DATABASE MANAGEMENT SYSTEMS

Couse code:CSC403

Prof. Juhi Janjua

Module 6 Transactions Management and Concurrency and Recovery

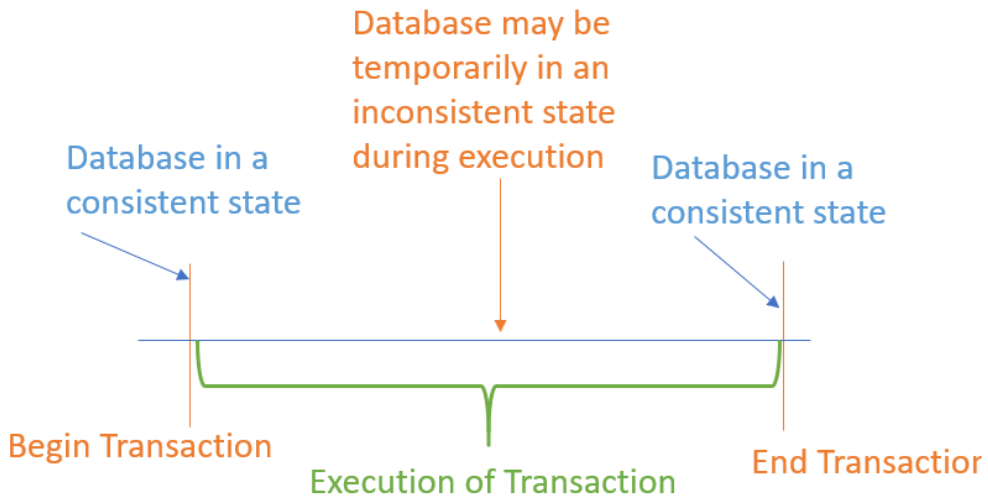
- Transaction concept
- Transaction states
- ACID properties
- Transaction Control Commands
- Concurrent Executions
- Serializability-Conflict and View
- Concurrency Control: Lock-based, Timestamp-based protocols
- Recovery System: Log based recovery, Deadlock handling

Transaction

- The transaction is a set of logically related operation. It contains a group of tasks.
- A transaction is an action or series of actions. It is performed by a single user to perform operations for accessing the contents of the database.

Transaction...

- All types of database access operation which are held between the beginning and end transaction statements are considered as a single logical transaction in DBMS.
- During the transaction the database is inconsistent. Only once the database is committed the state is changed from one consistent state to another.



Operations of transaction

Following are the main operations of transaction:

- **read_item(X):** Read operation is used to read the value of X from the database and stores it in a buffer in main memory.
- **write_item(X):** Write operation is used to write the value back to the database from the buffer.

read_item(X) command steps

read_item(X) command includes the following steps:

- Find the address of the disk block that contains item X.
- Copy that disk block into a buffer in main memory (if that disk block is not already in some main memory buffer).
- Copy item X from the buffer to the program variable named X.

write_item(X) command steps

write_item(X) command includes the following steps:

- Find the address of the disk block that contains item X.
- Copy that disk block into a buffer in main memory (if that disk block is not already in some main memory buffer).
- Copy item X from the program variable named X into its correct location in the buffer.
- Store the updated block from the buffer back to disk (either immediately or at some later point in time).

Two sample transactions

- We can see two sample transactions:
  - (a) Transaction T1
  - (b) Transaction T2

(a) T_1

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

(b) T_2

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

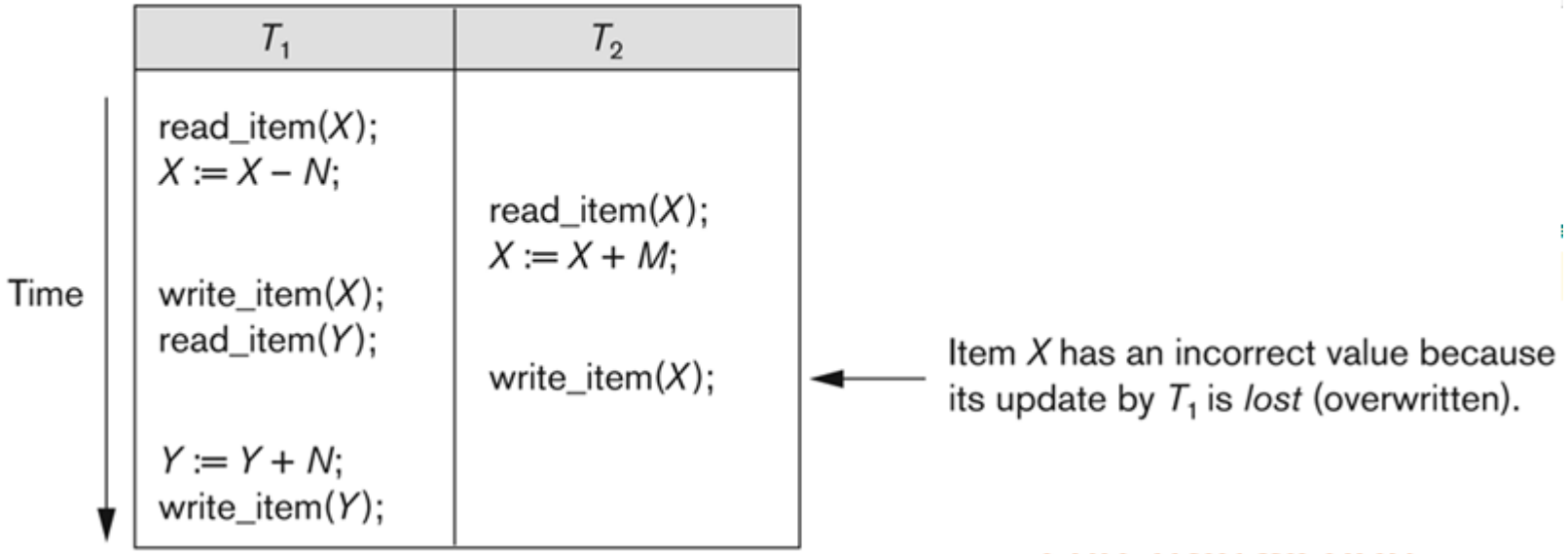
Why Concurrency Control is needed?

- The Lost Update Problem
  - This occurs when two transactions that access the same database items have their operations interleaved in a way that makes the value of some database item incorrect.
- The Temporary Update (or Dirty Read) Problem
  - This occurs when one transaction updates a database item and then the transaction fails for some reason.
  - The updated item is accessed by another transaction before it is changed back to its original value.
- The Incorrect Summary Problem
  - If one transaction is calculating an aggregate summary function on a number of records while other transactions are updating some of these records, the aggregate function may calculate some values before they are updated and others after they are updated.

Concurrent execution is uncontrolled: (a) The lost update problem.

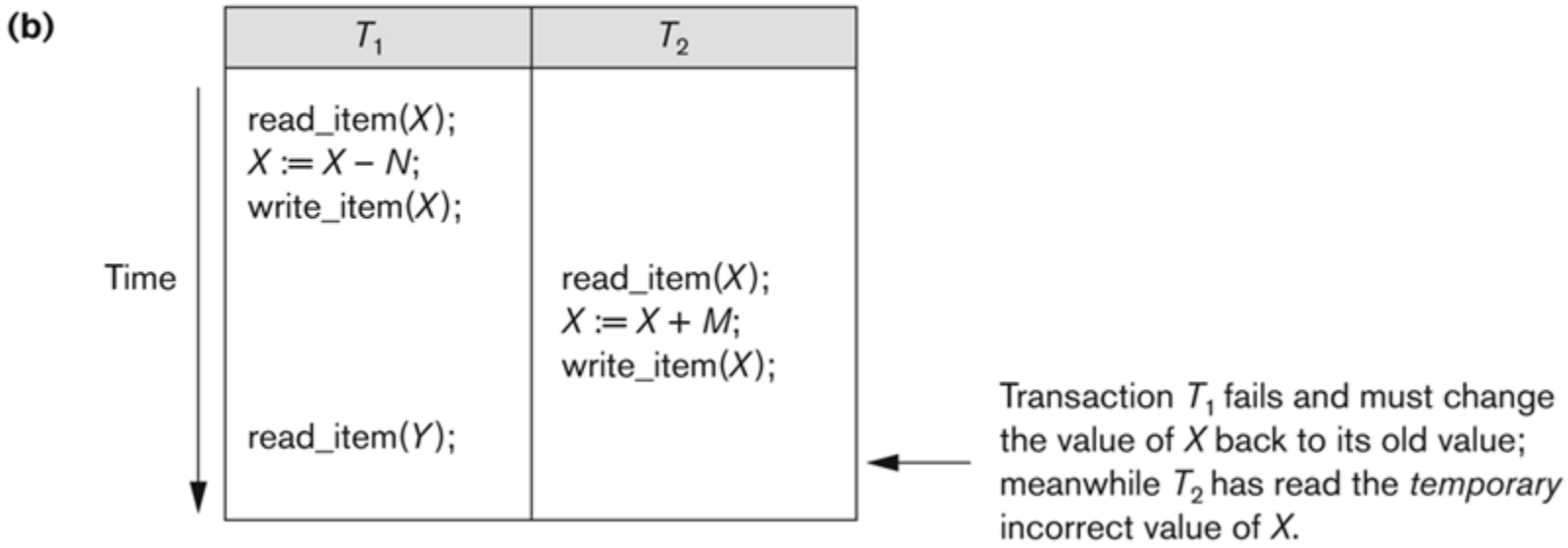
Some problems that occur when concurrent execution is uncontrolled. (a) The lost update problem. (b) The temporary update problem. (c) The incorrect summary problem.

