Transactions

WHY TRANSACTIONS?

- *Transaction* is a process involving database queries and/or modification.

- Database systems are normally being accessed by many users or processes at the same time.

- Example- ATM

- Formed in SQL from single statements or explicit programmer control

ACID Transactions

- *ACID transactions* are:
  - *Atomic* : Whole transaction or none is done.
  - *Consistent* : Database constraints preserved.
  - *Isolated* : It appears to the user as if only one process executes at a time.
  - *Durable* : Effects of a process survive a crash.

- Optional: weaker forms of transactions are often supported as well.

EXAMPLE OF *FUND TRANSFER*

- Transaction to transfer \$50 from account **A** to account **B**:
  1. **read**(A)
  2. $A := A - 50$
  3. **write**(A)
  4. **read**(B)
  5. $B := B + 50$
  6. **write**(B)

- **Atomicity requirement** :
  - if the transaction **fails** after step 3 and before step 6,
    - the **system** should **ensure** that :
      - its **updates** are *not reflected* in the database,
      - else an *inconsistency* will result.

Example of *Fund Transfer*

- Transaction to transfer \$50 from account **A** to account **B**:
  1. **read**(A)
  2. $A := A - 50$
  3. **write**(A)
  4. **read**(B)
  5. $B := B + 50$
  6. **write**(B)

- **Consistency requirement** :
  - the **sum** of **A** and **B** is:
    - **unchanged** by the execution of the transaction.

Example of *Fund Transfer* (Cont.)

- **Isolation requirement** —
  - if between steps 3 and 6,
    - another transaction is allowed to access the partially updated database,
      - it will see an inconsistent database
      - (the sum $A + B$ will be less than it should be).
  - Isolation can be **ensured** trivially by:
    - running transactions **serially**,
      - that is **one** after the **other**.
  - *However*, executing multiple *transactions* **concurrently**
    - has significant **benefits**, as we will see later.

Example of *Fund Transfer* (Cont.)

- **Durability requirement** :
  - once the user has been notified that the transaction has **completed** :
    - (i.e., the transfer of the $50 has taken place),
    - the **updates** to the database by the transaction **must persist**
      - despite *failures*.

Example: Interacting Processes

- Consider a relation Sells(shop, item, price)
- Let Ahmad sells Sprite for Rs12.50 and Pizza Slize for Rs 35.00.
- Tania is querying Sells for the highest and lowest price Ahmad charges.
- Ahmad decides to stop selling Sprite and Pizza, but to sell Biryani at Rs75.00 per plate.

Customer's Program

- Tania(customer) executes the following two SQL statements called (min) and (max).

(max) SELECT MAX(price) FROM Sells
WHERE shop = 'Ahmad''s shop';
(min) SELECT MIN(price) FROM Sells
WHERE shop = 'Ahmad''s shop';

Shop Keeper's Program

- At about the same time, Ahmad executes the following steps: (del) and (ins).

```
(del)   DELETE FROM Sells
        WHERE shop = 'Ahmad''s shop';
(ins)   INSERT INTO Sells
        VALUES('Ahmad''s shop', 'Biryani', 75.00);
```

Interleaving of Statements

- The statement (max) must come before (min), and
- The statement (del) must come before (ins),
- There are no other constraints on the order of these statements, unless we group Tania's and/or Ahmad's statements into transactions.

Example: Strange Interleaving

- Suppose the steps execute in the order (max)(del)(ins)(min).

Ahmad's Prices:	{12.50,35.00}	{12.50,35.00}	{75.00}	{75.00}
Statement:	(max)	(del)	(ins)	(min)
Result:	35.00			75.0

- Tania sees MAX < MIN!

Another Problem: Rollback

- Suppose Ahmad executes (del)(ins), not as a transaction.
- But after executing these statements, he change his mind and issues a ROLLBACK statement.
- If Tania executes her statements after (ins) but before the rollback, she sees a value, 75.00, that never existed in the database.

SOLUTION

- If Ahmad executes (del)(ins) as a transaction, its effect cannot be seen by others until the transaction executes COMMIT.
  - If the transaction executes ROLLBACK instead, then its effects can *never* be seen.

Fixing the Problem by Using Transactions

- Solution: Group Tania's statements (max)(min) into one transaction
  - Now, she cannot see this inconsistency.

- She sees Ahmad's prices at some fixed time.
  - Either before or after he changes prices, or in the middle, but the MAX and MIN are computed from the same prices.

Transaction Processing

- Basic operations in a DB are **read** and **write**
  - **read_item(X)**:
    - Reads a database item named X into a program variable.
    - To simplify our notation, we assume that the program variable is also named X.
  - **write_item(X)**:
    - Writes the value of program variable X into the database item named X.

Sample transactions

(a)	T_1	(b)	T_2
	read_item (X);		read_item (X);
	$X:=X-N$;		$X:=X+M$;
	write_item (X);		write_item (X);
	read_item (Y);		
	$Y:=Y+N$;		
	write_item (Y);		

Issues in Transaction Processing

Why Concurrency Control is needed:

- **The Lost Update Problem**
  - Two transactions (that access the same DB items) have their operations interleaved in a way that makes the value of some database item incorrect.

