Introduction to Transaction Processing Concepts and Theory

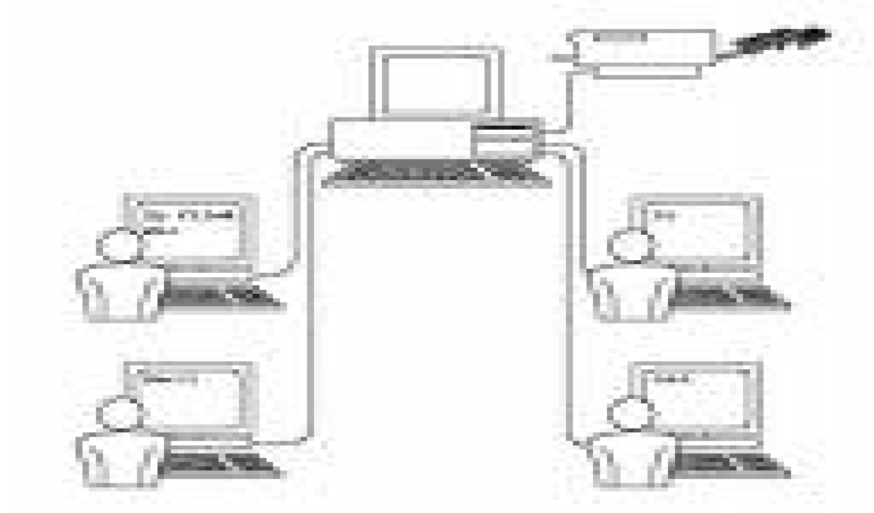
Chapter 17

Chapter Outline

- Introduction to Transaction Processing
- Transaction and System Concepts
- Desirable Properties of Transactions
- Concurrent executions

- Introduction to Transaction Processing ...

- **System Model:**
  - **Multiuser System**: Many users can access the system concurrently.
  - **Concurrency**
    - **Interleaved processing**: concurrent execution of processes is interleaved in a single CPU



... - Introduction to Transaction Processing ...

- **A Transaction:** logical unit of database processing that includes one or more access operations (read -retrieval, write - insert or update, delete).

- **A transaction (set of operations)** may be stand-alone specified in a high level language like SQL submitted interactively, or may be embedded within a program.

- **Transaction boundaries**: Begin and End transaction.

- An **application program** may contain several transactions separated by the Begin and End transaction boundaries.

... - Introduction to Transaction Processing ...

SIMPLE MODEL OF A DATABASE (for purposes of discussing transactions):

- **A database -** collection of named data items
- **Granularity of data** - a field, a record , or a whole disk block (Concepts are independent of granularity)
- Basic operations 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.

-- Read Operation

- Basic unit of data transfer from the disk to the computer main memory is one block. In general, a data item (what is read or written) will be the field of some record in the database, although it may be a larger unit such as a record or even a whole block.

- **read_item(X) command includes the following steps:**

1. Find the address of the disk block that contains item X.

2. Copy that disk block into a buffer in main memory (if that disk block is not already in some main memory buffer).

3. Copy item X from the buffer to the program variable named X.

-- Write Operation

- **write_item(X) command includes the following steps:**

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

-- A Sample transaction

- Transaction to transfer $50 from account *A* to account *B*:

```
read(A)
A := A - 50
write(A)
read(B)
B := B + 50
write(B)
```

- Why recovery is needed ...

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.

... - Why recovery is needed ...

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 (see Chapter 18).

… - Why recovery is needed …

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.