(a)



Concurrent execution is uncontrolled: (b) The temporary update problem.

Some problems that occur when concurrent execution is uncontrolled. (a) The lost update problem. (b) The temporary update problem. (c) The incorrect summary problem.



Concurrent execution is uncontrolled: (c) The incorrect summary problem.

Some problems that occur when concurrent execution is uncontrolled. (a) The lost update problem. (b) The temporary update problem. (c) The incorrect summary problem.

(c)

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).

What causes a Transaction to fail?

1. A computer failure (system crash):

A hardware or software error occurs in the computer system during transaction execution. If the hardware crashes, the contents of the computer's internal memory may be lost.

2. A transaction or system error:

- Some operation in the transaction may cause it to fail, such as integer overflow or division by zero.
- Transaction failure may also occur because of erroneous parameter values or because of a logical programming error.
- In addition, the user may interrupt the transaction during its execution.

What causes a Transaction to fail?

3. Local errors or exception conditions detected by the transaction:
- Certain conditions necessitate cancellation of the transaction.
- For example, data for the transaction may not be found.
- A condition, such as insufficient account balance in a banking database, may cause a transaction, such as a fund withdrawal from that account, to be canceled.
- A programmed abort in the transaction causes it to fail.

4. Concurrency control enforcement:

The concurrency control method may decide to abort the transaction, to be restarted later, because it violates serializability or because several transactions are in a state of deadlock.

What causes a Transaction to fail?

5. Disk failure:

Some disk blocks may lose their data because of a read or write malfunction or because of a disk read/write head crash. This may happen during a read or a write operation of the transaction.

6. Physical problems and catastrophes:

This refers to an endless list of problems that includes power or air conditioning failure, fire, theft, sabotage, overwriting disks or tapes by mistake, and mounting of a wrong tape by the operator.

Transaction 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.
- Recovery manager keeps track of the following operations:
  - **begin_transaction:** This marks the beginning of transaction execution.
  - **read or write:** These specify read or write operations on the database items that are executed as part of a transaction.

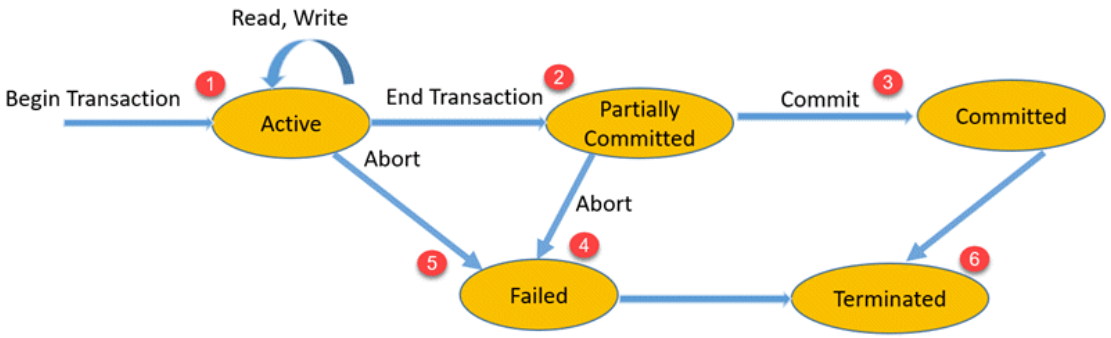
Transaction concepts...

- Recovery manager keeps track of the following operations:
  - **end_transaction:** This specifies that read and write transaction operations have ended and marks the end limit of transaction execution.
  - At this point it may be necessary to check whether the changes introduced by the transaction can be permanently applied to the database or whether the transaction has to be aborted because it violates concurrency control or for some other reason.
  - **commit_transaction**: This signals a successful end of the transaction so that any changes (updates) executed by the transaction can be safely committed to the database and will not be undone.
  - **rollback (or abort):** This signals that the transaction has ended unsuccessfully, so that any changes or effects that the transaction may have applied to the database must be undone

Transaction Concepts...

- Recovery techniques use the following operators:
  - **undo**: Similar to rollback except that it applies to a single operation rather than to a whole transaction.
  - **redo**: This specifies that certain *transaction operations* must be *redone* to ensure that all the operations of a committed transaction have been applied successfully to the database.

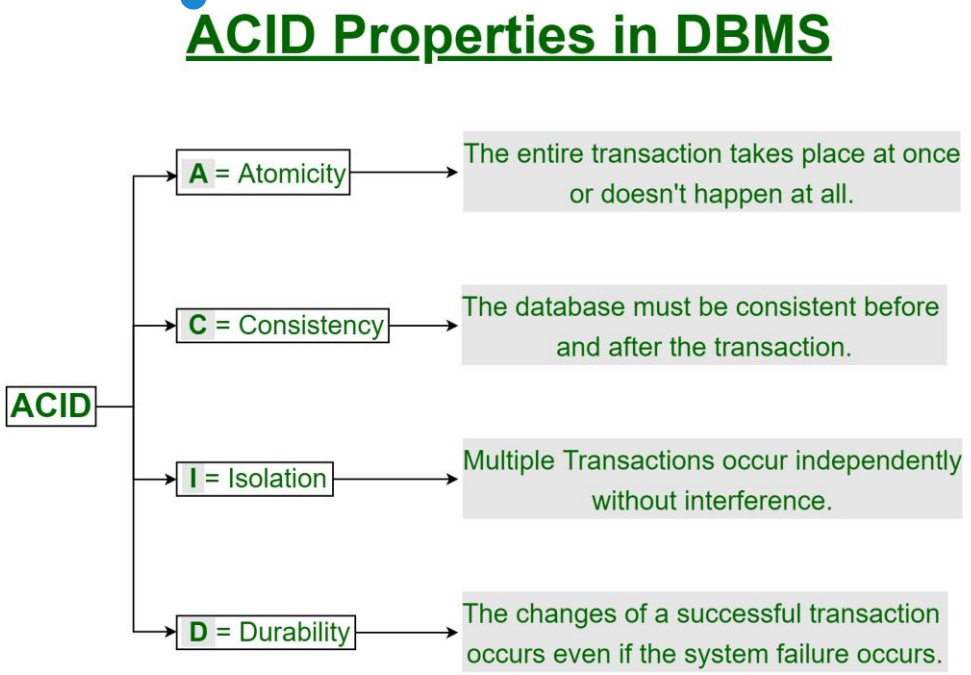
State of transactions



State	Transaction types
Active State	A transaction enters into an active state when the execution process begins. During this state read or write operations can be performed.
Partially Committed	A transaction goes into the partially committed state after the end of a transaction.
Committed State	When the transaction is committed to state, it has already completed its execution successfully. Moreover, all of its changes are recorded to the database permanently.
Failed State	A transaction considers failed when any one of the checks fails or if the transaction is aborted while it is in the active state.
Terminated State	State of transaction reaches terminated state when certain transactions which are leaving the system can't be restarted.

ACID properties

- A transaction is a very small unit of a program and it may contain several low-level tasks.
- A transaction in a database system must maintain **A**tomicity, **C**onsistency, **I**solation, and **D**urability – commonly known as **ACID properties** – in order to ensure accuracy, completeness, and data integrity.



Atomicity

- By this, we mean that either the entire transaction takes place at once or doesn't happen at all. There is no midway i.e. transactions do not occur partially.
- It involves the following two operations.
  - **Abort:** If a transaction aborts, changes made to database are not visible.
  - **Commit:** If a transaction commits, changes made are visible.
- Atomicity is also known as the **'All or nothing rule'.**

Atomicity..

- Consider the following transaction T consisting of T1 and T2: Transfer of 100 from account X to account Y.

Before: X : 500	Y: 200
Transaction T	
T1	**T2**
Read (X) X: = X − 100 Write (X)	Read (Y) Y: = Y + 100 Write (Y)
After: X : 400	Y : 300

- If the transaction fails after completion of T1 but before completion of T2, then amount has been deducted from X but not added to Y. This results in an inconsistent database state

Consistency

- This means that integrity constraints must be maintained so that the database is consistent before and after the transaction. It refers to the correctness of a database. Referring to the earlier example,
- The total amount before and after the transaction must be maintained.

  Total **before T** occurs = 500 + 200 = **700**.

  Total **after T** occurs = 400 + 300 = **700**.
- Therefore, database is consistent.
- Inconsistency occurs in case T1 completes but T2 fails. As a result T is incomplete.

Isolation

- This property ensures that multiple transactions can occur concurrently without leading to the inconsistency of database state.
- Transactions occur independently without interference.
- Changes occurring in a particular transaction will not be visible to any other transaction until that change in that transaction is written to memory or has been committed.

Isolation..

- Let X= 500, Y = 500.
- Consider two transactions T and T".