Time

T_1	T_2
read_item(X); $X := X - N$;	
	read_item(X); $X := X + M$;
write_item(X); read_item(Y);	
	write_item(X);
$Y := Y + N$; write_item(Y);	

Item X has an incorrect value because its update by T_1 is *lost* (overwritten).

ISSUES IN TRANSACTION PROCESSING

- **Temporary Update (or Dirty Read) Problem**

Time

T_1	T_2

read_item(X);
$X := X - N$;
write_item(X);	

	read_item(X);
	$X := X + M$;
	write_item(X);

read_item(Y);	

Transaction T_1 fails and must change the value of X back to its old value; meanwhile T_2 has read the *temporary* incorrect value of X.

Issues in Transaction Processing

- **The Incorrect Summary Problem**

T_1	T_3
	$sum := 0$; read_item(A); $sum := sum + A$; ⋮

read_item(X);
$X := X - N$;
write_item(X);	

	read_item(X);
	$sum := sum + X$;
	read_item(Y);
	$sum := sum + Y$;

read_item(Y);
$Y := Y + N$;
write_item(Y);	

T_3 reads X after N is subtracted and reads Y before N is added; a wrong summary is the result (off by N).

Transaction Processing &Recovery

- Why **recovery** is needed:
  - A computer failure (system crash):
  - A transaction or system error:
    - Integer overflow, division by zero, erroneous parameter values or the user may interrupt the transaction
  - Local errors or exception conditions
    - Data not found or
    - insufficient account balance may cause a fund withdrawal transaction to be canceled.
  - Concurrency control enforcement
    - Transaction violates serializability or several transactions are in a state of deadlock
  - Disk failure
  - Physical problems and catastrophes

Transaction and System Concepts

- A **transaction** is an atomic unit of work that is either completed in its entirety or not done at all.
- For recovery purposes, the system needs to keep track of when the transaction starts, terminates, and commits or aborts

- **Transaction states**:
  - Active state
  - Partially committed state
  - Committed state
  - Failed state
  - Terminated State

State transition diagram

Recovery manager keeps track of the following operations:

- **begin_transaction**
- **read** or **write**
- **end_transaction**
- **Commit**
- **Rollback** (or **abort**)
- **Undo**
- **Redo**

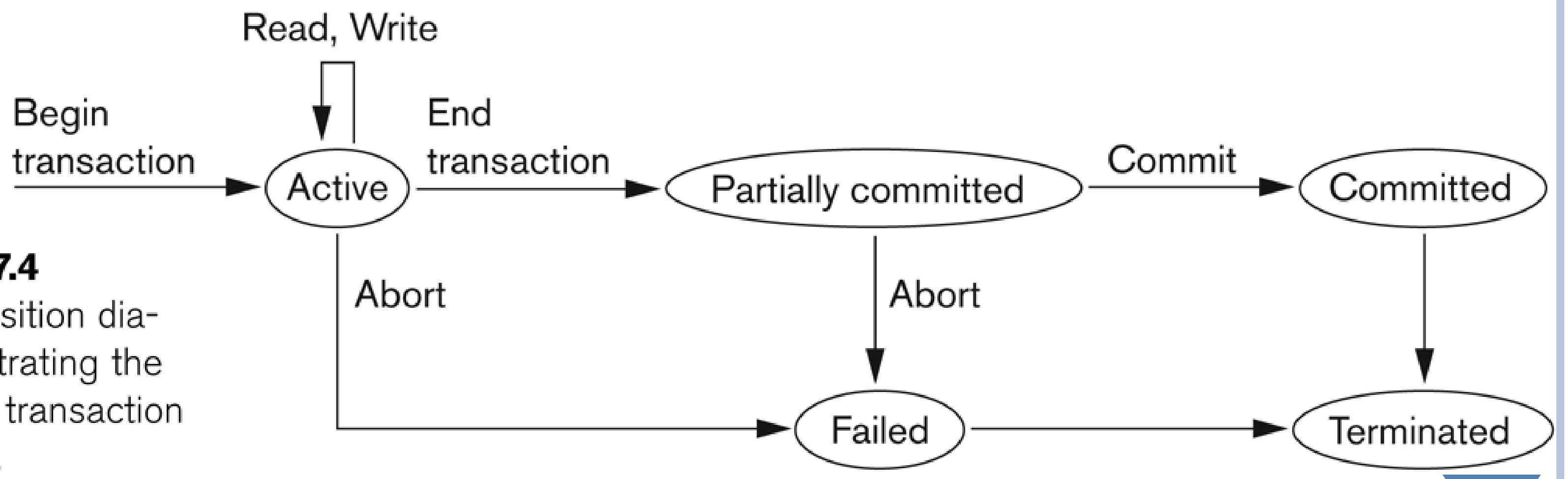


Figure 17.4
State transition diagram illustrating the states for transaction execution.

Schedules

- A **schedule** *S* of *n* transactions *T*1, *T*2, ..., *Tn* is an ordering of the operations of the transactions.
- Operations from different transactions can be interleaved in *S*.
- The operations of each *Ti* in *S* must appear in the same order in which they occur in *Ti*.

