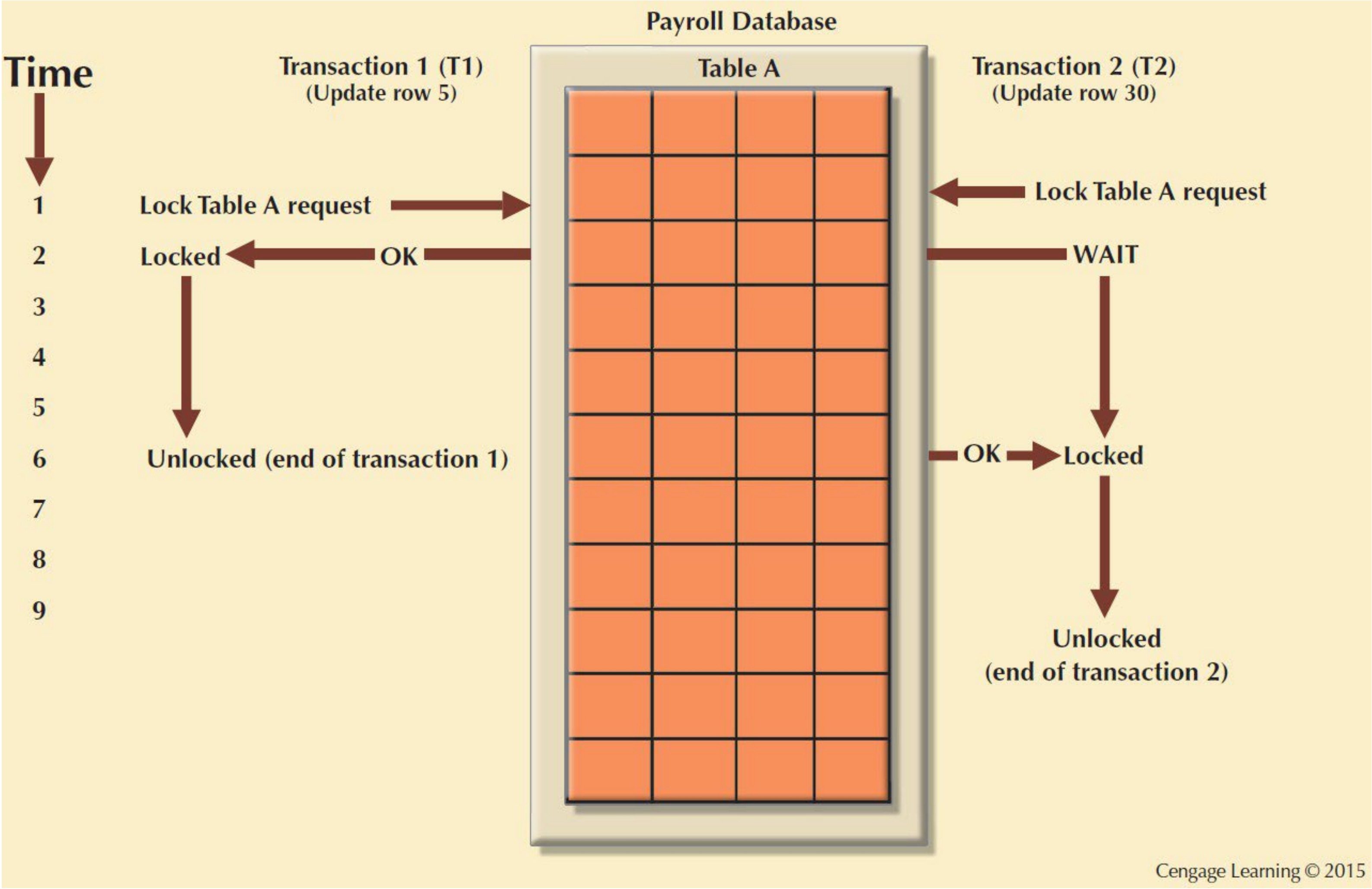


- The size of the database item locked
- A trade-off between waiting time (amount of concurrency permitted) and overhead (number of locks held)

