Lecture 9: Transaction

CS3402 Database Systems

ACID Properties

- Atomicity: A transaction is an atomic unit of processing. It is either performed completely or not performed at all (all or nothing)
- Consistency: A correct execution of a transaction must take the database from one consistent state to another (correctness)
- Isolation: A transaction should not make its updates visible to other transactions until it is committed (no partial results)
- Durability: Once a transaction changes the database state and the changes are committed, these changes must never be lost because of subsequent failure (committed and permanent results)

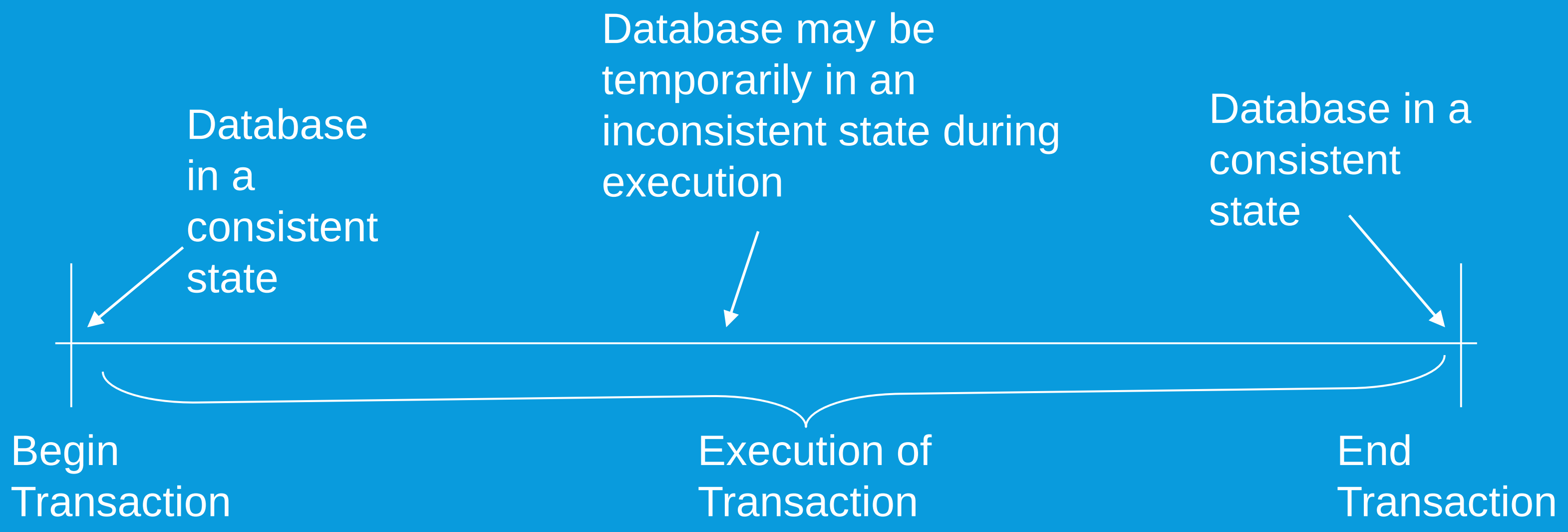
What is a Transaction? (1/2)

- The execution of a program/application on behalf of the user to perform a specific user function by accessing data items maintained in a database management system
  - E.g., use of ATM and buy online tickets
- Transactions – are processes and created by a DBMS and work within its environment
- A logical unit of database processing (a particular functional request from the user) that includes one or more database access operations (read, write, insert, delete)
- Structurally, each transaction is a process and consists of atomic steps. Each atomic step is called an operation
- A transaction = database operations + transaction operations

What is a Transaction? (2/2)

- Database operations: read and write operations on a database
  - Read operation: to read the value of a data item or a group of items, e.g., SELECT
  - Write operation: to create a new value for a data item or a group of items, e.g., UPDATE
  - In between the read/write operations, there may be computation
- Transaction operations: a begin operation and an end operation (commit or abort)
  - For transaction management (where to start and where to finish)
  - The new values from a transaction will become permanent only if the transaction is committed successfully
  - Partial results are not allowed and is considered to be incorrect
  - Atomicity: All or nothing
  - Example: T1: Begin; A:= R(a); B:= R(b) C:= A + B; W(c):= C; End