Time

T_1	T_2
read_item(X); $X := X - N$;	
	read_item(X); $X := X + M$;
write_item(X); read_item(Y);	
	write_item(X);
$Y := Y + N$; write_item(Y);	

S_a: $r_1(X)$; $r_2(X)$; $w_1(X)$; $r_1(Y)$; $w_2(X)$; $w_1(Y)$;

Characterizing Schedules based on Serializability

- **Serial schedule**: A schedule S is serial if, for every T in S, all the operations of T are executed consecutively in the schedule.

(a)

Time

T_1	T_2

read_item(X);
$X := X - N$;
write_item(X);
read_item(Y);
$Y := Y + N$;
write_item(Y);	

	read_item(X);
	$X := X + M$;
	write_item(X);

Schedule A

(b)

Time

T_1	T_2

	read_item(X);
	$X := X + M$;
	write_item(X);

read_item(X);
$X := X - N$;
write_item(X);
read_item(Y);
$Y := Y + N$;
write_item(Y);	

Schedule B

Characterizing Schedules based on Serializability

- **Non Serial schedule**
  - **Which one is correct ?**

(c)

Time ↓

T_1	T_2
read_item(X); $X := X - N$;	
	read_item(X); $X := X + M$;
write_item(X); read_item(Y);	
	write_item(X);
$Y := Y + N$; write_item(Y);	

Schedule C

Time ↓

T_1	T_2

read_item(X);
$X := X - N$;
write_item(X);	

	read_item(X);
	$X := X + M$;
	write_item(X);

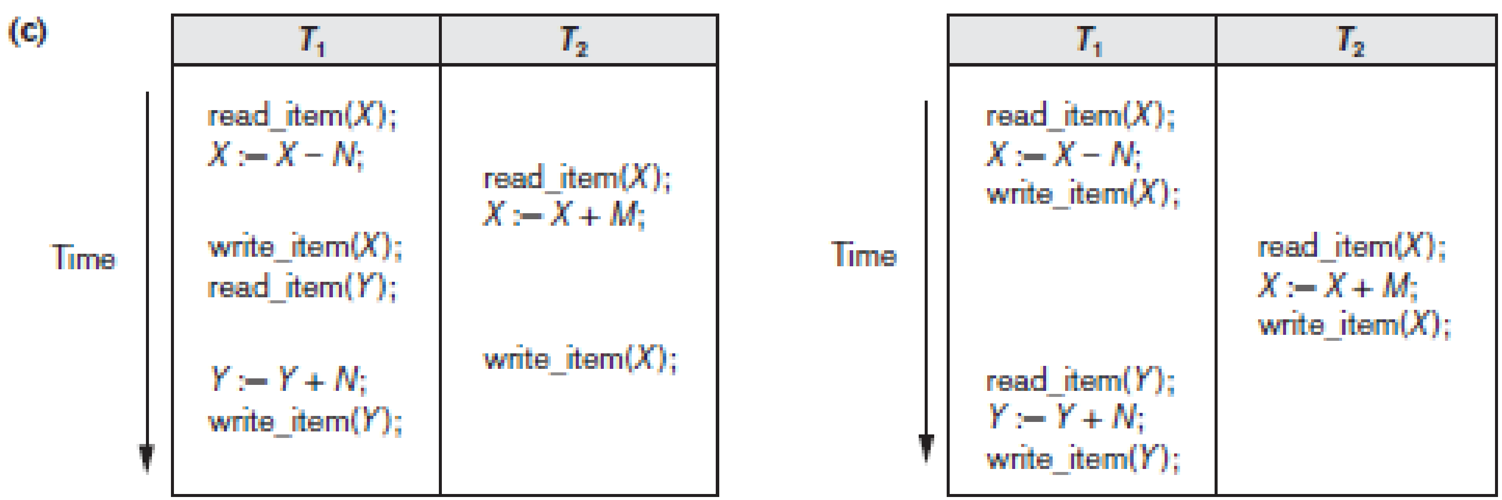
read_item(Y);
$Y := Y + N$;
write_item(Y);	

Schedule D

  - Schedule C gives an erroneous result because of the *lost update problem*
  - Schedule D gives correct results

Characterizing Schedules based on Serializability

- **Serializable schedule**: A schedule S is serializable if it is equivalent to some serial schedule of the same n transactions.
  - A nonserial schedule *S* is serializable means it is correct,



Schedule C

Schedule D

SCHEDULES- CONFLICT

- Two operations in S are in **conflict** if
  - they belong to *different transactions*;
  - they access the *same item* X; and
  - *at least one* of the operations is a write_item(X).

S: r1(x); w1(X); r2(X); w2(x); r1(y); a_1;

- Conflicting operations
  - $r1(X)$ and $w2(X)$
  - $r2(X)$ and $w1(X)$
  - $w1(X)$ and $w2(X)$
- Non-conflicting operations
  - $r1(X)$ and $r2(X)$
  - $w2(X)$ and $w1(Y)$
  - $r1(X)$ and $w1(X)$

Intuitively, two operations are conflicting if changing their order can result in a different outcome.

Types of Conflict: **read-write conflict and write-write conflict**

Characterizing Schedules based on Serializability

- **Conflict equivalent:** Two schedules are said to be conflict equivalent if the order of any two conflicting operations is same in both schedules.