T	T"
Read (X) X: = X*100 Write (X) Read (Y) Y: = Y – 50 Write	Read (X) Read (Y) Z: = X + Y Write (Z)

- Suppose T has been executed till Read (Y) and then T" starts. As a result , interleaving of operations takes place due to which T" reads correct value of X but incorrect value of Y and sum computed by

T": (X+Y = 50, 000+500=50, 500)

is thus not consistent with the sum at end of transaction:

T: (X+Y = 50, 000 + 450 = 50, 450).

- Hence, transactions must take place in isolation and changes should be visible only after they have been made to the main memory.

Durability

- This property ensures that once the transaction has completed execution, the updates and modifications to the database are stored in and written to disk and they persist even if a system failure occurs.
- These updates now become permanent and are stored in non-volatile memory.
- The effects of the transaction, thus, are never lost.

Transaction control command

- These are used to manage the changes made to the data in a table by DML statements.
- It also allows statements to be grouped together into logical transactions.
- We have following Transaction Control Language (TCL) commands:
  - COMMIT
  - SAVEPOINT
  - ROLLBACK

The COMMIT command

- COMMIT command is used to **permanently save** any transaction into the database.
- When we use any DML command like INSERT, UPDATE or DELETE, the changes made by these commands are not permanent, until the current session is closed, the changes made by these commands can be rolled back.
- To avoid that, we use the COMMIT command to mark the changes as permanent.
- Following is commit command's syntax,

```
COMMIT
```

The **SAVEPOINT** command

- SAVEPOINT command is used to **temporarily save** a transaction so that you can rollback to that point whenever required.
- Following is savepoint command's syntax,

  SAVEPOINT savepoint_name
- In short, using this command we can name the different states of our data in any table and then **rollback to that state** using the ROLLBACK command whenever required.

The ROLLBACK command

- This command restores the database to last commited state.
- It is also used with SAVEPOINT command to jump to a savepoint in an ongoing transaction.
- If we have used the UPDATE command to make some changes into the database, and realise that those changes were not required, then we can use the ROLLBACK command to rollback those changes, if they were not commited using the COMMIT command.
- Following is rollback command's syntax,

```
ROLLBACK
    or
ROLLBACK TO savepoint_name
```

Example

Following is the table **class**,

id	name
1	Abhi
2	Adam
4	Alex

Lets use some SQL queries on the above table and see the results.

```
INSERT INTO class VALUES(5, 'Rahul');

COMMIT;

UPDATE class SET name = 'Abhijit' WHERE id = '5';

SAVEPOINT A;

INSERT INTO class VALUES(6, 'Chris');

SAVEPOINT B;

INSERT INTO class VALUES(7, 'Bravo');

SAVEPOINT C;

SELECT * FROM class;
```

Example...

The resultant table will look like,

id	name
1	Abhi
2	Adam
4	Alex
5	Abhijit
6	Chris
7	Bravo

Query:

```
ROLLBACK TO B;

SELECT * FROM class;
```

Result:

id	name
1	Abhi
2	Adam
4	Alex
5	Abhijit
6	Chris

Example...

Query:

```
ROLLBACK TO A;

SELECT * FROM class;
```

Result:

id	name
1	Abhi
2	Adam
4	Alex
5	Abhijit

Concurrent execution

- In a multi-user system, multiple users can access and use the same database at one time, which is known as the concurrent execution of the database.
- It means that the same database is executed simultaneously on a multi-user system by different users.
- The thing is that the simultaneous execution that is performed should be done in an **interleaved manner**, and no operation should affect the other executing operations, thus maintaining the consistency of the database.

Concurrency execution...

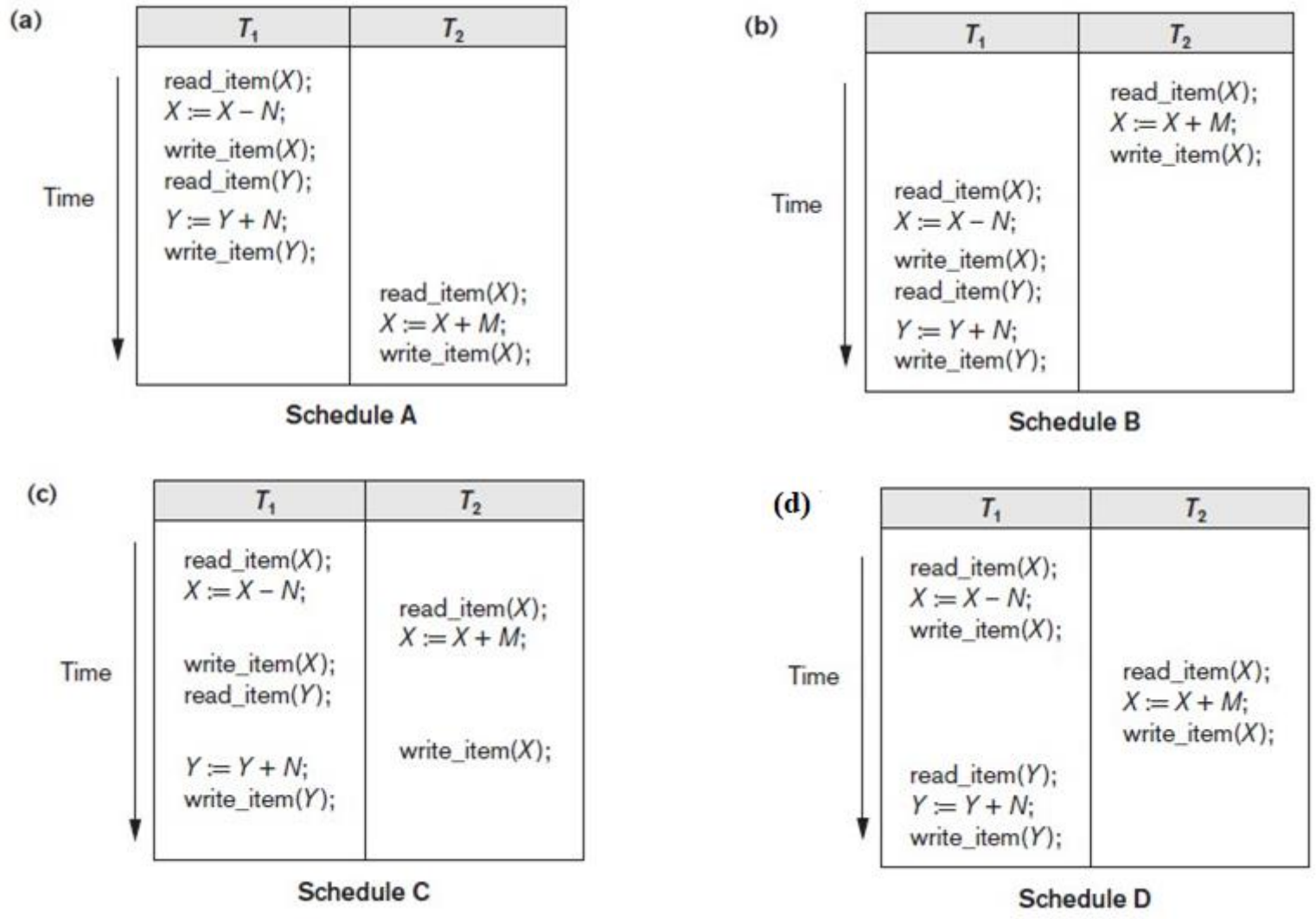
- Thus, on making the concurrent execution of the transaction operations, there occur several challenging problems that need to be solved.
- Some of the problems we have discussed already:
  - The Lost Update Problem
  - The Temporary Update (or Dirty Read) Problem
  - The Incorrect Summary Problem

Serializibility

- When multiple transactions are running concurrently then there is a possibility that the database may be left in an inconsistent state.
- A **serializable schedule** is the one that always leaves the database in consistent state.
- **Serializability** is a concept that helps us to check which schedules are serializable.

Serializibility...

- A **serial schedule** is always a serializable schedule because in serial schedule, a transaction only starts when the other transaction finished execution.
- However, a non-serial schedule needs to be checked for Serializability.
- A **non-serial schedule** of n number of transactions is said to be serializable schedule, if it is equivalent to the serial schedule of those n transactions.
- A serial schedule doesn't allow concurrency, only one transaction executes at a time and the other starts when the already running transaction finished.



Examples of serial and nonserial schedules involving transactions *T*1 and *T*2 (a) Serial schedule A: *T*1 followed by *T*2 (b) Serial schedule B: *T*2 followed by *T*1 (c), (d) Two nonserial schedules C and D with interleaving of operations

Characterizing Schedules Based on Serializability

Problem with serial schedules

- Limit concurrency by prohibiting interleaving of operations
- Unacceptable in practice

Solution: determine which non serial schedules are equivalent to a serial schedule and allow those to occur

Serializable schedule of n transactions

- Equivalent to some serial schedule of same n transactions

Characterizing Schedules Based on Serializability...

Result equivalent schedules

- Produce the same final state of the database
  - May be accidental
- Cannot be used alone to define equivalence of schedules.