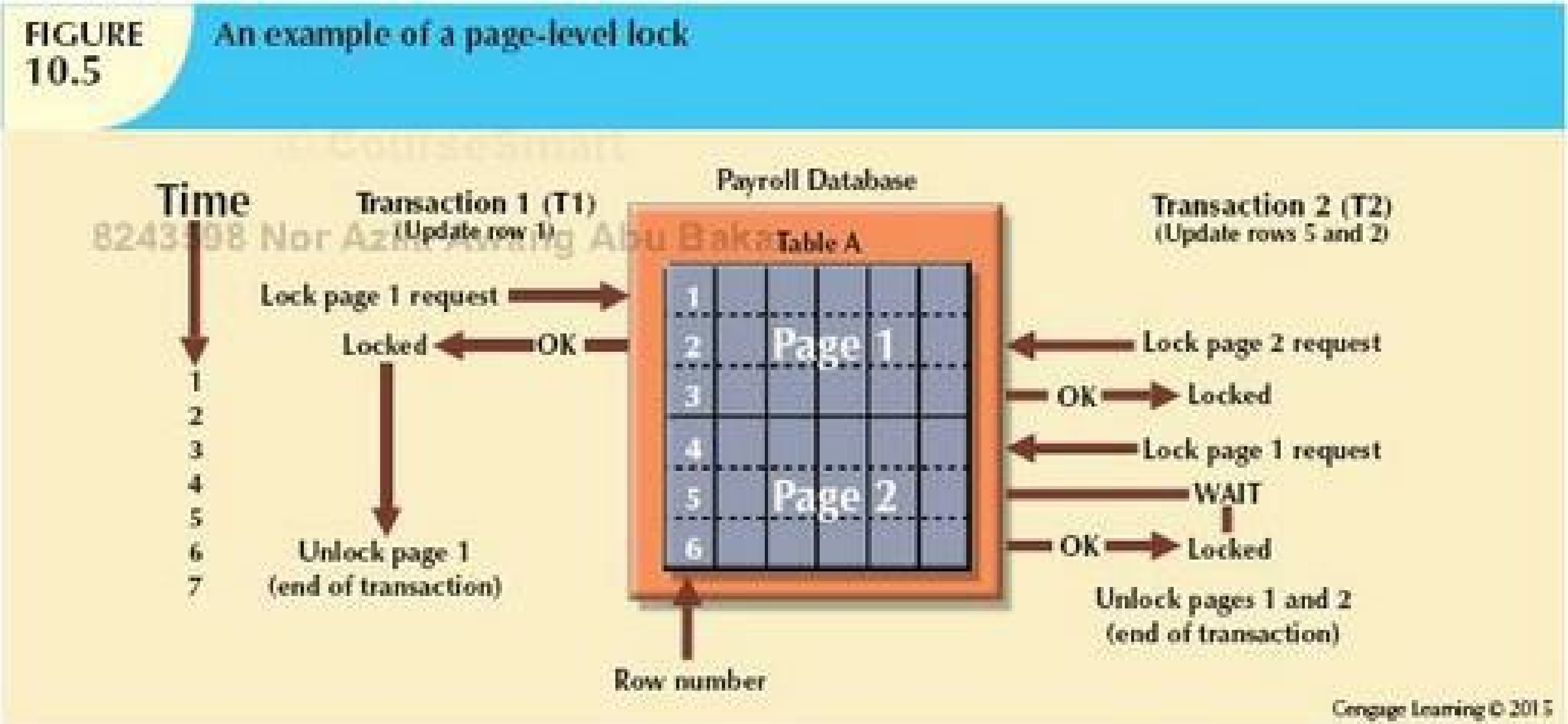
Database-Level Locking Sequence



An Example of a Table-Level Lock



An Example of a Page-Level Lock



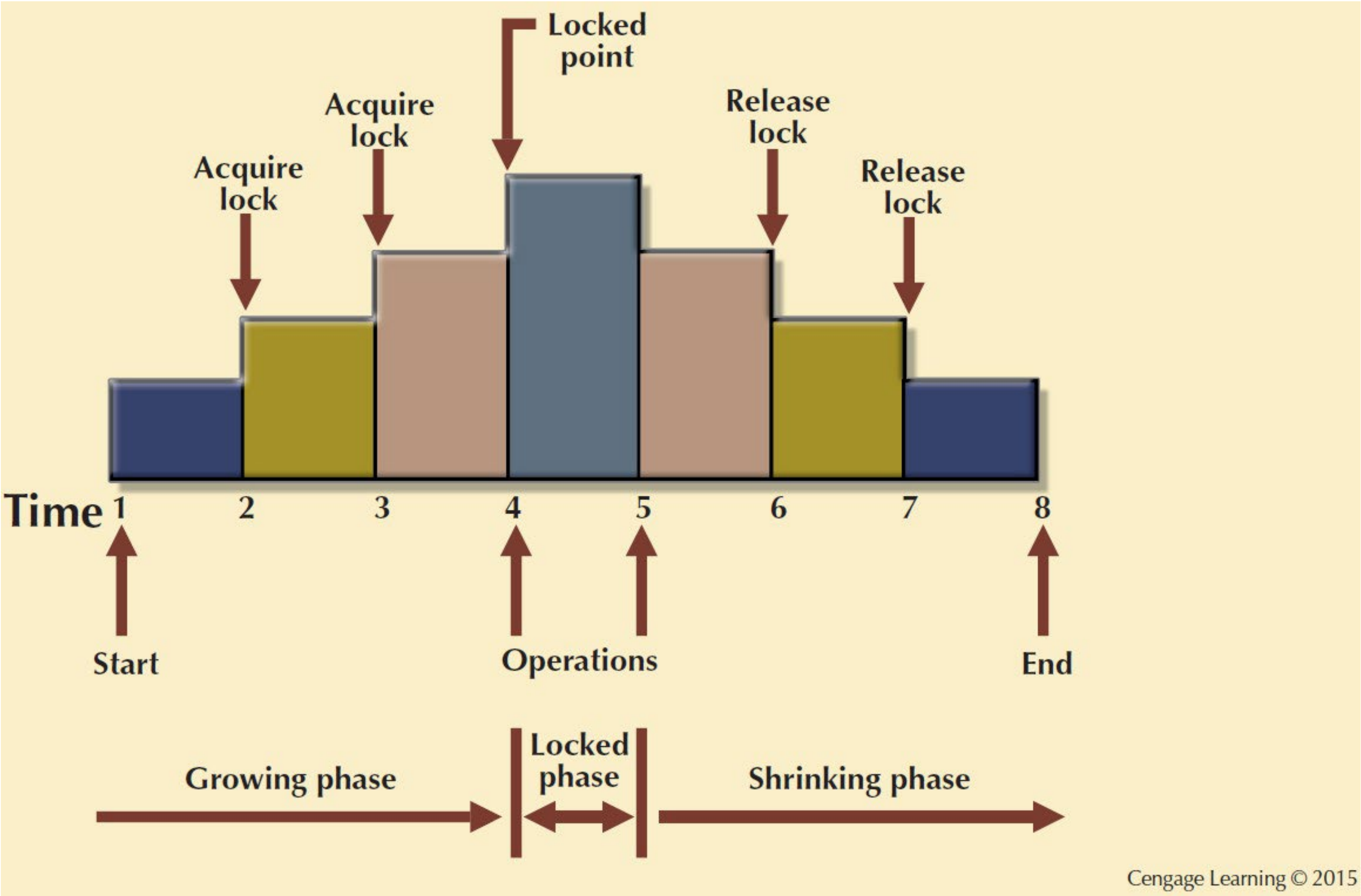
Two-Phase Locking (2PL)

- Defines how transactions **acquire** and **relinquish locks**.
- Guarantees serializability but does not prevent deadlocks.
- Two phases for transaction:
  1. **Growing phase** – Transaction acquires all locks but cannot release any locks.
  2. **Shrinking phase** – Transaction releases all locks but cannot acquire any new locks.

Two-Phase Locking (2PL)

- Governing rules
  - Two transactions cannot have conflicting locks
  - No unlock operation can precede a lock operation in the same transaction
  - No data are affected until all locks are obtained

Two-Phase Locking Protocol



Two Phase Locking (2PL) Protocol

Protocol to prevent lost update problems

All transactions must follow (protocol) to ensure that concurrency problems do not occur

Conditions