Two operations on the same data item conflict if at least one of the operations is a write

- $r(X)$ and $w(X)$ conflict
- $w(X)$ and $r(X)$ conflict
- $w(X)$ and $w(X)$ conflict

- If two conflicting operations are applied in *different orders* in two schedules, the effect can be different

Serializability

- A schedule S is said to be conflict serializable if it is conflict equivalent to some serial schedule S’.

T_1	T_2
read_item(X);	
$X := X - N$;	
write_item(X);	
read_item(Y);	
$Y := Y + N$;	
write_item(Y);	
	read_item(X);
	$X := X + M$;
	write_item(X);

Schedule A

(b)

T_1	T_2
	read_item(X);
	$X := X + M$;
	write_item(X);
read_item(X);	
$X := X - N$;	
write_item(X);	
read_item(Y);	
$Y := Y + N$;	
write_item(Y);	

Schedule B

T_1	T_2
read_item(X);	
$X := X - N$;	
	read_item(X);
	$X := X + M$;
write_item(X);	
read_item(Y);	
	write_item(X);
$Y := Y + N$;	
write_item(Y);	

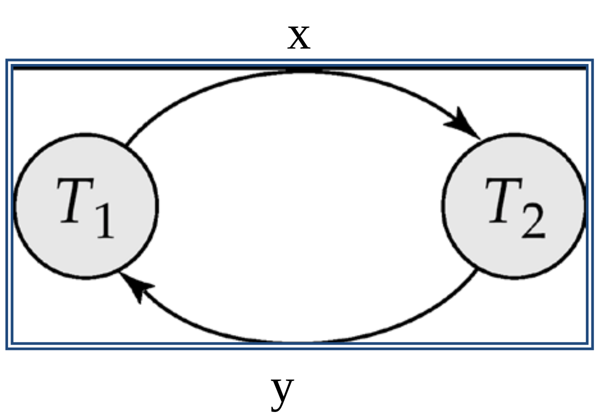
Schedule C

T_1	T_2
read_item(X);	
$X := X - N$;	
write_item(X);	
	read_item(X);
	$X := X + M$;
	write_item(X);
read_item(Y);	
$Y := Y + N$;	
write_item(Y);	

Schedule D

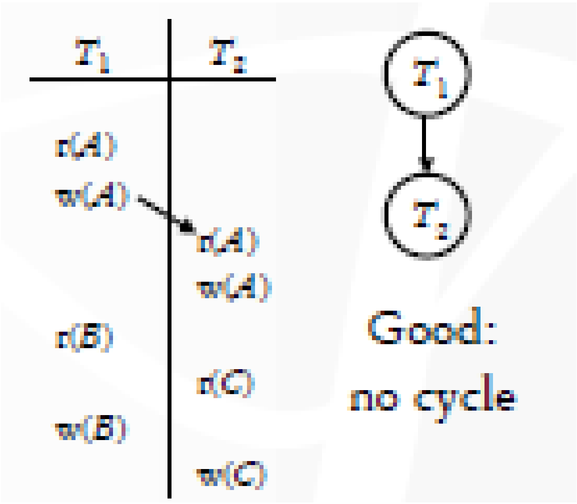
Testing for Serializability

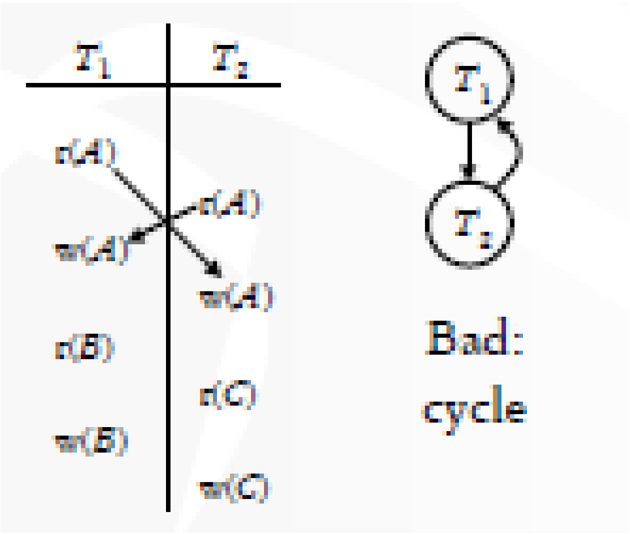
- Consider some schedule of a <u>set of transactions</u> $T_1, T_2, \ldots, T_n$
- **Precedence graph** :
  - a ***direct graph*** where the <u>vertices</u> ***are*** the <u>*transactions*</u> (names).
- Draw an <u>arc</u> from T_i to T_j
  - <u>if</u> the two transaction <u>conflict</u>, and
  - T_i <u>accessed the data</u> item on which the conflict arose **earlier**.
- We may <u>label</u> the <u>arc</u> by the **item** that was **accessed**.