S_1

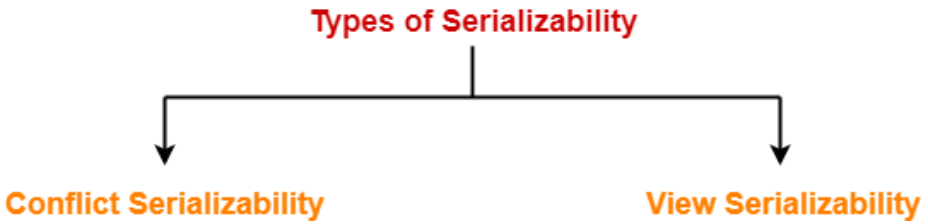
read_item(X);
$X := X + 10$;
write_item(X);

S_2
read_item(X); $X := X * 1.1$; write_item (X);

Two schedules that are result equivalent for the initial value of $X = 100$ but are not result equivalent in general

Types of Serializibility

Two definitions of equivalence of schedules are generally used: **conflict equivalence** and **view equivalence.**



<u>Conflict equivalence:</u>

- Two schedules are said to be conflict equivalent if one schedule can be converted into other schedule after **swapping non-conflicting operations.**
- Schedule S is **serializable** if it is conflict equivalent to some serial schedule S'.

Conflicting operations

- Two operations are said to be in conflict, if they satisfy all the following three conditions:

1. Both the operations should belong to **different transactions.**
2. Both the operations are working on **same data item.**
3. At least **one of the operation is a write operation.**

Conflicting operations...

Example:
W(X) of T1 & R(X) of T2
Conflict
W(X) of T1 & W(X) of T2
Conflict
W(X) of T1 & W(Y) of T2
Non-Conflict
R(X) of T1 & R(X) of T2
Non-Conflict
W(X) of T1 & R(X) of T2
Non-Conflict

Example1: Consider a schedule, let's check for conflict serializability

T1	T2
R(A)	
R(B)	
	R(A)
	R(B)
	W(B)
W(A)	

we cannot swap R(A) of T2 & W(A) of T1 operations because they are conflicting operations, thus we can say that this given schedule is **not Conflict Serializable.**

Example2: Consider a schedule, let's check for conflict serializability

```
T1          T2
-----       ------
R(A)
            R(A)
            R(B)
            W(B)
R(B)
W(A)
```

Let's swap non conflicting operations,

After swapping R(A) of T1 and R(A) of T2 we get:

```
T1          T2
-----       ------
            R(A)
R(A)
            R(B)
            W(B)
R(B)
W(A)
```

After swapping R(A) of T1 and R(B) of T2 we get:

```
T1          T2
-----       ------
            R(A)
            R(B)
R(A)
            W(B)
R(B)
W(A)
```

After swapping R(A) of T1 and W(B) of T2 we get:

```
T1          T2
-----       ------
            R(A)
            R(B)
            W(B)
R(A)
R(B)
W(A)
```

Given schedule is conflict serializable.

Precedence Graph For Testing Conflict Serializability

Testing for serializability of a schedule

1. For each transaction T_i participating in schedule S, create a node labeled T_i in the precedence graph.
2. For each case in S where T_j executes a read_item(X) after T_i executes a write_item(X), create an edge ($T_i \rightarrow T_j$) in the precedence graph.
3. For each case in S where T_j executes a write_item(X) after T_i executes a read_item(X), create an edge ($T_i \rightarrow T_j$) in the precedence graph.
4. For each case in S where T_j executes a write_item(X) after T_i executes a write_item(X), create an edge ($T_i \rightarrow T_j$) in the precedence graph.
5. The schedule S is serializable if and only if the precedence graph has no cycles.

Algorithm Testing conflict serializability of a schedule S

(a)

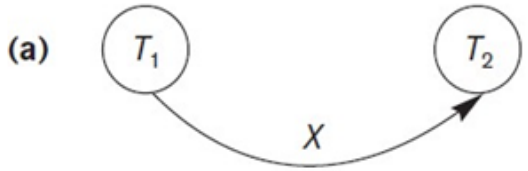
T_1	T_2

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);

Time

Schedule A



(b)

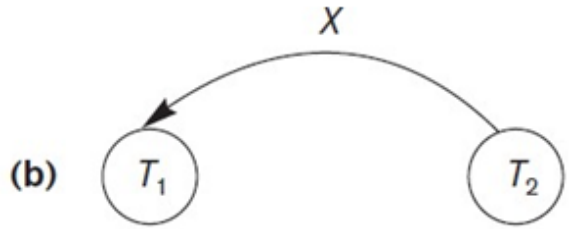
T_1	T_2

	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);	

Time

Schedule B

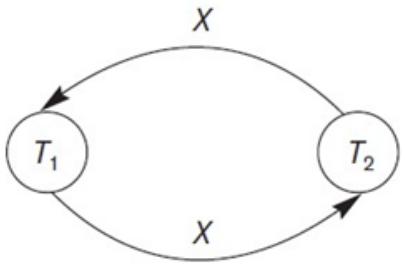


(c)

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);	

Time

Schedule C



(d)

T_1	T_2

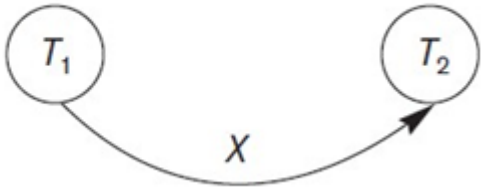
read_item(X);
$X := X - N$;
write_item(X);	

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

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

Time

Schedule D



Constructing the precedence graphs for schedules A to D from given schedules to test for conflict serializability (a) Precedence graph for serial schedule A (b) Precedence graph for serial schedule B (c) Precedence graph for schedule C (not serializable) (d) Precedence graph for schedule D (serializable, equivalent to schedule A)

View Serializability

- A schedule is said to be **View-Serializable** if it is **view equivalent to a Serial Schedule** (where no interleaving of transactions is possible).
- Two schedules S1 and S2 are said to be view equivalent if they satisfy the following conditions:

1. Initial Read
2. Final Write
3. Update Read

Initial read

- An initial read of both schedules must be the same. Suppose two schedule S1 and S2. In schedule S1, if a transaction T1 is reading the data item A, then in S2, transaction T1 should also read A.

T1	T2
Read(A)	
	Write(A)

Schedule S1

T1	T2
	Write(A)
Read(A)	

Schedule S2

- Note: data item X can be read multiple times in a schedule but the first read operation on X is called the initial read.

Final write

- A final write must be the same between both the schedules. In schedule S1, if a transaction T1 updates A at last then in S2, final writes operations should also be done by T1.

T1	T2	T3
Write(A)	Read(A)	Write(A)

Schedule S1

T1	T2	T3
Write(A)	Read(A)	Write(A)

Schedule S2

Update read

- In schedule S1, if Ti is reading A which is updated by Tj then in S2 also, Ti should read A which is updated by Tj.

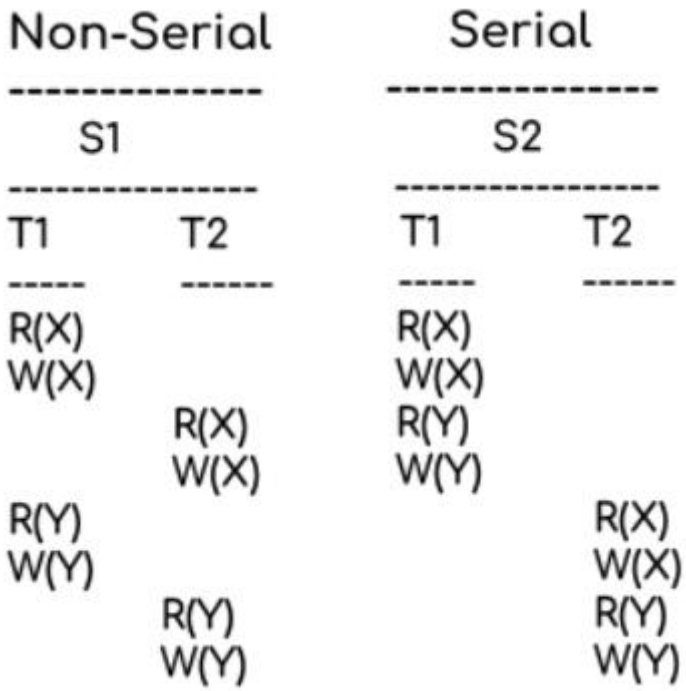
T1	T2	T3
Write(A)		
	Write(A)	
		Read(A)

Schedule S1

T1	T2	T3
	Write(A)	
Write(A)		
		Read(A)

Schedule S2

Example



Initial Read:

- In schedule S1, transaction T1 first reads the data item X. In S2 also transaction T1 first reads the data item X.
- Initial read condition is satisfied in S1 & S2.

Final Write:

- In schedule S1, the final write operation on Y is done by transaction T2. In S2 also transaction T2 performs the final write on Y.
- Final write condition is satisfied in S1 & S2.

Update Read:

- In S1, transaction T2 reads the value of X, written by T1. In S2, the same transaction T2 reads the X after it is written by T1.
- Update Read condition is satisfied in S1 & S2.

Concurrency Control: Lock-Based Protocols

- A lock is a mechanism to control concurrent access to a data item
- Data items can be locked in two modes :
  1. *exclusive (X) mode*. Data item can be both read as well as written. X-lock is requested using **lock-X** instruction.
  2. *shared (S) mode*. Data item can only be read. S-lock is requested using **lock-S** instruction.
- Lock requests are made to the concurrency-control manager by the programmer. Transaction can proceed only after request is granted.

Lock-Based Protocols (Cont.)