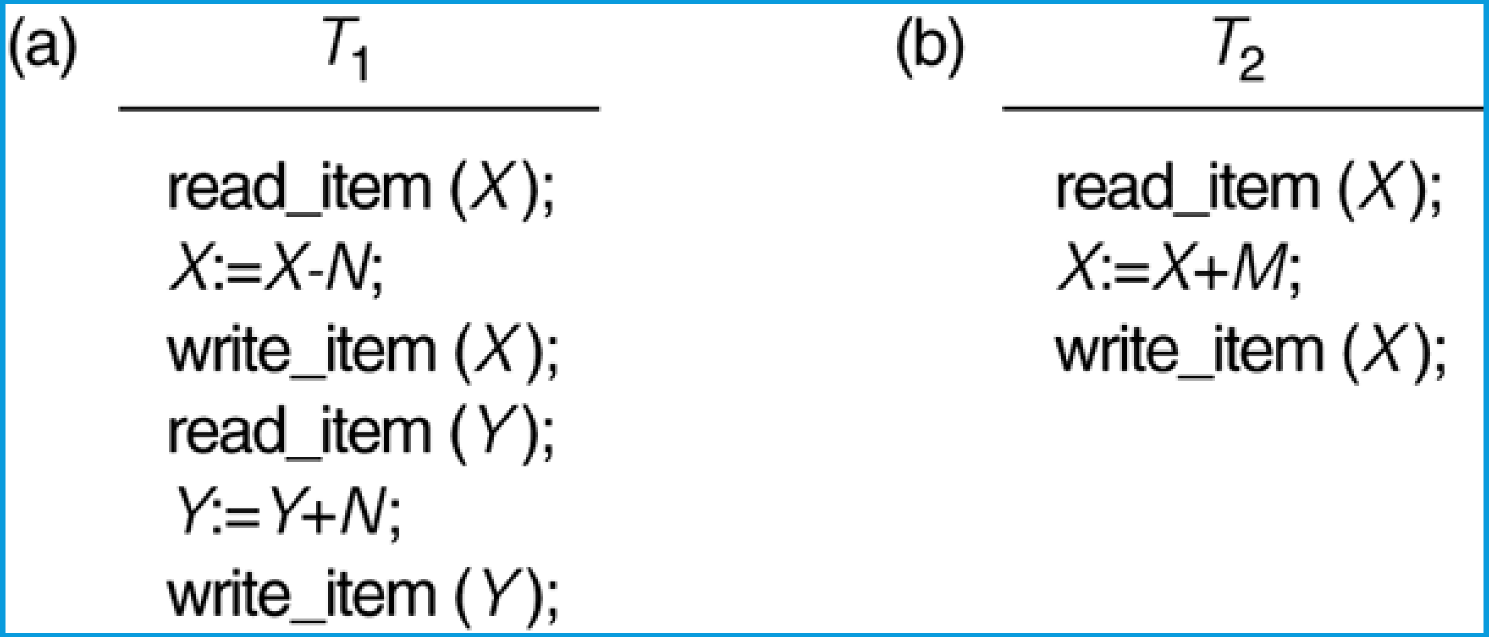
Transaction Structure & Database Consistency


Database in a consistent state
Database may be temporarily in an inconsistent state during execution
Database in a consistent state
Begin Transaction
Execution of Transaction
End Transaction

- The whole transaction is considered as an atomic unit
- Multiple steps (operations) → single user application

Two Sample Transactions

- Two sample transactions (only showing the database operations):



(a) T_1

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

(b) T_2

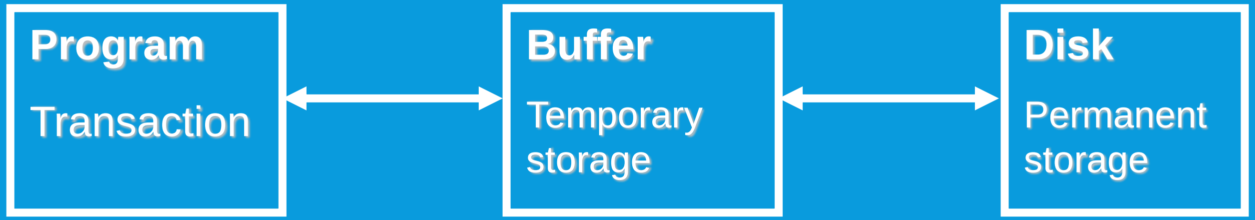
```
read_item (X);
X:=X+M;
write_item (X);
```

Read and Write Operations (1/2)

- Data are resided on disk and the basic unit of data transferring from the disk to the main memory is one disk block
- In general, a data item (what is read or written) will be the field/fields of some records in the database (in a disk block)
- 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
  - Search for the required value in the buffer
  - Copy item X from the buffer to the program variable named X

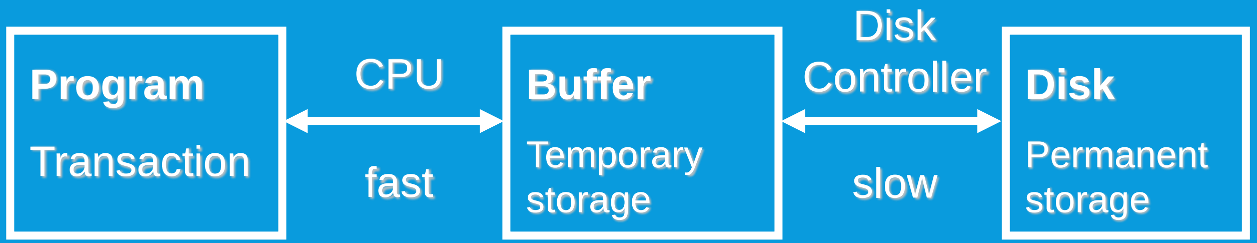
Read and Write Operations (2/2)

- 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)
  - Search for the required value in the 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)
  - Note that we DO NOT need to read an item before update it



Transaction Processing Performance (1/2)