Precedence Graph

- A node for each transaction
- A directed edge from Ti to Tj if an operation of Ti precedes and conflicts with an operation of Tj in the schedule





Constructing the Precedence Graphs

T_1	T_2

read_item(X);
$X := X - N$;
write_item(X);
read_item(Y);
$Y := Y + N$;
write_item(Y);	

	read_item(X);
	$X := X + M$;
	write_item(X);

Schedule A

(b)

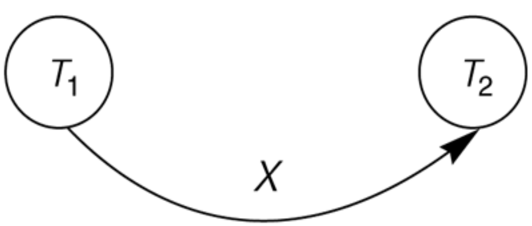
T_1	T_2

	read_item(X);
	$X := X + M$;
	write_item(X);

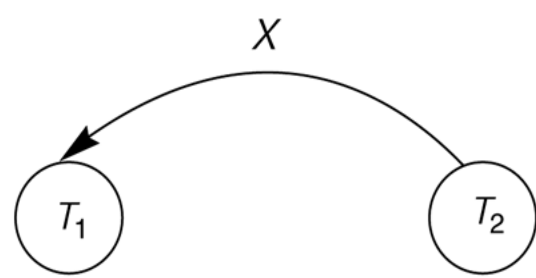
read_item(X);
$X := X - N$;
write_item(X);
read_item(Y);
$Y := Y + N$;
write_item(Y);	

Schedule B

(a)



(b)



Constructing the Precedence Graphs

me

T_1	T_2
read_item(X);	
$X := X - N$;	
	read_item(X);
	$X := X + M$;
write_item(X);	
read_item(Y);	
	write_item(X);
$Y := Y + N$;	
write_item(Y);	

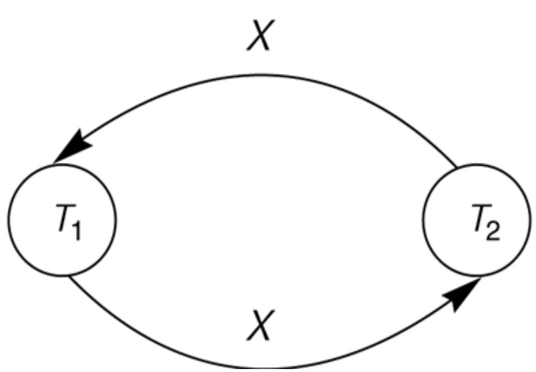
Schedule C

Time

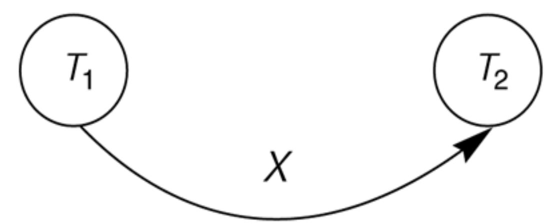
T_1	T_2
read_item(X);	
$X := X - N$;	
write_item(X);	
	read_item(X);
	$X := X + M$;
	write_item(X);
read_item(Y);	
$Y := Y + N$;	
write_item(Y);	

Schedule D

(c)



(d)



Example of Serializability Testing

Transaction T_1
read_item(X);
write_item(X);
read_item(Y);
write_item(Y);

Transaction T_2
read_item(Z);
read_item(Y);
write_item(Y);
read_item(X);
write_item(X);

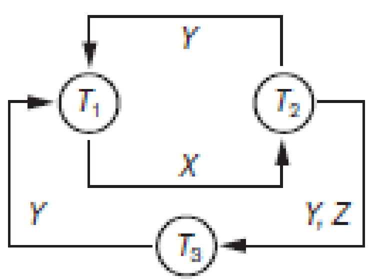
Transaction T_3
read_item(Y);
read_item(Z);
write_item(Y);
write_item(Z);

Example of Serializability Testing

Transaction T_1	Transaction T_2	Transaction T_3
	read_item(Z);	
	read_item(Y);	
	write_item(Y);	
		read_item(Y);
		read_item(Z);
read_item(X);		
write_item(X);		
		write_item(Y);
		write_item(Z);
	read_item(X);	
read_item(Y);		
write_item(Y);		
	write_item(X);	

Time

Schedule E



Equivalent serial schedules

None

Reason

Cycle $X(T_1 \rightarrow T_2), Y(T_2 \rightarrow T_1)$
Cycle $X(T_1 \rightarrow T_2), YZ\ (T_2 \rightarrow T_3), Y(T_3 \rightarrow T_1)$

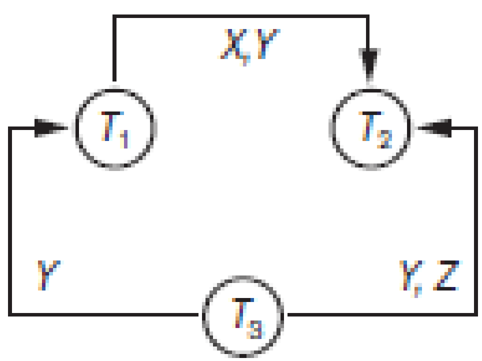
Example of Serializability Testing

Time ↓

Transaction T_1	Transaction T_2	Transaction T_3
		read_item(Y); read_item(Z);
read_item(X); write_item(X);		
		write_item(Y); write_item(Z);
	read_item(Z);	
read_item(Y); write_item(Y);		

	read_item(Y);
	write_item(Y);
	read_item(X);
	write_item(X);	