- Lock-compatibility matrix

	S	X
S	true	false
X	false	false

- A transaction may be granted a lock on an item if the requested lock is compatible with locks already held on the item by other transactions
- Any number of transactions can hold shared locks on an item,
  - But if any transaction holds an exclusive on the item no other transaction may hold any lock on the item.
- If a lock cannot be granted, the requesting transaction is made to wait till all incompatible locks held by other transactions have been released. The lock is then granted.

Lock-Based Protocols (Cont.)

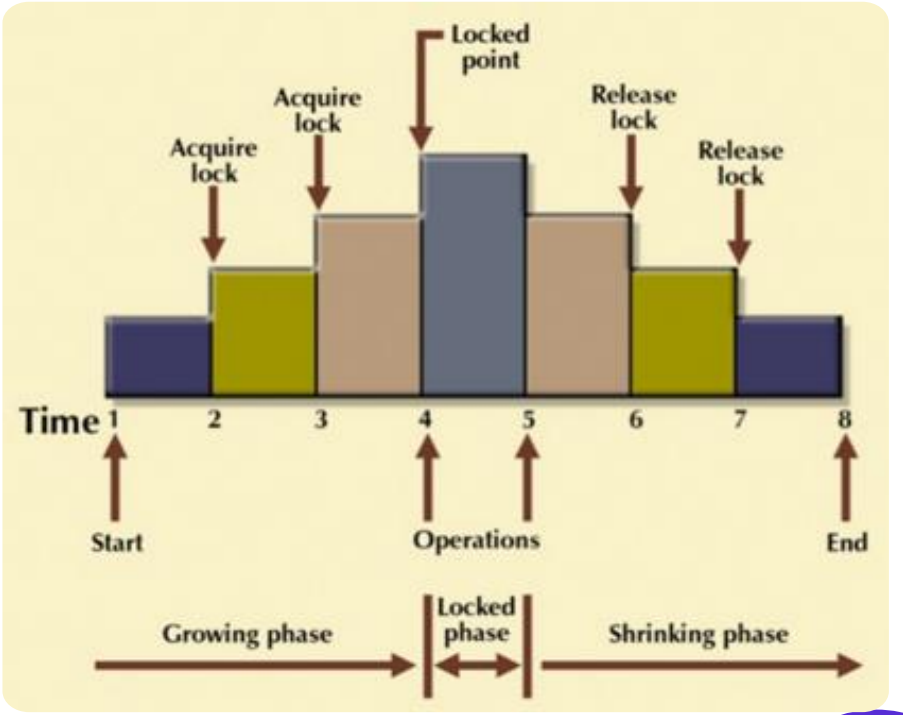
- Example of a transaction performing locking:

  T_2: lock-S(*A*);
  read (*A*);
  unlock(*A*);
  lock-S(*B*);
  read (*B*);
  unlock(*B*);
  display(*A*+*B*)

- Locking as above is not sufficient to guarantee serializability – if *A* and *B* get updated in-between the read of *A* and *B*, the displayed sum would be wrong.
- A locking protocol is a set of rules followed by all transactions while requesting and releasing locks. Locking protocols restrict the set of possible schedules.

The Two-Phase Locking Protocol

- This protocol 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
- The protocol assures serializability. It can be proved that the transactions can be serialized in the order of their **lock points** (i.e., the point where a transaction acquired its final lock).



Lock Conversions

- Two-phase locking with lock conversions:
  - First Phase:
    - can acquire a lock-S on item
    - can acquire a lock-X on item
    - can convert a lock-S to a lock-X (upgrade)
  - Second Phase:
    - can release a lock-S
    - can release a lock-X
    - can convert a lock-X to a lock-S (downgrade)
- This protocol assures serializability. But still relies on the programmer to insert the various locking instructions.

Automatic Acquisition of Locks

- A transaction T_i issues the standard read/write instruction, without explicit locking calls.
- The operation **read**(D) is processed as:

```
if Ti has a lock on D
  then
        read(D)
  else begin
        if necessary wait until no other
           transaction has a lock-X on D
        grant Ti a  lock-S on D;
        read(D)
       end
```

Automatic Acquisition of Locks (Cont.)

- **write**(*D*) is processed as:

```
if Ti has a lock-X on D
  then
    write(D)
  else begin
      if necessary wait until no other transaction has any lock-X on D,
      if Ti has a lock-S on D
          then
              upgrade lock on D to lock-X
          else
              grant Ti a lock-X on D
          write(D)
    end;
```

- All locks are released after commit or abort

Deadlocks

- Consider the partial schedule

T_3	T_4
lock-x (B)	
read (B)	
$B := B - 50$	
write (B)	
	lock-s (A)
	read (A)
	lock-s (B)
lock-x (A)	

- Neither T_3 nor T_4 can make progress – executing **lock-S**(B) causes T_4 to wait for T_3 to release its lock on B, while executing **lock-X**(A) causes T_3 to wait for T_4 to release its lock on A.
- Such a situation is called a deadlock.
  - To handle a deadlock one of T_3 or T_4 must be rolled back and its locks released.

Deadlocks (Cont.)

- Two-phase locking *does not* ensure freedom from deadlocks.
- In addition to deadlocks, there is a possibility of **starvation.**
- **Starvation** occurs if the concurrency control manager is badly designed. For example:
  - A transaction may be waiting for an X-lock on an item, while a sequence of other transactions request and are granted an S-lock on the same item.
  - The same transaction is repeatedly rolled back due to deadlocks.
- Concurrency control manager can be designed to prevent starvation.

Deadlocks (Cont.)

- The potential for deadlock exists in most locking protocols. Deadlocks are a necessary evil.
- When a deadlock occurs there is a possibility of cascading roll-backs.
- Cascading roll-back is possible under two-phase locking. To avoid this, follow a modified protocol called **strict two-phase locking** -- a transaction must hold all its exclusive locks till it commits/aborts.
- **Rigorous two-phase locking** is even stricter. Here, *all* locks are held till commit/abort. In this protocol transactions can be serialized in the order in which they commit.

Timestamp-Based Protocols

- Each transaction is issued a timestamp when it enters the system. If an old transaction T_i has time-stamp TS(T_i), a new transaction T_j is assigned time-stamp TS(T_j) such that TS(T_i) <TS(T_j).
- The protocol manages concurrent execution such that the time-stamps determine the serializability order.
- In order to assure such behavior, the protocol maintains for each data Q two timestamp values:
  - **W-timestamp**(Q) is the largest time-stamp of any transaction that executed **write**(Q) successfully.
  - **R-timestamp**(Q) is the largest time-stamp of any transaction that executed **read**(Q) successfully.

Timestamp-Based Protocols (Cont.)

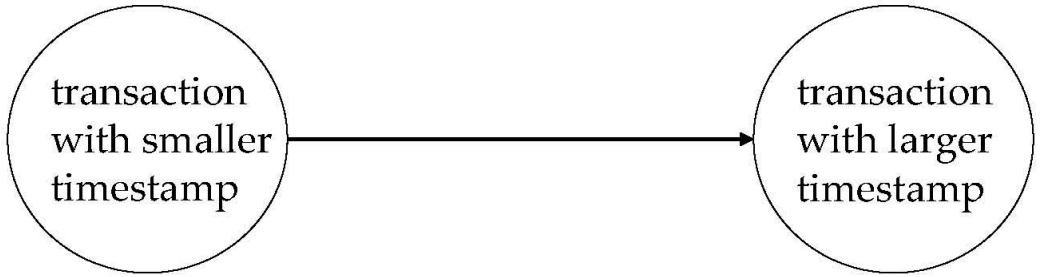
- The timestamp ordering protocol ensures that any conflicting **read** and **write** operations are executed in timestamp order.
- Suppose a transaction T_i issues a **read**(Q)
  1. If TS(T_i) ≤ **W**-timestamp(Q), then T_i needs to read a value of Q that was already overwritten.
     - Hence, the **read** operation is rejected, and T_i is rolled back.
  2. If TS(T_i) ≥ **W**-timestamp(Q), then the **read** operation is executed, and R-timestamp(Q) is set to **max**(R-timestamp(Q), TS(T_i)).

Timestamp-Based Protocols (Cont.)

- Suppose that transaction T_i issues **write**(Q).
  1. If TS(T_i) < R-timestamp(Q), then the value of Q that T_i is producing was needed previously, and the system assumed that that value would never be produced.
     - Hence, the **write** operation is rejected, and T_i is rolled back.
  2. If TS(T_i) < W-timestamp(Q), then T_i is attempting to write an obsolete value of Q.
     - Hence, this **write** operation is rejected, and T_i is rolled back.
  3. Otherwise, the **write** operation is executed, and W-timestamp(Q) is set to TS(T_i).

Correctness of Timestamp-Ordering Protocol

- The timestamp-ordering protocol guarantees serializability since all the arcs in the precedence graph are of the form:



Thus, there will be no cycles in the precedence graph

- Timestamp protocol **ensures freedom from deadlock** as no transaction ever waits.
- But the schedule may not be cascade-free, and may not even be recoverable.