- Fast CPU in accessing buffered data (in main memory)
- Slow disk data access performance (use of buffer to improve the performance)
- Long transactions → many database operations
- Undo and redo of transactions to maintain atomicity in cases of failures



Transaction Processing Performance (2/2)

- Multiuser systems
  - Many users (transactions) can access the database concurrently at the same time Concurrent execution
  - Interleaved processing
    - Concurrent execution of processes/transactions are interleaved in a single CPU system (E.g., use the CPU while waiting for disk block)
  - Parallel processing
    - Processes/transactions are concurrently executed in multiple CPUs system
- Higher Concurrency (more than one transaction is executing)
  - Better performance, i.e., lower response time
  - Problem: difficult to maintain the ACID properties

Interleaved vs Parallel

- While waiting disk data, the CPU processes another transaction

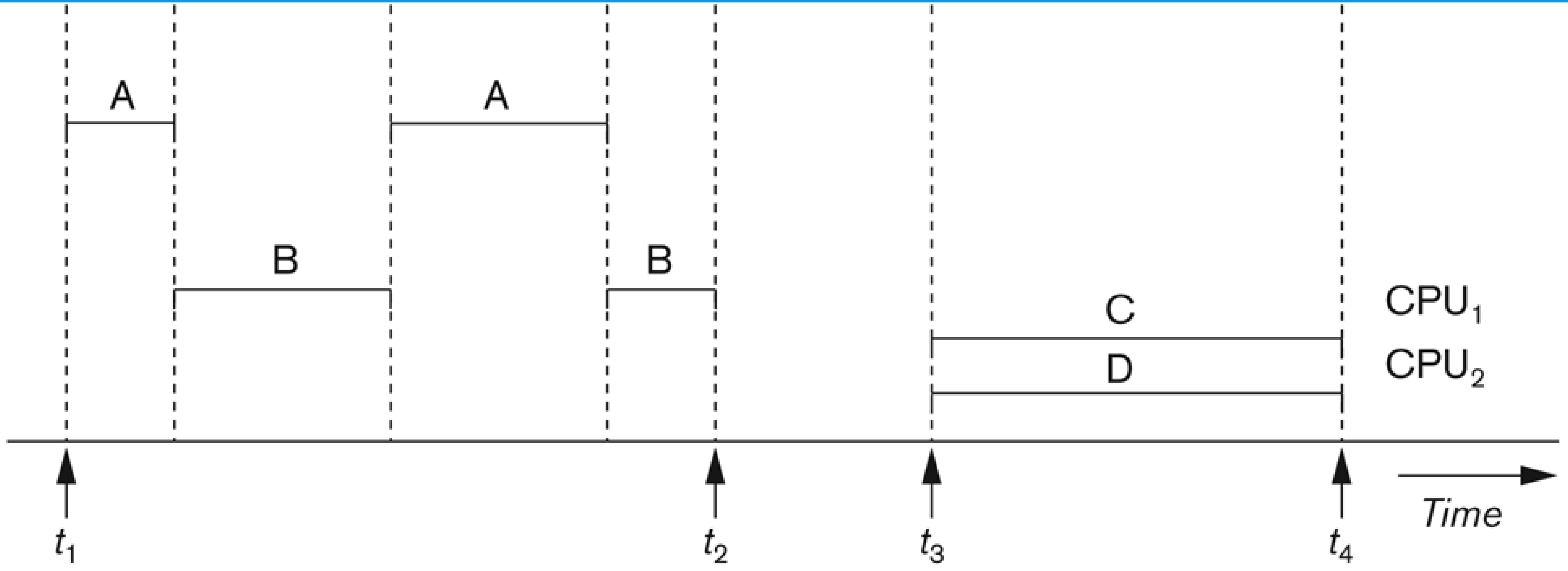


Figure 17.1
Interleaved processing versus parallel processing of concurrent transactions.

Transaction Schedule

- Transaction schedule
  - When transactions are executing concurrently in an interleaved fashion or serially, the order of execution of operations from the transactions forms a transaction schedule
- A schedule S of n transactions T_1, T_2, …, T_n is an ordering of the operations of the transactions subject to the constraint:
  - For each transaction T_i that participates in S, the operations of T_i in S must appear in the same order as in T_i (operations from other transactions T_j can be interleaved with the operations of T_i in S)
- A concurrent schedule: a new transaction starts BEFORE the completion of the current transaction
- A serial schedule: a new transaction only starts AFTER the current transaction is completed

Serial Schedule

- In a serial schedule, the transactions are execution one after one

Transaction *V*:	Transaction *W*:
a.withdraw(100); *b.deposit(100);*	*aBranch.branchTotal();*

a.withdraw(100);	$100		
b.deposit(100);	$300		
		total = a.getBalance();	$100
		total = total+b.getBalance();	$400
		total = total+c.getBalance();	
		...	