Schedule F



Equivalent serial schedules

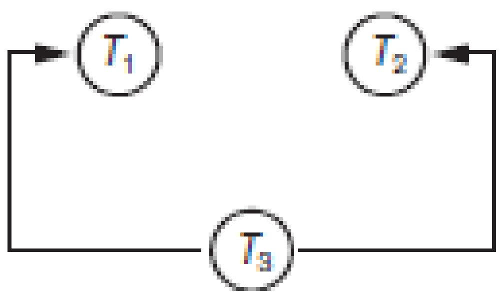
$T_3 \rightarrow T_1 \rightarrow T_2$

EXAMPLE OF SERIALIZABILITY TESTING

Transaction T_1
read_item(X);
write_item(X);
read_item(Y);
write_item(Y);

Transaction T_2
read_item(Z);
read_item(Y);
write_item(Y);
read_item(X);
write_item(X);

Transaction T_3
read_item(Y);
read_item(Z);
write_item(Y);
write_item(Z);



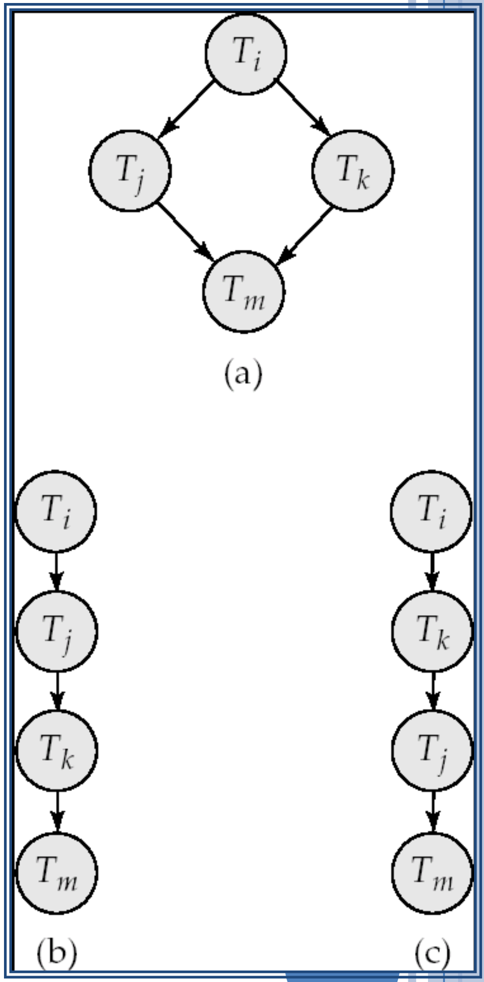
Equivalent serial schedule

$T_3 \rightarrow T_1 \rightarrow T_2$

$T_3 \rightarrow T_2 \rightarrow T_1$

Test for *Conflict* Serializability

- A schedule is *conflict serializable*
  - **if** and only **if** its *precedence graph is* ***acyclic***.
- **Cycle-detection** algorithm:
  - Use Depth-first search to detect cycle
    - DFS for a connected graph produces a tree.
    - There is a cycle in a graph only if there is a **back edge** present in the graph
- **If** precedence graph is **acyclic**, the *serializability order* can be obtained by a ***topological sorting*** of the graph.
  - A **topological ordering** of a directed graph is a linear ordering of its vertices such that for every directed edge *uv* from vertex *u* to vertex *v*, *u* comes before *v* in the ordering.



Recoverable Schedules

Need to address the effect of *transaction* ***failures*** on concurrently *running transactions*

- **Recoverable schedule**:
  - if a transaction T_b **reads** a data item *previously* **written** by T_a ,
  - then the **commit** of T_a should appears **before** the **commit** of T_b.

- The following schedule is **not** recoverable
  - if T_b **commits** immediately after the read

Ta	Tb
read(A)	
write(A)	
	read(A)
read(B)	

  - If T_a **abort**, T_b have done a dirty read .
- Database must ensure that schedules are recoverable.

CHARACTERIZING SCHEDULES BASED ON RECOVERABILITY

- In a recoverable schedule, no committed transaction ever needs to be rolled back

$$S_a': r_1(X); r_2(X); w_1(X); r_1(Y); w_2(X); c_2; w_1(Y); c_1;$$

$$S_c: r_1(X); w_1(X); r_2(X); r_1(Y); w_2(X); c_2; a_1;$$

Cascading Rollbacks

- **Cascading rollback**:
  - a **single** *transaction failure* **leads** to a series of transaction rollbacks.
  - Consider the following schedule where:
    - **none** of the transactions has yet **committed**
    - (so the schedule is **recoverable**)

T1	T2	T3
read(A)		
read(B)		
write(A)		
	read(A)	
	write(A)	
		read(A)

  - If T_1 *fails*, T_2 and T_3 must also be *rolled back*.
- Can **lead** to the *undoing* of a *significant amount of work*

Characterizing Schedules based on Recoverability

- **Cascadeless schedule**
  - One where every transaction reads only the items that are written by committed transactions.
  - Avoid cascade rollback
    - **Sd: r1(X); w1(X); r2(X); r1(Y); w2(X); w1(Y); c1; c2;**
    - **Se: r1(X); w1(X); r2(X); r1(Y); w2(X); w1(Y); a1; a2;**