- Obtain lock before accessing item
- Wait if a conflicting lock is held
- Cannot obtain new locks after releasing locks

Preventing Lost Update Problem Using 2PL

Time	T1	T2	Bal_x
t1		Begin_transaction	100
t2	Begin_transaction	write_lock (bal_x)	100
t3	Write_lock (bal_x)	read (bal_x)	100
t4	WAIT	$Bal_x = bal_x + 100$	100
t5	WAIT	Write (bal_x)	200
t6	WAIT	Commit/unlock(bal_x)	200
t7	read (bal_x)		200
t8	$Bal_x = Bal_x – 10$		200
t9	Write (bal_x)		190
t10	Commit/unlock (bal_x)		190

Preventing Uncommited Dependency Problem Using 2PL

Time	T3	T4	Bal_x
t1		Begin_transaction	100
t2		write_lock (bal_x)	100
t3		read (bal_x)	100
t4	Begin_transaction	$Bal_x = bal_x + 100$	100
t5	Write_lock (bal_x)	Write (bal_x)	200
t6	WAIT	Rollback/unlock (bal_x)	100
t7	read (bal_x)		100
t8	$Bal_x = Bal_x – 10$		100
t9	Write (bal_x)		90
t10	Commit/unlock (bal_x)		90

Preventing Inconsistent Retrievals 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

Deadlocks

- 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).