Recoverability and Cascade Freedom

- Problem with timestamp-ordering protocol:
  - Suppose T_i aborts, but T_j has read a data item written by T_i
  - Then T_j must abort; if T_j had been allowed to commit earlier, the schedule is not recoverable.
  - Further, any transaction that has read a data item written by T_j must abort
  - This can lead to cascading rollback --- that is, a chain of rollbacks
- Solution 1:
  - A transaction is structured such that its writes are all performed at the end of its processing
  - All writes of a transaction form an atomic action; no transaction may execute while a transaction is being written
  - A transaction that aborts is restarted with a new timestamp
- Solution 2: Limited form of locking: wait for data to be committed before reading it
- Solution 3: Use commit dependencies to ensure recoverability

Failure Classification

- Transaction failure :
  - **Logical errors**: transaction cannot complete due to some internal error condition
  - **System errors**: the database system must terminate an active transaction due to an error condition (e.g., deadlock)
- **System crash**: a power failure or other hardware or software failure causes the system to crash.
  - Fail-stop assumption: non-volatile storage contents are assumed to not be corrupted by system crash
    - Database systems have numerous integrity checks to prevent corruption of disk data
- **Disk failure**: a head crash or similar disk failure destroys all or part of disk storage
  - Destruction is assumed to be detectable: disk drives use checksums to detect failures

Recovery Algorithms

- Consider transaction T_i that transfers \$50 from account A to account B
  - Two updates: subtract 50 from A and add 50 to B
- Transaction T_i requires updates to A and B to be output to the database.
  - A failure may occur after one of these modifications have been made but before both of them are made.
  - Modifying the database without ensuring that the transaction will commit may leave the database in an inconsistent state
  - Not modifying the database may result in lost updates if failure occurs just after transaction commits
- Recovery algorithms have two parts
  1. Actions taken during normal transaction processing to ensure enough information exists to recover from failures
  2. Actions taken after a failure to recover the database contents to a state that ensures atomicity, consistency and durability

Storage Structure

- Volatile storage:
  - does not survive system crashes
  - examples: main memory, cache memory
- Nonvolatile storage:
  - survives system crashes
  - examples: disk, tape, flash memory,
    non-volatile (battery backed up) RAM
  - but may still fail, losing data
- Stable storage:
  - a mythical form of storage that survives all failures
  - approximated by maintaining multiple copies on distinct nonvolatile media

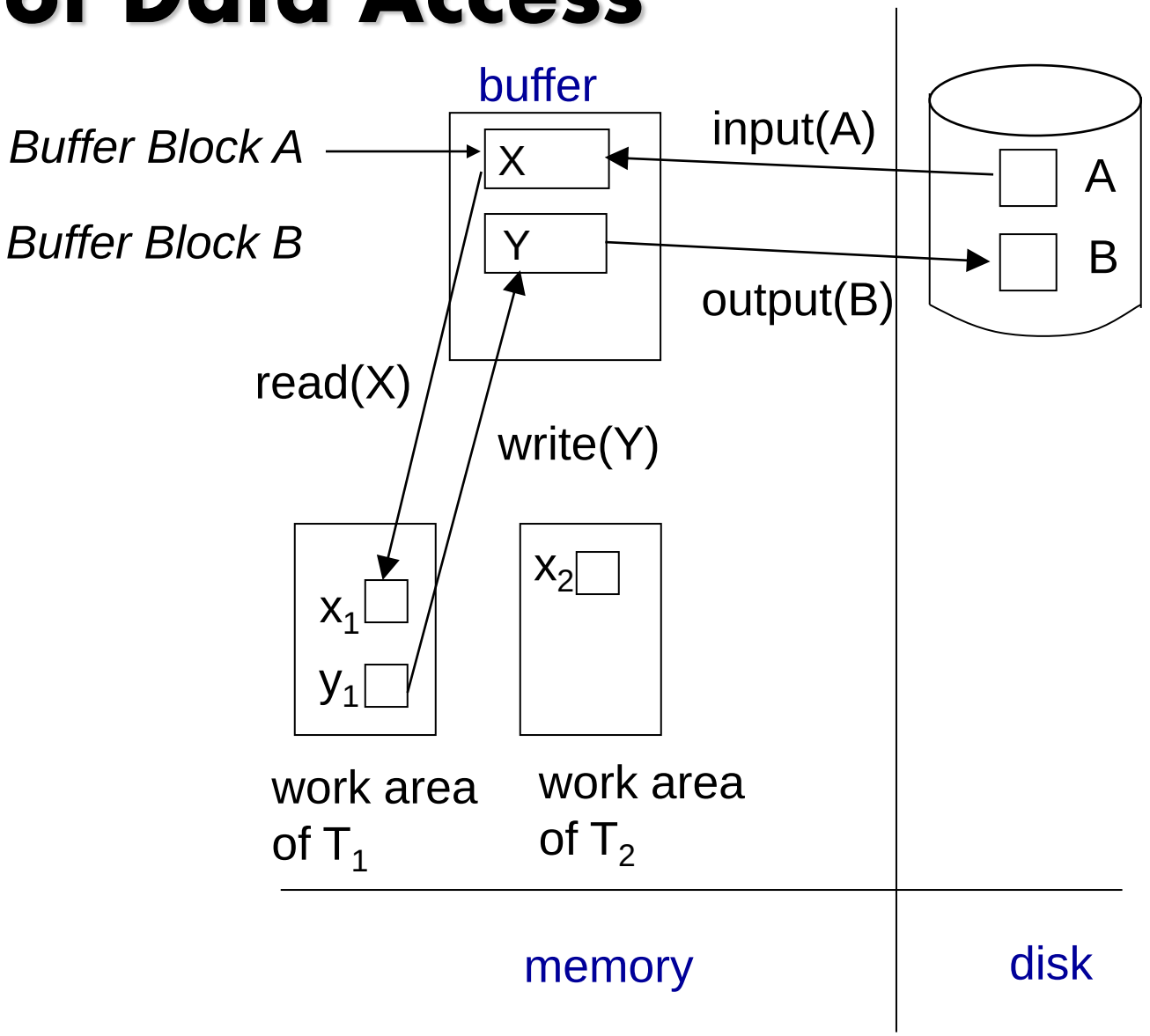
Stable-Storage Implementation

- Maintain multiple copies of each block on separate disks
  - copies can be at remote sites to protect against disasters such as fire or flooding.
- Failure during data transfer can still result in inconsistent copies: Block transfer can result in
  - Successful completion
  - Partial failure: destination block has incorrect information
  - Total failure: destination block was never updated
- Protecting storage media from failure during data transfer (one solution):
  - Execute output operation as follows (assuming two copies of each block):
    1. Write the information onto the first physical block.
    2. When the first write successfully completes, write the same information onto the second physical block.
    3. The output is completed only after the second write successfully completes.

Data Access

- Physical blocks are those blocks residing on the disk.
- Buffer blocks are the blocks residing temporarily in main memory.
- Block movements between disk and main memory are initiated through the following two operations:
  - input(*B*) transfers the physical block *B* to main memory.
  - output(*B*) transfers the buffer block *B* to the disk, and replaces the appropriate physical block there.
- We assume, for simplicity, that each data item fits in, and is stored inside, a single block.

Example of Data Access



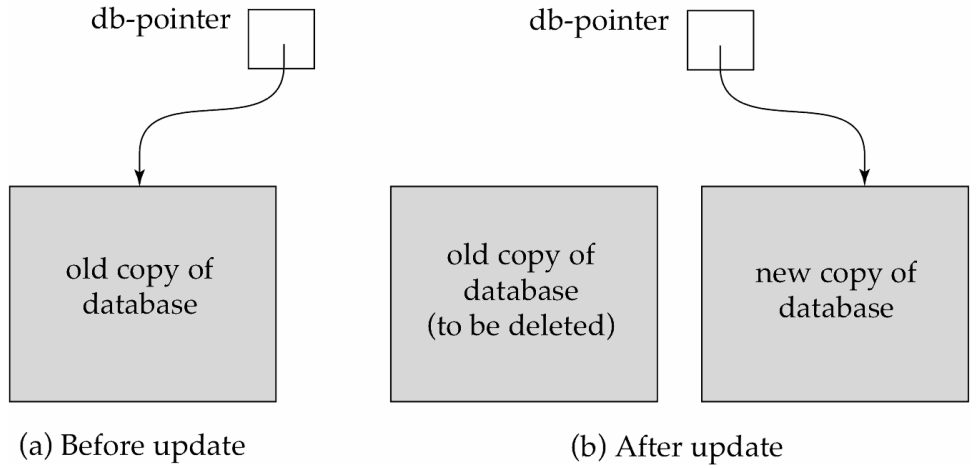
Data Access (Cont.)