    - The $r_2(X)$ command in schedules S_d and S_e must be postponed until after T_1 has committed (or aborted),
    - This delays T_2 but ensuring no cascading rollback if T_1 aborts

Characterizing Schedules based on Recoverability

- **Strict Schedules**:
  - A schedule in which a transaction can neither read or write an item X until the last transaction that wrote X has committed.
  - Strict schedules simplify the recovery process.
  - The process of undoing a **write_item(*X*)** of an aborted transaction is to restore the **before image** (old_value) of *X*.

$$S_f: w_1(X, 5); w_2(X, 8); a_1;$$

  - *Sf* is cascadeless, but it is not a strict schedule,
    - As it permits *T*2 to write item *X* even though *T*1 that last wrote *X* had not yet committed (or aborted).

Characterizing Schedules based on Recoverability

- Every strict schedule
  - is also cascadeless,
- Every **cascadeless** schedule:
  - is also **recoverable**
- It is ***desirable*** to **restrict** the schedules to:
  - those that are ***cascadeless***

Characterizing Schedules based on Serializability

- Serializability is hard to check.
  - Interleaving of operations occurs in an operating system through some scheduler
  - Difficult to determine beforehand how the operations in a schedule will be interleaved.
- Current approach used in most DBMSs:
  - Use of locks with two phase locking

SERIALIZABILITY

- Enforced with locks, like in Operating Systems !
- But this is not enough:

User 1

LOCK A
[write A=1]
UNLOCK A
. . .
. . .
. . .
. . .
LOCK B
[write B=2]
UNLOCK B

User 2

LOCK A
[write A=3]
UNLOCK A
LOCK B
[write B=4]
UNLOCK B

time

What is wrong ?

SERIALIZABILITY

- A **lock** is a ***mechanism*** to *control concurrent access* to a data item

- Solution: two-phase locking
  - Lock everything at the beginning
  - Unlock everything at the end

- Read locks: many simultaneous read locks allowed
- Write locks: only one write lock allowed

LOCK-BASED PROTOCOLS

- Data items can be locked in *two modes* :
  - ***shared*** *(S) mode*.
    - Data item can only be read.
    - More than one transaction can apply share lock on X for reading its value but no write lock can be applied on X by any other transaction.
  - ***exclusive*** *(X) mode*.
    - Data item can be both read as well as written.
    - Only one write lock on X can exist at any time and no shared lock can be applied by any other transaction on X.

	Read	Write
Read	Y	N
Write	N	N

Conflict matrix

Lock-Based Protocols

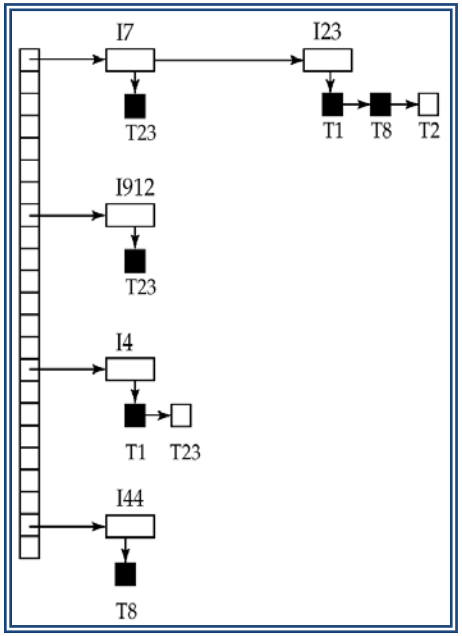
- Lock requests are made to *concurrency-control* ***manager***.
  - Transaction **can proceed** only after *request* is granted

- If a lock cannot be granted,
  - the requesting transaction is made to **wait** till:
    - **all** incompatible locks held by other transactions have been released. Then the lock is then granted.

- A **locking protocol** is a <u>set of rules</u> *followed by*
  - **all transactions** while requesting and releasing locks.

- <u>Locking protocols</u> **restrict** the <u>set of possible</u> **schedules**.

The Two-Phase Locking Protocol



Essential components

- Lock Manager:
  - Managing locks on data items.
- Lock table:
  - Lock manager uses it to store
    - the identify of transaction locking a data item,
    - the data item,
    - lock mode and
    - pointer to the next data item locked.
  - One simple way to implement a lock table is through linked list.

Transaction ID	Data item id	lock mode	Ptr to next data item
T1	X1	Read	Next

The Two-Phase Locking Protocol

- This is a protocol which ensures conflict-serializable schedules.