A Concurrent Schedule

Transaction *T*:	Transaction *U*:

balance = b.getBalance();
*b.setBalance(balance*1.1);*
a.withdraw(balance/10);
	balance = b.getBalance();
	*b.setBalance(balance*1.1);*
	c.withdraw(balance/10);

balance = b.getBalance();	$200		
		balance = b.getBalance();	$200
		*b.setBalance(balance*1.1);*	$220
*b.setBalance(balance*1.1);*	$220		
a.withdraw(balance/10);	$80		
		c.withdraw(balance/10);	$280

Time

Schedule Examples

- T_1: $w_1(x)$, $r_1(y)$, $w_1(y)$
- T_2: $r_2(x)$, $r_2(y)$, $w_2(y)$
- A possible schedule for T_1 and T_2:
  - $w_1(x)$, $r_2(x)$, $r_1(y)$, $w_1(y)$, $r_2(y)$, $w_2(y)$
- Is the following one a serial schedule?
  - $r_1(y)$, $w_1(y)$, $r_2(x)$, $r_2(y)$, $w_2(y)$, $w_1(x)$ (concurrent schedule)
- How about this one?
  - $r_1(y)$, $w_1(y)$, $w_1(x)$, $r_2(x)$, $r_2(y)$, $w_2(y)$ (serial schedule)

Example Consistency Problems

- Lost update problem (write/write conflicts)
  - 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 (inconsistent)
- Inconsistent retrieval problem (read/write conflicts)
  - If a 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

Lost Update Problem

- A schedule shows the execution order of the operations of two concurrently executing transactions (Initial account value: A=100; B=200; C=300)

Transaction *T*:	Transaction *U*:
balance = Read(b)	*balance = Read(b)*
Write(b) = balance + 20	*Write(b) = balance + 30*
Write(a) = balance - 20	*Write(c) = balance - 30*

balance = Read(b)	$200		
		balance = Read(b)	$200
		Write(b)= balance + 30	$230
Write(b) = balance + 20	$220		
Write(a) = balance - 20	$80		
		Write(c) = balance - 30	$270

Time ↓

Schedules Classified on Serializability

- Serial schedule
  - A schedule S is serial if, for every transaction T participating in the schedule, all the operations of T are executed consecutively in the schedule
  - Serial schedules can maintain the database consistency
    - But, poor performance
- Serializable schedule
  - A concurrent schedule S which is equivalent to a serial schedule
  - Can guarantee the database consistency and can have better performance

Serial Equivalence

- A serially equivalent schedule means that the results from the schedule is equivalent to a serial schedule, i.e., execute one after one

Transaction *T*:	Transaction *U*:
balance = Read(b)	*balance = Read(b)*
Write(b) = balance + 20	*Write(b) = balance + 30*
Write(a) = balance - 20	*Write(c) = balance - 30*
balance = Read(b) $200	
Write(b) = balance + 20 $220	
	balance = Read(b) $220
	Write(b) = balance + 30 $250
Write(a) = balance - 20 $80	
	Write(c) = balance - 30 $270

Read and Write Operation Conflict Rules

Operations of Different Transactions		Conflict	Reason
Read	Read	No	Because the effect of a pair of read operations does not depend on the order in which they are executed
Read	Write	Yes	Because the effect of a read and a write operation depends on the order of their execution
Write	Write	Yes	Because the effect of a pair of write operations depends on the order of their execution

Schedules Classified on Serializability

- Conflict equivalent schedule
  - Two schedules are said to be conflict equivalent if the order of any two conflicting operations (RW,WW) is the same in both schedule. Example:

S1

T1	T2
Read(A)	
Write(A)	
	Read(A)
Read(B)	