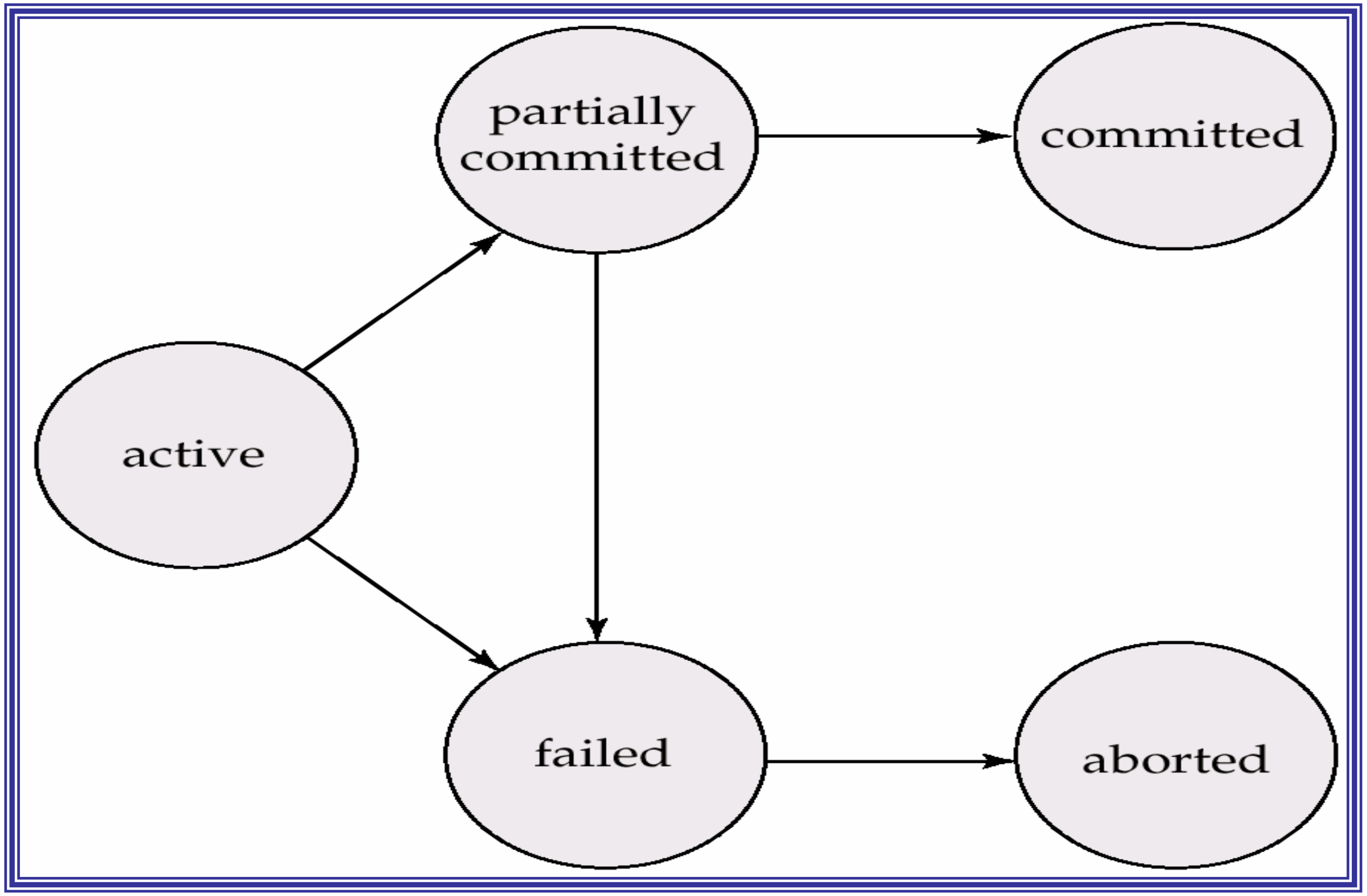
6. **Physical problems and catastrophes:** This refers to an endless list of problems that includes power or air-conditioning failure, fire, theft, sabotage, overwriting disks or tapes by mistake, and mounting of a wrong tape by the operator.

- Transaction 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

... - Transaction and System Concepts ...



... - Transaction and System Concepts ...

- 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.
  - **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.

... - Transaction and System Concepts ...

- ... Recovery manager keeps track of the following operations
  - **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 and System 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.

... - Transaction and System Concepts ...