How a Deadlock Condition is Created

Time	T_{17}	T_{18}
t_1	begin_transaction	
t_2	write_lock($\mathbf{bal_x}$)	begin_transaction
t_3	read($\mathbf{bal_x}$)	write_lock($\mathbf{bal_y}$)
t_4	$\mathbf{bal_x} = \mathbf{bal_x} - 10$	read($\mathbf{bal_y}$)
t_5	write($\mathbf{bal_x}$)	$\mathbf{bal_y} = \mathbf{bal_y} + 100$
t_6	write_lock($\mathbf{bal_y}$)	write($\mathbf{bal_y}$)
t_7	WAIT	write_lock($\mathbf{bal_x}$)
t_8	WAIT	WAIT
t_9	WAIT	WAIT
t_{10}	⋮	WAIT
t_{11}	⋮	⋮

Deadlocks - Control techniques

- Timeouts
- Deadlock prevention
- Deadlock detection and recovery

Deadlock - Timeouts

- **Transaction** that **requests lock** will only **wait for a system defined period**.
- 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

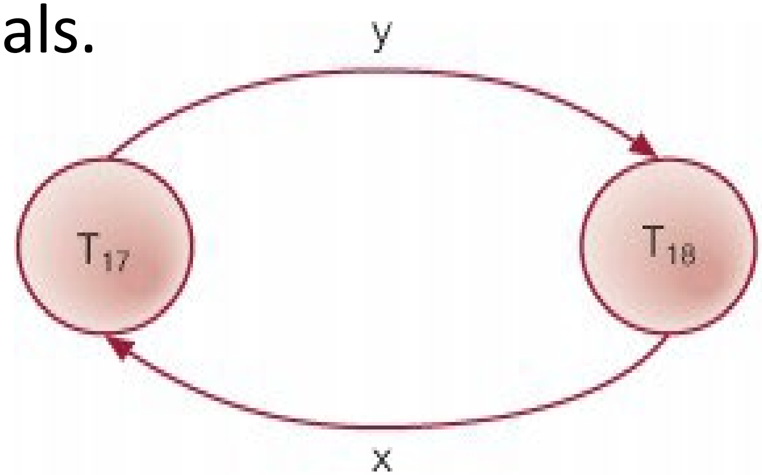
- DBMS looks ahead to see if transaction would cause deadlock and never allows deadlock to occur.
- Could order transactions using transaction timestamps.

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).

https://youtu.be/WZtebOyiu0M

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 Ti ➔ Tj, if Ti waiting to lock item locked by Tj.
- Deadlock exists if and only if WFG contains cycle.
- WFG is created at regular intervals.



Timestamping Methods

Timestamp:

A unique identifier created by the DBMS that indicates the relative starting time of a transaction.

Timestamping:

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

Exercise 1

a) Consider the following schedule:

$$S = [R3(Z), R2(X), W1(X), W3(Y), R1(Y), W2(Z), W1(Z)]$$

i. Draw the transaction table and by referring above schedule

ii. Draw a precedence or a serialization graph for the schedule S.

iii. Is the schedule serializable?

Exercise 2

Check whether a schedule is conflict serializable or not. Produce a precedence graph.

T1	T2	T3
R(x)		
		R(y)
		R(x)
	R(y)	
	R(z)	
		W(y)
	W(z)	
R(z)		
W(x)		
W(z)		

https://youtu.be/nvXWxObRR1U

Exercise 3

Consider the transactions T1 and T2 below:

T1	Time	T2	Value
Read Balance	1		1200
Balance = Balance + 1000	2		1200
Write Balance	3	Read Balance	2200
Rollback	4	Balance = Balance – 1000	1200
	5	Write Balance	1200

i) Name the interference problem above.

(1 mark)

ii) Rewrite the transactions so that they obey the two-phase locking protocol

(6 marks)

Exercise 4

Given the following transactions T1 and T2:

Time	T1	T2	A	B
1		Begin Trans	200	150
2	Begin Trans	Read (A)		
3	Read (A)	A=A-X		
4	Read (B)	Read (B)		
5	If (A>B) then A=A+X	B=B+Y		
6		Write (A)		
7	Write (A)	Write (B)		
8	Commit	Commit		

a) Name the concurrency problem for the transaction above. (1 mark)

b) Suppose X=80 and Y=65, fill the details for A and B of the above transactions. (6 marks)

c) What is the final value of A and B if T1 and T2 are executed serially? (4 marks)

d) Rewrite the above transactions so that they obey the two-phased locking (2PL). (6 marks)