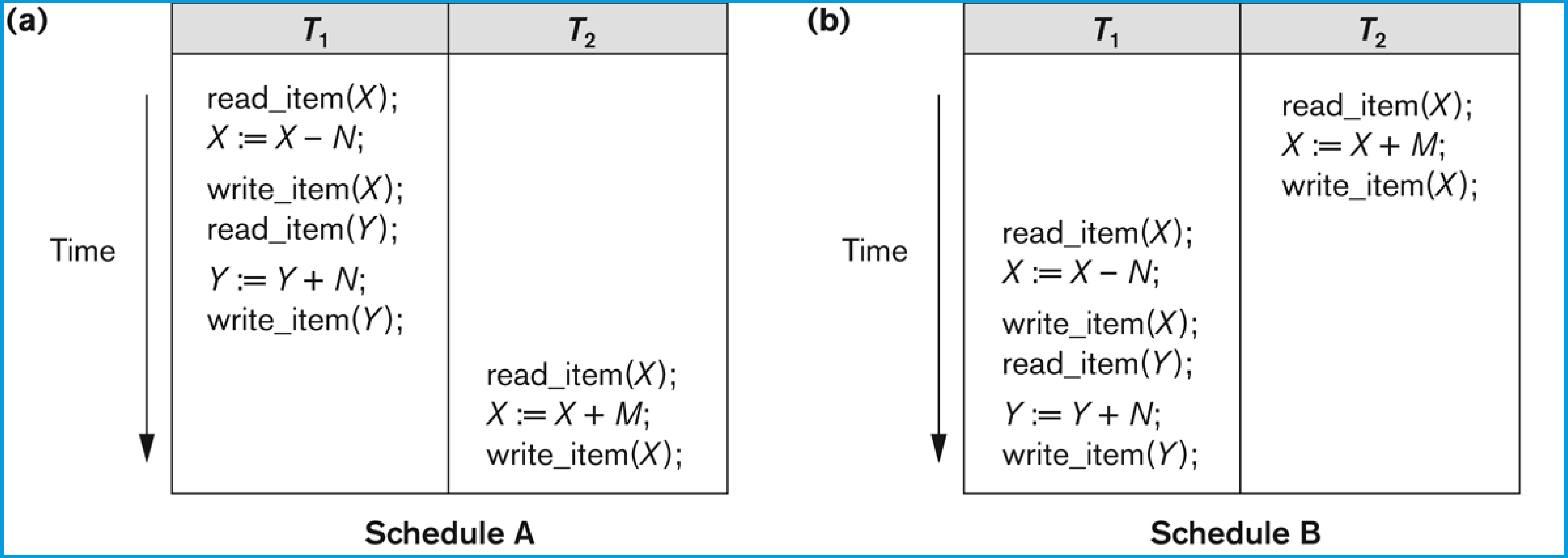
S2

T1	T2
Read(A)	
Write(A)	
Read(B)	
	Read(A)

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

Examples of Serial Schedules

- Examples of serial 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



(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

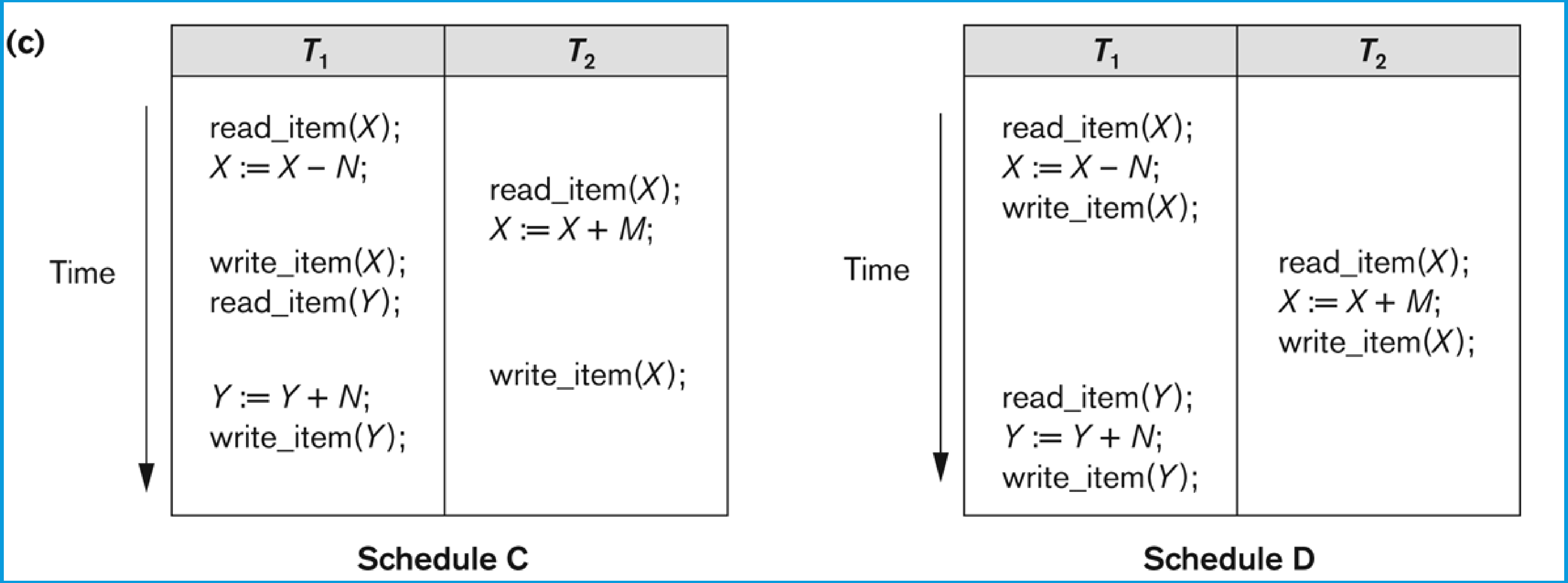
(b)

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

Examples of Nonserial Schedules

- Examples of nonserial schedules involving transactions T_1 and T_2
  - Two nonserial schedules C and D with interleaving of operations



(c)

Schedule 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);	

Schedule 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);	

Serialization Graphs

- The determination of a conflict serializable schedule can be done by the use of serialization graph (SG) or called precedence graph
- A serialization graph tells the effective execution order of a set of transactions
- The set of edges consists of all edges $T_i \rightarrow T_j$ for which one of the following three conditions holds:
  - W/R conflict: T_i executes write(x) before T_j executes read(x)
  - R/W conflict: T_i executes read(x) before T_j executes write(x)
  - W/W conflict: T_i executes write(x) before T_j executes write(x)
- Each edge $T_i \rightarrow T_j$ in a SG means that at least one of T_i's operations precede and conflict with one of T_j's operations
- Serializability theorem: A schedule is serializable if and only if the SG is acyclic