- **The System Log ...**
  - **Log or Journal** : The log keeps track of all transaction operations that affect the values of database items. This information may be needed to permit recovery from transaction failures. The log is kept on disk, so it is not affected by any type of failure except for disk or catastrophic failure. In addition, the log is periodically backed up to archival storage (tape) to guard against such catastrophic failures.
  - T in the following discussion refers to a unique **transaction-id** that is generated automatically by the system and is used to identify each transaction:

... - Transaction and System Concepts ...

- **... The System Log** - **Types of log record:**

1. **[start_transaction,T]:** Records that transaction T has started execution.
2. **[write_item,T,X,old_value,new_value]:** Records that transaction T has changed the value of database item X from old_value to new_value.
3. **[read_item,T,X]:** Records that transaction T has read the value of database item X.
4. **[commit,T]:** Records that transaction T has completed successfully, and affirms that its effect can be committed (recorded permanently) to the database.
5. **[abort,T]:** Records that transaction T has been aborted.

... - Transaction and System Concepts ...

- **Recovery using log records:**
  - If the system crashes, we can recover to a consistent database state by examining the log and using one of the techniques described in Chapter 19.
    1. Because the log contains a record of every write operation that changes the value of some database item, it is possible to **undo** the effect of these write operations of a transaction T by tracing backward through the log and resetting all items changed by a write operation of T to their old_values.
    2. We can also **redo** the effect of the write operations of a transaction T by tracing forward through the log and setting all items changed by a write operation of T (that did not get done permanently) to their new_values.

... - Transaction and System Concepts ...

- **Commit Point of a Transaction: ...**
  - **Definition:** A transaction T reaches its **commit point** when all its operations that access the database have been executed successfully *and* the effect of all the transaction operations on the database has been recorded in the log. Beyond the commit point, the transaction is said to be **committed,** and its effect is assumed to be *permanently recorded* in the database. The transaction then writes an entry [commit,T] into the log.
  - **Roll Back of transactions:** Needed for transactions that have a [start_transaction,T] entry into the log but no commit entry [commit,T] into the log.

... - Transaction and System Concepts ...

- **... Commit Point of a Transaction:**
  - **Redoing transactions:** Transactions that have written their commit entry in the log must also have recorded all their write operations in the log; otherwise they would not be committed, so their effect on the database can be *redone* from the log entries. (Notice that the log file must be kept on disk. At the time of a system crash, only the log entries that have been *written back to disk* are considered in the recovery process because the contents of main memory may be lost.)
  - **Force writing a log:** *before* a transaction reaches its commit point, any portion of the log that has not been written to the disk yet must now be written to the disk. This process is called force-writing the log file before committing a transaction.

- Desirable Properties of Transactions

ACID properties:

- **Atomicity**: A transaction is an atomic unit of processing; it is either performed in its entirety or not performed at all.
- **Consistency preservation**: A correct execution of the transaction must take the database from one consistent state to another.
- **Isolation**: A transaction should not make its updates visible to other transactions until it is committed; this property, when enforced strictly, solves the temporary update problem and makes cascading rollbacks of transactions unnecessary (see Chapter 21).
- **Durability or permanency**: Once a transaction changes the database and the changes are committed, these changes must never be lost because of subsequent failure.

-- 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.

- 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

- 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.

- 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).
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.

- Concurrent Executions

- Multiple transactions are allowed to run concurrently in the system. Advantages are:
  - **increased processor and disk utilization**, leading to better transaction *throughput:* one transaction can be using the CPU while another is reading from or writing to the disk
  - **reduced average response time** for transactions: short transactions need not wait behind long ones.
- *Concurrency control schemes* – mechanisms to achieve isolation, i.e., to control the interaction among the concurrent transactions in order to prevent them from destroying the consistency of the database

-- 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.