- Each transaction T_i has its private work-area in which local copies of all data items accessed and updated by it are kept.
  - T_i's local copy of a data item X is called x_i.
- Transferring data items between system buffer blocks and its private work-area done by:
  - **read**(X) assigns the value of data item X to the local variable x_i.
  - **write**(X) assigns the value of local variable x_i to data item X in the buffer block.
  - **Note: output**(X) need not immediately follow **write**(X). System can perform the **output** operation when it deems fit.
- Transactions
  - Must perform **read**(X) before accessing X for the first time (subsequent reads can be from local copy)
  - **write**(X) can be executed at any time before the transaction commits

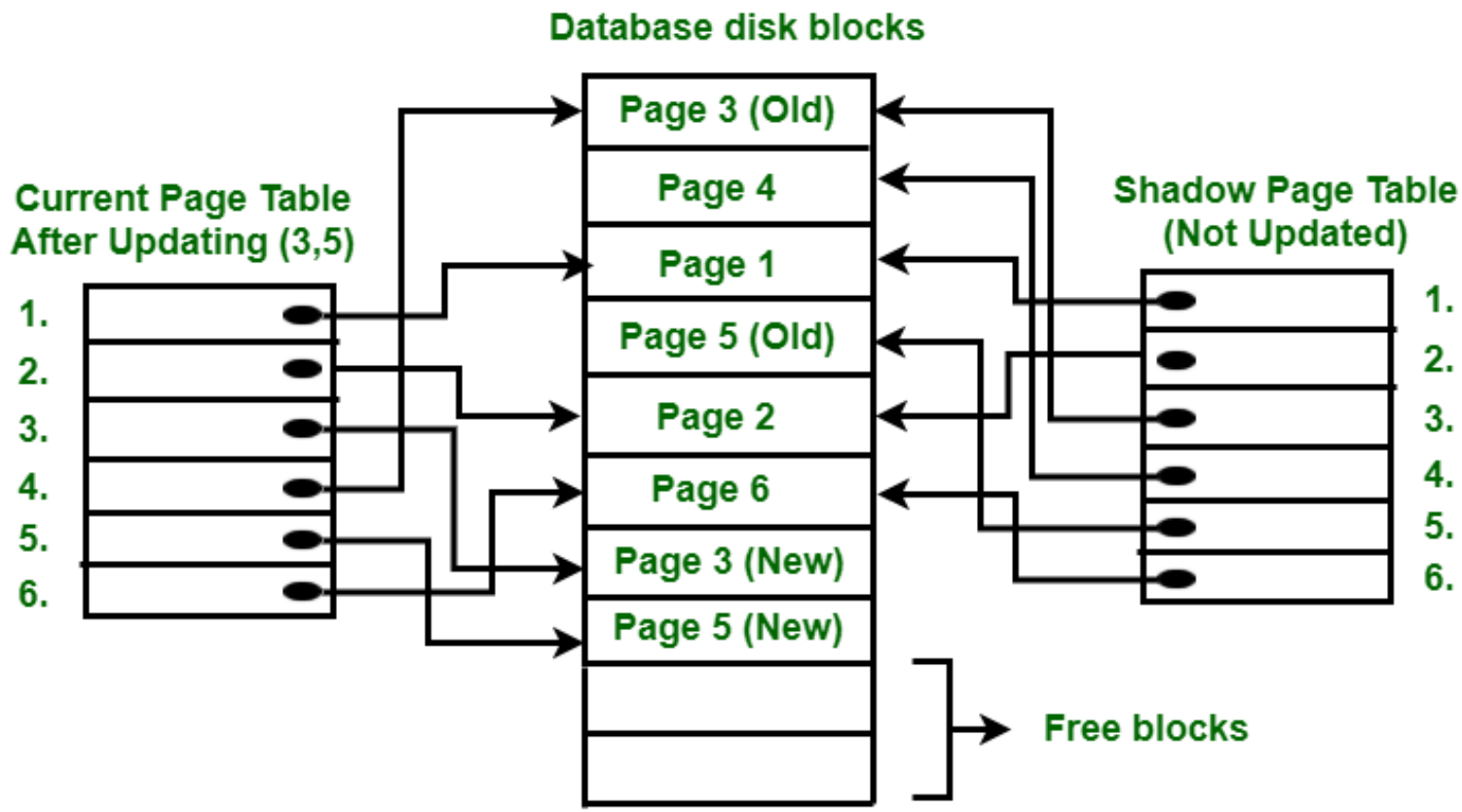
Recovery and Atomicity

- To ensure atomicity despite failures, we first output information describing the modifications to stable storage without modifying the database itself.
- We study log-based recovery mechanisms in detail
  - We first present key concepts
  - And then present the actual recovery algorithm
- Less used alternative: shadow-copy and shadow-paging

shadow-copy



Shadow Paging



Log-Based Recovery

- A log is kept on stable storage.
  - The log is a sequence of log records, and maintains a record of update activities on the database.
- When transaction T_i starts, it registers itself by writing a
  $<T_i$ **start**>log record
- *Before* T_i executes **write**(X), a log record
  $<T_i, X, V_1, V_2>$
  is written, where V_1 is the value of X before the write (the old value), and V_2 is the value to be written to X (the new value).
- When T_i finishes it last statement, the log record $<T_i$ **commit**> is written.
- Two approaches using logs
  - Deferred database modification
  - Immediate database modification

Immediate Database Modification

- The immediate-modification scheme allows updates of an uncommitted transaction to be made to the **buffer, or the disk** itself, before the transaction commits
- Update log record must be written *before* database item is written
  - We assume that the log record is output directly to stable storage
- Output of updated blocks to **stable storage** can take place at any time before or after transaction commit
- Order in which blocks are output can be different from the order in which they are written.
- The deferred-modification scheme performs updates to buffer/disk only at the time of transaction commit
  - Simplifies some aspects of recovery
  - But has overhead of storing local copy

Immediate Database Modification Example

Log	Write	Output
$<T_0$ start>		
$<T_0$, A, 1000, 950>		
$<T_0$, B, 2000, 2050		
	$A = 950$	
	$B = 2050$	
$<T_0$ commit>		
$<T_1$ start>		
$<T_1$, C, 700, 600>		
	$C = 600$	
		B_B, B_C
$<T_1$ commit>		
		B_A

B_C output before T_1 commits

B_A output after T_0 commits

- Note: B_X denotes block containing X.

Concurrency Control and Recovery

- With concurrent transactions, all transactions share a single disk buffer and a single log
  - A buffer block can have data items updated by one or more transactions
- We assume that *if a transaction T_i has modified an item, no other transaction can modify the same item until T_i has committed or aborted*
    - Otherwise how to perform undo if T1 updates A, then T2 updates A and commits, and finally T1 has to abort?
  - Can be ensured by obtaining exclusive locks on updated items and holding the locks till end of transaction (strict two-phase locking)
- Log records of different transactions may be interspersed in the log.

Undo and Redo Operations

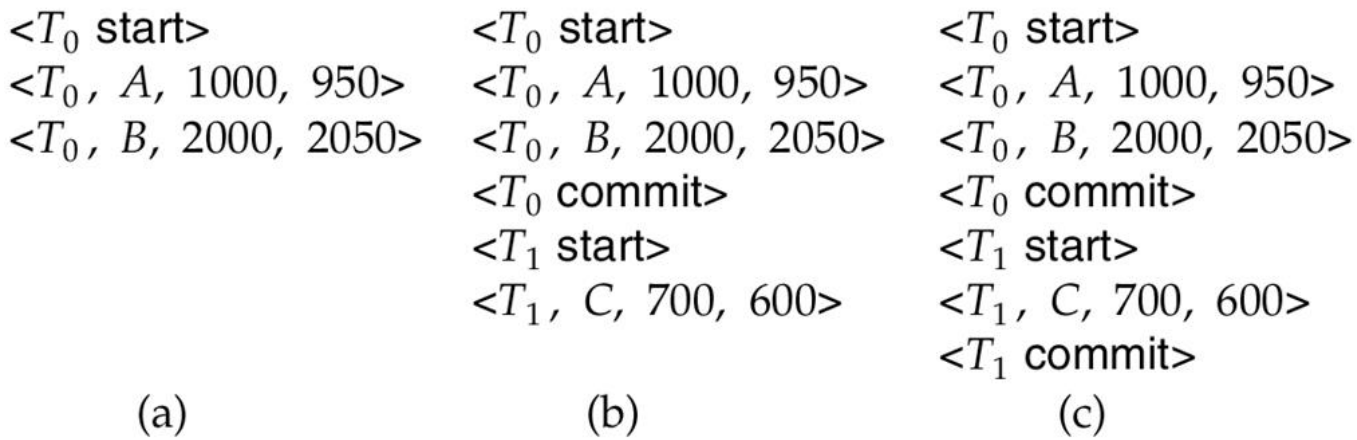
- Undo of a log record $<T_i, X, V_1, V_2>$ writes the **old** value V_1 to X
- Redo of a log record $<T_i, X, V_1, V_2>$ writes the **new** value V_2 to X
- Undo and Redo of Transactions
  - **undo**(T_i) restores the value of all data items updated by T_i to their old values, going backwards from the last log record for T_i
    - each time a data item X is restored to its old value V a special log record $<T_i, X, V>$ is written out
    - when undo of a transaction is complete, a log record $<T_i$ **abort**$>$ is written out.
  - **redo**(T_i) sets the value of all data items updated by T_i to the new values, going forward from the first log record for T_i
    - No logging is done in this case

Undo and Redo on Recovering from Failure

- When recovering after failure:
  - Transaction T_i needs to be undone if the log
    - contains the record $<T_i$ **start**$>$,
    - but does not contain either the record $<T_i$ **commit**$>$ *or* $<T_i$ **abort**$>$.
  - Transaction T_i needs to be redone if the log
    - contains the records $<T_i$ **start**$>$
    - and contains the record $<T_i$ **commit**$>$ *or* $<T_i$ **abort**$>$
- Note that If transaction T_i was undone earlier and the $<T_i$ **abort**$>$ record written to the log, and then a failure occurs, on recovery from failure T_i is redone
  - **such a redo redoes all the original actions *including the steps that restored old values*** Known as repeating history
    - Seems wasteful, but simplifies recovery greatly

Immediate DB Modification Recovery Example

Below we show the log as it appears at three instances of time.