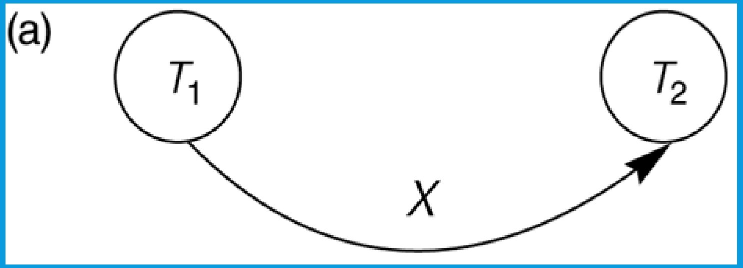
Serialization/Precedence Graph

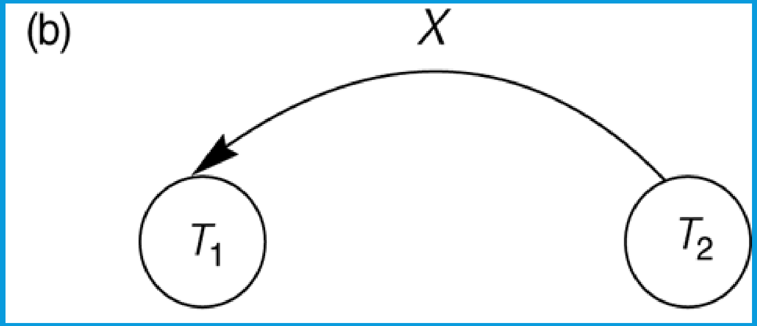
Transaction *T*:	**Transaction *U*:**
x = read(i)	
write(i, 10)	
	y = read(j)
	write(j, 30)
write(j, 20)	
	z = read(i)

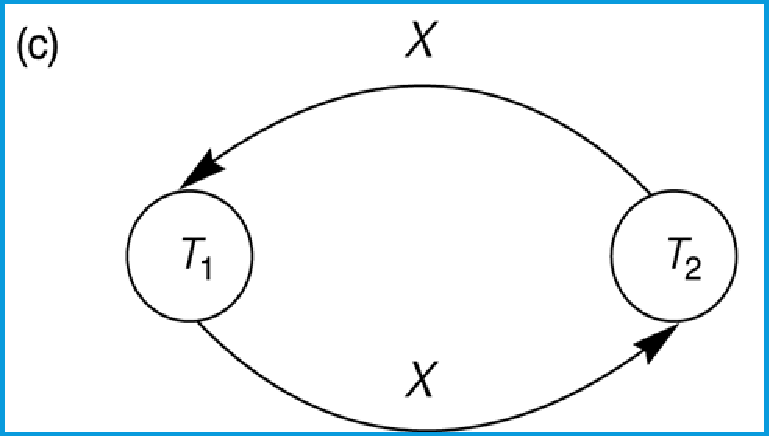
$$\begin{tikzcd} T \arrow[r, bend left] & U \arrow[l, bend left] \end{tikzcd}$$

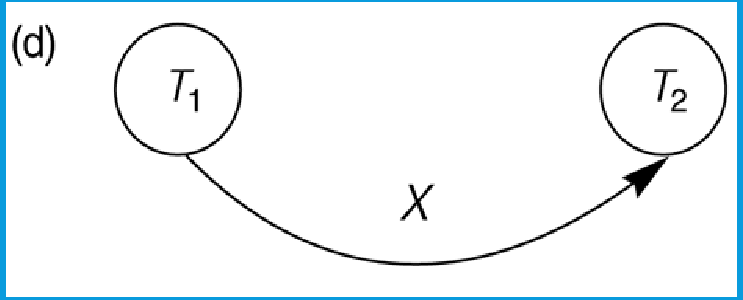
Serialization/Precedence Graph: Examples

- Constructing the precedence graphs for schedules A and D on Slides #22 and #23 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 conflict serializable).
  - (d) Precedence graph for schedule D (conflict serializable, equivalent to schedule A).









Schedules Classified on Recoverability

- Recoverable schedule:
  - If a transaction is aborted, all its effects have to be undone
  - Rollback of a transaction: undo those processed operations of an aborted transaction
  - Why needs to be rollback $\rightarrow$ maintaining consistency and all or nothing property
- A schedule S is recoverable if
  - For all T_i and T_j where T_i read an item written by T_j, T_j commits before T_i
- Cascaded rollback
  - A single rollback leads to a series of rollback
  - All uncommitted transactions that read data items from a failed (aborted) transaction must be rolled back

How to Ensure Recoverability?

- To ensure recoverability (undo to previous database state):
- No dirty read: reading uncommitted data items
- No premature write: no update on a data item if another transaction has updated it and the transaction has not committed
- Strict execution: delay the reading and updating of a data item until the previous transaction that has updated the same data item has committed/aborted

Non-recoverable vs. Recoverable

T_1	T_2
read(A)	
write(A)	
	read(A)
	commit/abort
read(b)	
commit/abort	

- Non-recoverable
  - If T_2 commits and then T_1 aborts

T1	T2
read(A)	
write(A)	
	read(A)
read(b)	
commit/abort	
	commit/abort

- Recoverable

Cascade Rollback

T_1	T_2	T_3
read(A)		
read(B)		
write(A)		
	read(A)	
	write(A)	
		read(A)
commit/abort		
	commit/abort	
		commit/abort

- Recoverable but when T_1 fails, T_2 and T_3 should rollback

Dirty Read

Transaction *T*:		Transaction *U*:	
balance = read (a)	$100		
write(a) = balance + 10	$110		
		balance = read(a)	$110
		write(a) = balance + 20	$130
		commit	
abort			

Dirty read

A Transaction may Fail (1/2)

- Why recovery is needed? (What causes a Transaction to abort)
  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 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.