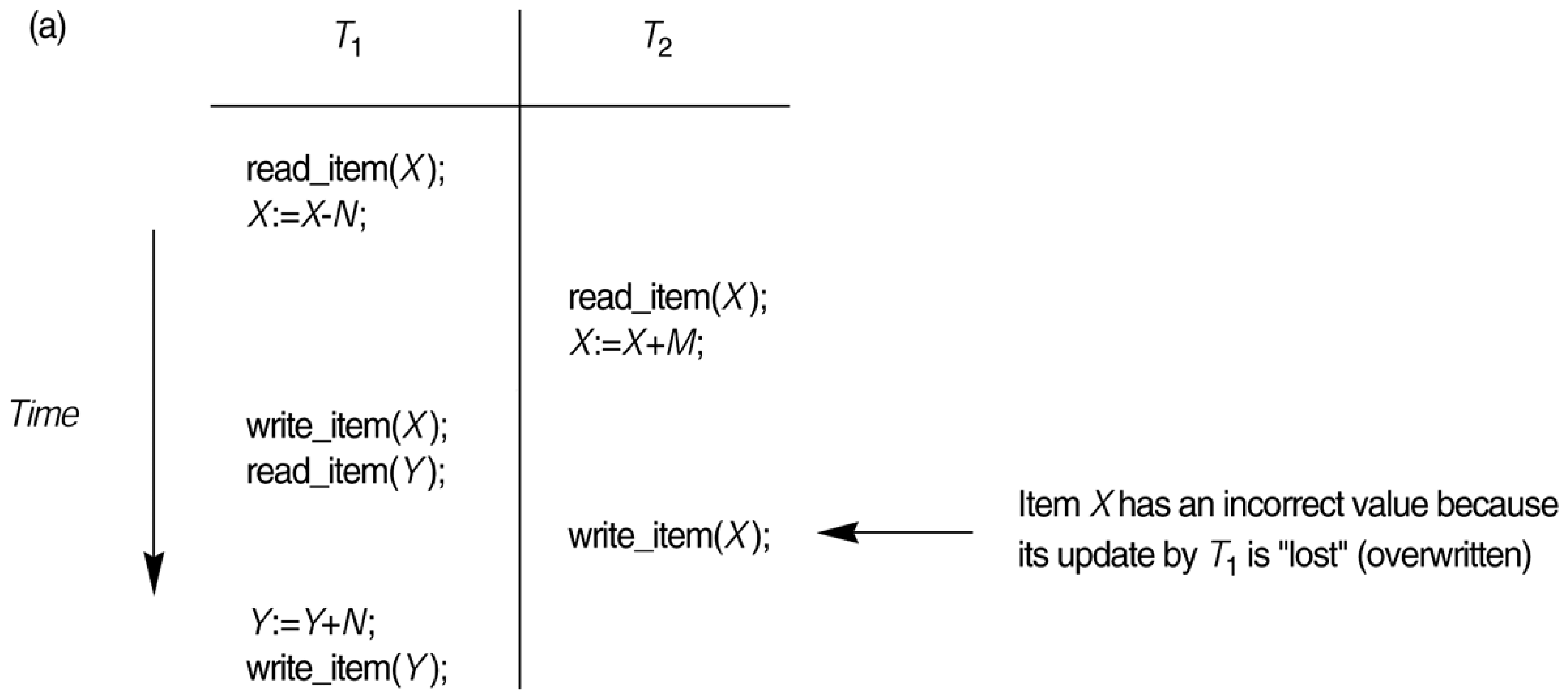
- **The Temporary Update (or Dirty Read) Problem.**

  This occurs when one transaction updates a database item and then the transaction fails for some reason (see Section 17.1.4). The updated item is accessed by another transaction before it is changed back to its original value.