Recovery actions in each case above are:

(a) undo (T_0): B is restored to 2000 and A to 1000, and log records $< T_0$, B, 2000>, $< T_0$, A, 1000>, $< T_0$, **abort**> are written out

(b) redo (T_0) and undo (T_1): A and B are set to 950 and 2050 and C is restored to 700. Log records $< T_1$, C, 700>, $< T_1$, **abort**> are written out.

(c) redo (T_0) and redo (T_1): A and B are set to 950 and 2050 respectively. Then C is set to 600

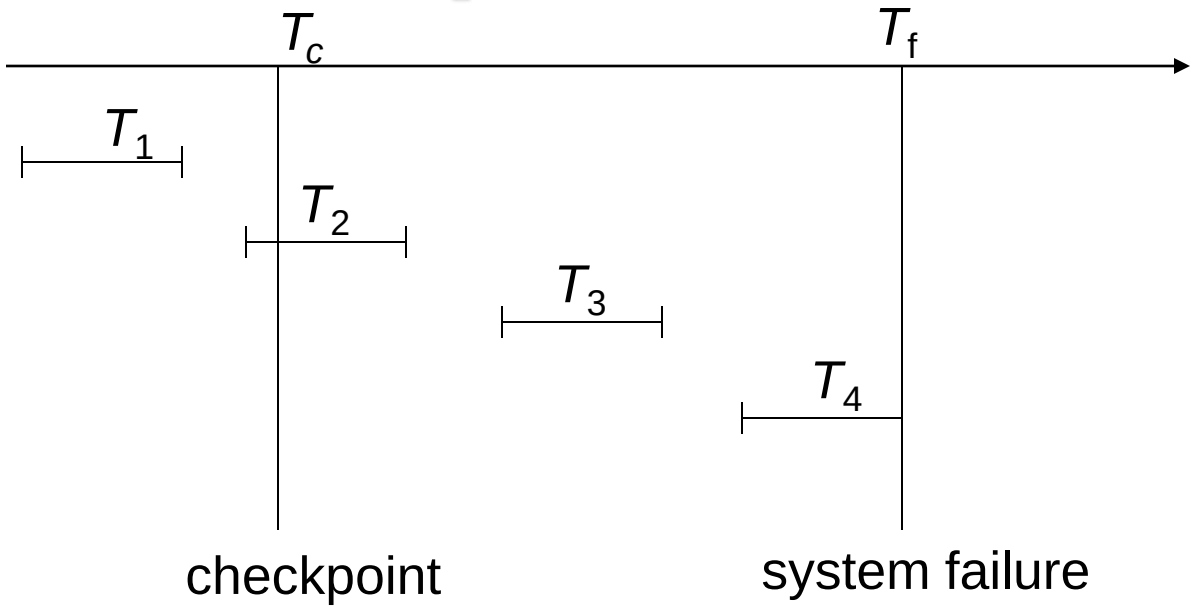
Checkpoints

- Redoing/undoing all transactions recorded in the log can be very slow
  1. processing the entire log is time-consuming if the system has run for a long time
  2. we might unnecessarily redo transactions which have already output their updates to the database.
- Streamline recovery procedure by periodically performing checkpointing
  1. Output all log records currently residing in main memory onto stable storage.
  2. Output all modified buffer blocks to the disk.
  3. Write a log record < **checkpoint** *L*> onto stable storage where *L* is a list of all transactions active at the time of checkpoint.
  - All updates are stopped while doing checkpointing

Checkpoints (Cont.)

- During recovery we need to consider only the most recent transaction T_i that started before the checkpoint, and transactions that started after T_i.
  - Scan backwards from end of log to find the most recent <**checkpoint** *L*> record
  - Only transactions that are in *L* or started after the checkpoint need to be redone or undone
  - Transactions that committed or aborted before the checkpoint already have all their updates output to stable storage.

Example of Checkpoints



- T_1 can be ignored (updates already output to disk due to checkpoint)
- T_2 and T_3 redone.
- T_4 undone

Deadlock Handling

- System is deadlocked if there is a set of transactions such that every transaction in the set is waiting for another transaction in the set.
- *Deadlock prevention* protocols ensure that the system will *never* enter into a deadlock state. Some prevention strategies :
  - Require that each transaction locks all its data items before it begins execution (predeclaration).
  - Impose partial ordering of all data items and require that a transaction can lock data items only in the order specified by the partial order.

More Deadlock Prevention Strategies

- Following schemes use transaction timestamps for the sake of deadlock prevention alone.
- wait-die scheme — non-preemptive
  - older transaction may wait for younger one to release data item. (older means smaller timestamp) Younger transactions never wait for older ones; they are rolled back instead.
  - a transaction may die several times before acquiring needed data item
- wound-wait scheme — preemptive
  - older transaction *wounds* (forces rollback) of younger transaction instead of waiting for it. Younger transactions may wait for older ones.
  - may be fewer rollbacks than *wait-die* scheme.

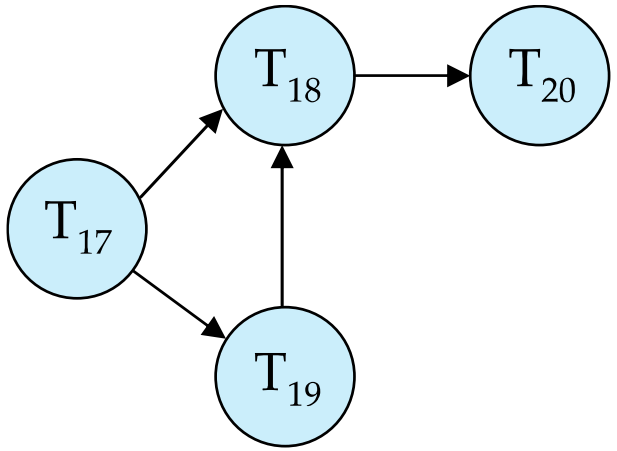
Deadlock prevention (Cont.)

- Both in *wait-die* and in *wound-wait* schemes, a rolled back transactions is restarted with its original timestamp. Older transactions thus have precedence over newer ones, and starvation is hence avoided.
- Timeout-Based Schemes:
  - Transaction waits for a lock only for a specified amount of time. If the lock has not been granted within that time, the transaction is rolled back and restarted
  - Thus, deadlocks are not possible
  - Simple to implement; but starvation is possible. Also difficult to determine good value of the timeout interval

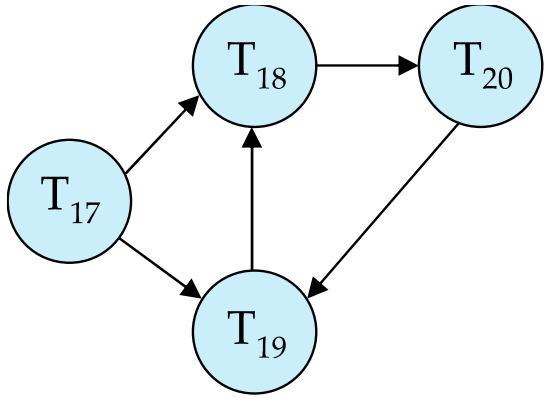
Deadlock Detection

- Deadlocks can be described as a *wait-for graph*, which consists of a pair $G = (V, E)$,
  - V is a set of vertices (all the transactions in the system)
  - E is a set of edges; each element is an ordered pair $T_i \rightarrow T_j$.
- If $T_i \rightarrow T_j$ is in E, then there is a directed edge from T_i to T_j, implying that T_i is waiting for T_j to release a data item.
- When T_i requests a data item currently being held by T_j, then the edge $T_i \rightarrow T_j$ is inserted in the wait-for graph. This edge is removed only when T_j is no longer holding a data item needed by T_i.
- The system is in a deadlock state if and only if the wait-for graph has a cycle. Must invoke a deadlock-detection algorithm periodically to look for cycles.

Deadlock Detection (Cont.)



Wait-for graph without a cycle



Wait-for graph with a cycle

Deadlock Recovery

- When deadlock is detected:
- The most common solution is to roll back one or more transactions to break the deadlock.
- Three actions need to be taken:

1. Selection of a victim
2. Rollback
3. Starvation

Deadlock Recovery (Cont.)

1. Selection of victim:

- Some transaction will have to rolled back (made a victim) to break deadlock.
- Select that transaction as victim that will incur minimum cost, factors include:

a) How long the transaction has computed, and how much longer the transaction will compute before it completes its designated task.

b) How many data items the transaction has used.

c) How many more data items the transaction needs for it to complete.

d) How many transactions will be involved in the rollback.

Deadlock Recovery (Cont.)

2. Rollback:

- Rollback -- determine how far to roll back transaction
- **Total rollback:** Abort the transaction and then restart it.
- More effective to roll back transaction only as far as necessary to break deadlock – **partial rollback**.
- Partial rollback requires the system to maintain additional information about the state of all the running transactions.
- Deadlock detection mechanism should decide which locks the selected transaction needs to release in order to break the deadlock.

Deadlock Recovery (Cont.)

3. Starvation:

- Starvation happens if same transaction is always chosen as victim.
- We must ensure that a transaction can be picked as a victim only a (small) finite number of times.
- The most common solution is to include the number of rollbacks in the cost factor.