A Transaction may Fail (2/2)

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
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
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, overwriting disks

Transaction State

- A transaction is an atomic unit of work that is either completed in its entirety or not done at all (atomicity)
  - For recovery purposes, the system needs to keep track of when a transaction starts, terminates, and commits or aborts
- Transaction states
  - Active state
  - Partially committed state
  - Committed state
  - Failed state
  - Terminated State

State Transition Diagram

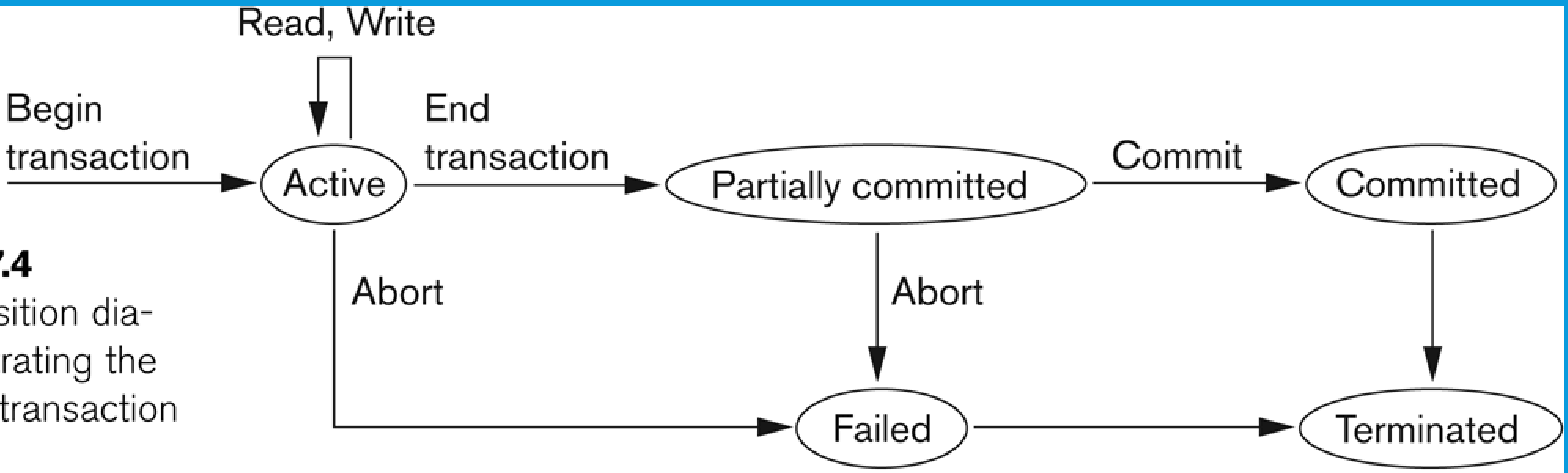
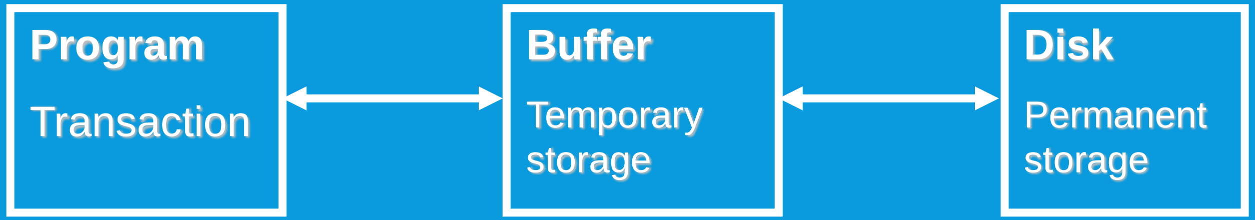


Figure 17.4
State transition diagram illustrating the states for transaction execution.

Primitive Operations of Transactions

- INPUT(X): copy the disk block containing database element X to a memory buffer
- READ(X, t): copy the database element X to the transaction's local variable t. If the block containing database element X is not in a memory buffer, first INPUT(X). Next, assign the value of X to a local variable t
- WRITE(X, t): copy the value of local variable t to database element X in a memory buffer. If the block containing database element is not in a memory buffer, execute INPUT(X). Next copy the value of t to X in the buffer
- OUTPUT(X): copy the block containing X from its buffer to disk



Primitive Operations of Transactions

- Transaction
  - Read(A, t); t:= t * 2; Write(A, t); Read(B, t); t:= t * 2; Write(B, t)

Action	t	MEM A	MEM B	DISK A	DISK B
Read(A, t)	8	8		8	8
t:= t * 2;	16	8		8	8
Write(A, t);	16	16		8	8
Read(B, t);	8	16	8	8	8
t:= t * 2;	16	16	8	8	8
Write(B, t)	16	16	16	8	8
Output(A)	16	16	16	16	8
Output(B)	16	16	16	16	16