- **The Incorrect Summary Problem .**

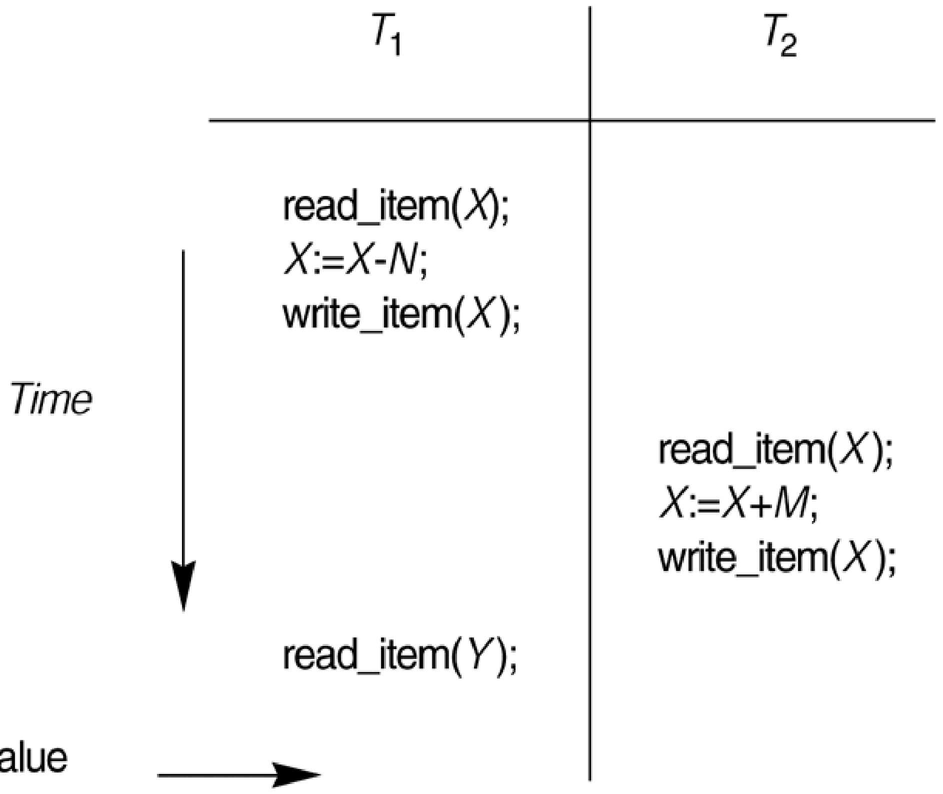
  If one transaction is calculating an aggregate summary function on a number of records while other transactions are updating some of these records, the aggregate function may calculate some values before they are updated and others after they are updated

-- The lost update problem



-- The temporary update problem

(b)



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.

-- 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*).

-- Schedules

- *Schedules* – sequences that indicate the chronological order in which instructions of concurrent transactions are executed
  - a schedule for a set of transactions must consist of all instructions of those transactions
  - must preserve the order in which the instructions appear in each individual transaction.

--- Example Schedules ...

- Let T_1 transfer $50 from A to B, and T_2 transfer 10% of the balance from A to B. The following is a serial schedule, in which T_1 is followed by T_2.

Schedule 1

T_1	T_2
read(A)	
$A := A - 50$	
write (A)	
read(B)	
$B := B + 50$	
write(B)	
	read(A)
	$temp := A * 0.1$
	$A := A - temp$
	write(A)
	read(B)
	$B := B + temp$
	write(B)

… --- Example Schedule …

- Let T_1 and T_2 be the transactions defined previously. The following schedule is not a serial schedule, but it is *equivalent* to Schedule 1.

Schedule 2

T_1	T_2
read(A)	
$A := A - 50$	
write(A)	
	read(A)
	$temp := A * 0.1$
	$A := A - temp$
	write(A)
read(B)	
$B := B + 50$	
write(B)	
	read(B)
	$B := B + temp$
	write(B)

In both Schedule 1 and 2, the sum A + B is preserved.