- Phase 1: **Growing** Phase
  - transaction may obtain locks
  - transaction *may not release* locks
- Phase 2: **Shrinking** Phase
  - transaction may release locks
  - transaction *may not obtain* locks

- **Requirement:**
  - For a transaction these two phases must be mutually exclusively,

The Two-Phase Locking Protocol

Two-Phase Locking Techniques: The algorithm

<u>T1</u>	<u>T2</u>	<u>Result</u>
read_lock (Y);	**read_lock (X);**	Initial values: X=20; Y=30
read_item (Y);	read_item (X);	Result of serial execution
unlock (Y);	**unlock (X);**	T1 followed by T2
write_lock (X);	**Write_lock (Y);**	X=50, Y=80.
read_item (X);	read_item (Y);	Result of serial execution
X:=X+Y;	Y:=X+Y;	T2 followed by T1
write_item (X);	write_item (Y);	X=70, Y=50
unlock (X);	**unlock (Y);**	

The Two-Phase Locking Protocol

Two-Phase Locking Techniques: The algorithm

T1	T2	Result
read_lock (Y);		X=50; Y=50
read_item (Y);		Nonserializable because it.
unlock (Y);		violated two-phase policy.
	read_lock (X);	
	read_item (X);	
	unlock (X);	
	write_lock (Y);	
	read_item (Y);	
	Y:=X+Y;	
	write_item (Y);	
	unlock (Y);	
write_lock (X);		
read_item (X);		
X:=X+Y;		
write_item (X);		
unlock (X);		

Time

THE TWO-PHASE LOCKING PROTOCOL

Two-Phase Locking Techniques: The algorithm

T'1	T'2
read_lock (Y);	**read_lock (X);**
read_item (Y);	read_item (X);
write_lock (X);	**Write_lock (Y);**
unlock (Y);	**unlock (X);**
read_item (X);	read_item (Y);
X:=X+Y;	Y:=X+Y;
write_item (X);	write_item (Y);
unlock (X);	**unlock (Y);**

T1 and T2 follow two-phase policy

but they are subject to deadlock,
which must be dealt with.

Database Concurrency Control

Dealing with Deadlock and Starvation

Deadlock

<u>T'1</u>	<u>T'2</u>
read_lock (Y);	
read_item (Y);	
	read_lock (X);
	read_item (Y);
write_lock (X);	
(waits for X)	write_lock (Y);
	(waits for Y)

Deadlock (T'1 and T'2)

T1 and T2 did follow two-phase policy but they are deadlock

Deadlock and Starvation

Deadlock prevention

- A transaction locks all data items it refers to before it begins execution.
- This way of locking prevents deadlock since a transaction never waits for a data item.
- The conservative two-phase locking uses this approach.

Deadlock and Starvation

Deadlock detection and resolution

- Deadlocks are allowed to happen.
- The scheduler maintains a wait-for-graph for detecting cycle.
- If a cycle exists, then one transaction involved in the cycle is selected (victim) and rolled-back.
- A wait-for-graph is created using the lock table.

Deadlock and Starvation

Deadlock avoidance

- There are many variations of two-phase locking algorithm.
- Some avoid deadlock by not letting the cycle to complete.
  - That is as soon as the algorithm discovers that blocking a transaction is likely to create a cycle, it rolls back the transaction.
  - Wound-Wait and Wait-Die algorithms use timestamps to avoid deadlocks by rolling-back victim.
- Starvation

T-SQL AND TRANSACTIONS

SQL has following transaction modes.

- Autocommit transactions
  - Each individual SQL statement = transaction.

- Explicit transactions

  BEGIN TRANSACTION

  [SQL statements]

  COMMIT or ROLLBACK

Transaction Support in TSQL

- DECLARE @intErrorCode INT
- BEGIN TRAN
- UPDATE Department
- SET Mgr_ssn = 123456789
- WHERE DNumber = 1
- SELECT @intErrorCode = @@ERROR
- IF (@intErrorCode <> 0) ROLLBACK TRAN
- UPDATE Department
- SET Mgr_start_date = '1981-06-19'
- WHERE Dnumber = 1
- SELECT @intErrorCode = @@ERROR
- IF (@intErrorCode <> 0) ROLLBACK TRAN
- COMMIT TRAN

Transaction Support in TSQL

Table 21.1 Possible Violations Based on Isolation Levels as Defined in SQL

	Type of Violation		
Isolation Level	**Dirty Read**	**Nonrepeatable Read**	**Phantom**
READ UNCOMMITTED	Yes	Yes	Yes
READ COMMITTED	No	Yes	Yes
REPEATABLE READ	No	No	Yes
SERIALIZABLE	No	No	No

Transaction Support in TSQL

1. "Dirty reads"

   SET TRANSACTION ISOLATION LEVEL READ UNCOMMITTED

2. "Committed reads"

   SET TRANSACTION ISOLATION LEVEL READ COMMITTED

3. "Repeatable reads"

   SET TRANSACTION ISOLATION LEVEL REPEATABLE READ

4. Serializable transactions (default):

   SET TRANSACTION ISOLATION LEVEL SERIALIZABLE

Transaction Support in SQL

Potential problem with lower isolation levels:

- **Dirty Read**:
  - Reading a value that was written by a failed transaction.

- **Nonrepeatable Read**:
  - Allowing another transaction to write a new value between multiple reads of one transaction.
    - A transaction T1 reads a given value from a table.
    - If another transaction T2 later updates that value and T1 reads that value again, T1 will see a different value.

Transaction Support in SQL

- Potential problem with lower isolation levels (contd.):
  - Phantoms:
    - New rows being read using the same read with a condition.
      - A transaction T1 may read a set of rows from a table, perhaps based on some condition specified in the SQL WHERE clause.
      - Now suppose that a transaction T2 inserts a new row that also satisfies the WHERE clause condition of T1, into the table used by T1.
      - If T1 is repeated, then T1 will see a row that previously did not exist, called a phantom.