Undo Logging for Recovery

- A log is a file of log records, each telling something about what some transaction has done
- As transactions execute, the log manager records in the log each important event (e.g., write operations)
- Logs are initially created in main memory and are allocated by the buffer manager
    - Why not on disk? Disk I/O takes a lot of time
- Logs are copied to disk by “flush-log” operation
- If log records appear in nonvolatile storage (disk), we can use them to restore the database to a consistent state after a system crash
- After a system failure, all data in volatile storage (memory) will lose but the data in nonvolatile storage remain

Log Records

- <START T>: This record indicates that transaction T has begun
- <COMMIT T>:
    - Transaction T has completed successfully and will make no more changes to database.
    - Because we cannot control when the buffer manager chooses to copy blocks from memory to disk, we cannot be sure that the changes are already on disk when we see <COMMIT T>
- <ABORT T>: Transaction T could not complete successfully. If transaction T aborts, no change it made can be copied to disk, and it is the job of the transaction manager to make sure that such changes never appear on disk

Undo-Logging Rules

- U1: if transaction T modifies database element X (current value is v), the log record of the form <T, X, v> must be written to the disk BEFORE the new value of X is written to disk
- U2: if a transaction commits, its COMMIT log record must be written to disk ONLY after all database elements changed by the transaction have been written to disk
- A transaction must be written to disk in the following order
  1. The log records indicating changed database elements
  2. The changed database elements themselves
  3. The COMMIT log record

Undo Rules and Actions

Action	t	MEM A	MEM B	DISK A	DISK B	Log
						<START T>
Read(A, t)	8	8		8	8	
t:= t * 2;	16	8		8	8	
Write(A, t);	16	16		8	8	<T, A, 8>
Read(B, t);	8	16	8	8	8	
t:= t * 2;	16	16	8	8	8	
Write(B, t)	16	16	16	8	8	<T, B, 8>
FLUSH LOG						(1)
Output(A)	16	16	16	16	8	(2)
Output(B)	16	16	16	16	16	(2)
						<COMMIT T> (3)
FLUSH LOG						

Recovery Using Undo-Logging (1/2)

- Suppose a system failure occurs while a transaction is committing. It is possible certain changes made by the transaction were written to disk while others changes never reached the disk
- Recovery manager has to REMOVE the partial changes of a transaction
- If there is a log record <COMMIT T>, by undo rule U2, all changes made by T were previously written to disk
- If we find <START T> on the log but no <COMMIT T> record, T may be incomplete and must be undone
- Rule U1 assures that if T changed X on disk before the crash, there will be a <T, X, v> record on the log, and that record will have been copied to disk before the crash. During recovery, we must write the value v for X

Recovery Using Undo-Logging (2/2)

- The recovery manager scans the log from the end and remembers all those transactions T for which it has a <COMMIT T> record or an <ABORT T> record
- If it sees a record <T, X, v>
  - If T is a transaction whose COMMIT record has been seen, do nothing. T is committed and must not be undone
  - Otherwise, T is an incomplete transaction, or an aborted transaction. The recovery manager must change the value of X in the database to v in case X had been altered just before the crash
  - After making the changes, the recovery manager must write a log record <ABORT T> for each incomplete transaction that was not previously aborted and the flush the log

Redo Logging for Recovery

- Undo logging has a potential problem that we cannot commit a transaction without first writing all changed data to disk (flush $\rightarrow$ commit)
- Differences between redo and undo logging
  - While undo logging cancels the effect of incomplete transactions and ignores committed ones during recovery, redo logging ignores incomplete transactions and repeats the changes made by committed transactions
  - While undo logging requires us to write changed database elements to disk before the COMMIT log records reaches disk, redo logging requires that the COMMIT record appears on disk before any changed values reach disk

Redo Logging Rules

- R1: before modifying any database element X on disk, it is necessary that all log records pertaining to this modification of X, including both the update record <T, X, v> (v is the new value) and <COMMIT T> record, must appear on disk
- The order in which material associated with one transaction gets to written to disk is:
  1. The log record indicating changed database elements
  2. The COMMIT log record
  3. The changed database elements themselves
- Undo: Log → change → COMMIT
- Redo: Log → COMMIT → change

Redo Rules and Actions

Action	t	MEM A	MEM B	DISK A	DISK B	Log
						<START T>
Read(A, t)	8	8		8	8	
t:= t * 2;	16	8		8	8	
Write(A, t);	16	16		8	8	<T, A, 16>
Read(B, t);	8	16	8	8	8	
t:= t * 2;	16	16	8	8	8	
Write(B, t)	16	16	16	8	8	<T, B, 16>
						<COMMIT T>
FLUSH LOG						
Output(A)	16	16	16	16	8	
Output(B)	16	16	16	16	16	

Recovery Using Redo-Logging

- To recover using a redo log, after a system crash:
  - Identify the committed transactions
  - Scan the log forward from the beginning. For each log record <T, X, v> encountered:
    - If T is not a committed transaction, do nothing
    - If T is a committed transaction, write value v for database element X
  - For each incomplete transaction T, write an <ABORT T> record to the log and flush the log