… --- Example Schedules

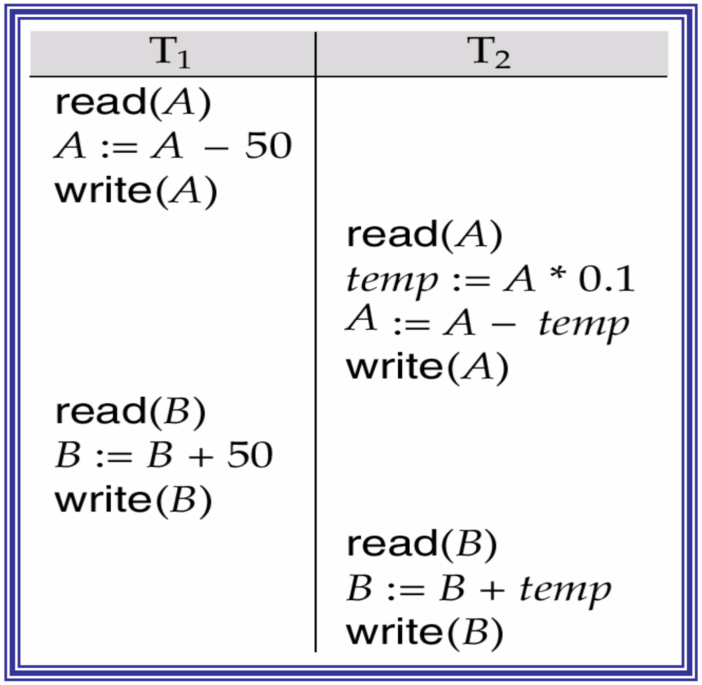
- The following concurrent schedule does not preserve the value of the the sum *A* + *B*.

T_1	T_2
read(A)	
$A := A - 50$	
	read(A)
	$temp := A * 0.1$
	$A := A - temp$
	write(A)
	read(B)
write(A)	
read(B)	
$B := B + 50$	
write(B)	
	$B := B + temp$
	write(B)

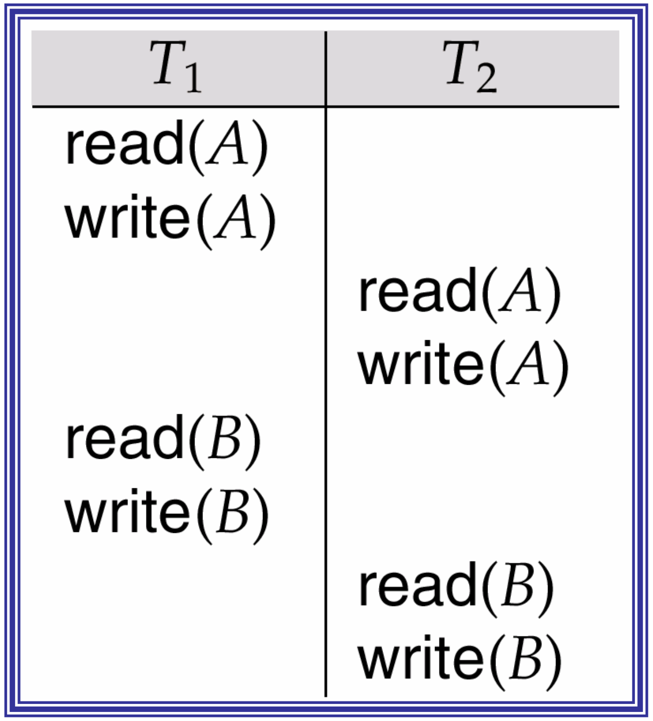
-- Serializability ...

- Basic Assumption – Each transaction preserves database consistency.
- Thus serial execution of a set of transactions preserves database consistency.
- A (possibly concurrent) schedule is serializable if it is equivalent to a serial schedule. Different forms of schedule equivalence give rise to the notions of:
  1. conflict serializability
  2. view serializability
- We ignore operations other than **read** and **write** instructions, and we assume that transactions may perform arbitrary computations on data in local buffers in between reads and writes. Our simplified schedules consist of only **read** and **write** instructions.

... -- Serializability



T_1	T_2
read(A)	
$A := A - 50$	
write(A)	
	read(A)
	$temp := A * 0.1$
	$A := A - temp$
	write(A)
read(B)	
$B := B + 50$	
write(B)	
	read(B)
	$B := B + temp$
	write(B)



T_1	T_2
read(A)	
write(A)	
	read(A)
	write(A)
read(B)	
write(B)	
	read(B)
	write(B)

--- Conflict Serializability ...

- Instructions I_i and I_j of transactions T_i and T_j respectively, **conflict** if and only if there exists some item Q accessed by both I_i and I_j, and at least one of these instructions wrote Q.

  1. I_i = **read**(Q), I_j = **read**(Q). I_i and I_j don't conflict.
  2. I_i = **read**(Q), I_j = **write**(Q). They conflict.
  3. I_i = **write**(Q), I_j = **read**(Q). They conflict
  4. I_i = **write**(Q), I_j = **write**(Q). They conflict

- Intuitively, a conflict between I_i and I_j forces a (logical) temporal order between them. If I_i and I_j are consecutive in a schedule and they do not conflict, their results would remain the same even if they had been interchanged in the schedule.

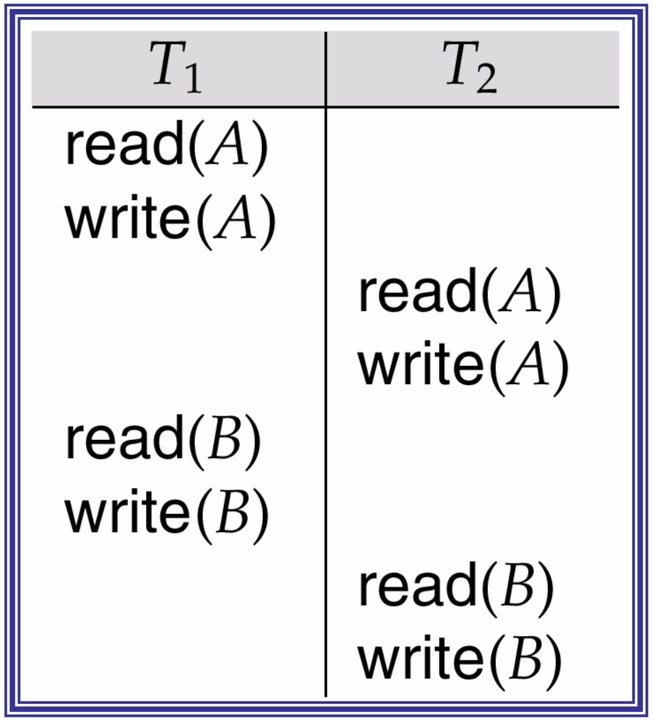
… --- Conflict Serializability …

- If a schedule *S* can be transformed into a schedule *S´* by a series of swaps of non-conflicting instructions, we say that *S* and *S´* are **conflict equivalent**.

- We say that a schedule *S* is **conflict serializable** if it is conflict equivalent to a serial schedule

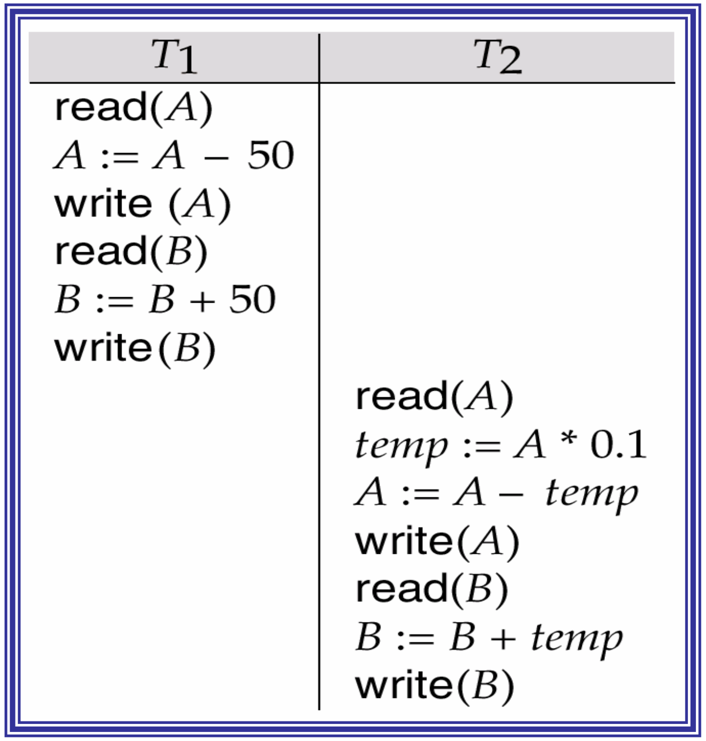
... --- Conflict Serializability ...

- Schedule 3 below can be transformed into Schedule 1, a serial schedule where T_2 follows T_1, by series of swaps of non-conflicting instructions. Therefore Schedule 3 is conflict serializable.



T_1	T_2
read(A)	
write(A)	
	read(A)
	write(A)
read(B)	
write(B)	
	read(B)
	write(B)

3



T_1	T_2
read(A)	
$A := A - 50$	
write (A)	
read(B)	
$B := B + 50$	
write(B)	
	read(A)
	$temp := A * 0.1$
	$A := A - temp$
	write(A)
	read(B)
	$B := B + temp$
	write(B)

1

... --- Conflict Serializability

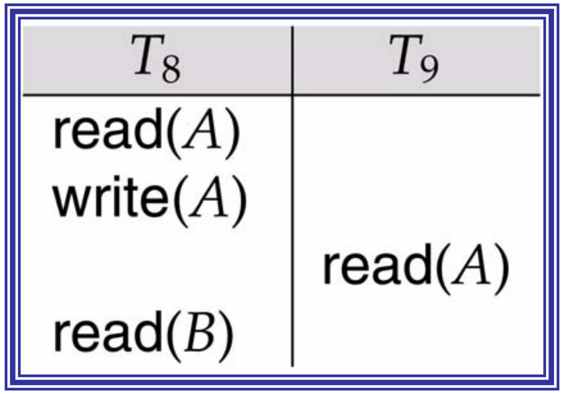
- Example of a schedule that is not conflict serializable:

T_3	T_4
read(Q)	
	write(Q)
write(Q)	

We are unable to swap instructions in the above schedule to obtain either the serial schedule < T_3, T_4 >, or the serial schedule < T_4, T_3 >.

-- Recoverability ...

- **Recoverable schedule** — if a transaction T_j reads a data items previously written by a transaction T_i, the commit operation of T_i appears before the commit operation of T_j.
- The following schedule is not recoverable if T_9 commits immediately after the read

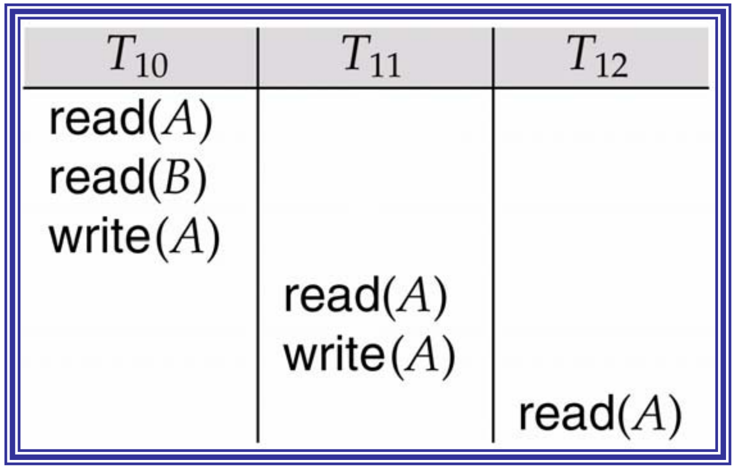


T_8	T_9
read(A)	
write(A)	
	read(A)
read(B)	

- If T_8 should abort, T_9 would have read (and possibly shown to the user) an inconsistent database state. Hence database must ensure that schedules are recoverable.

... -- Recoverability ...

- **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)



T_{10}	T_{11}	T_{12}
read(A)		
read(B)		
write(A)		
	read(A)	
	write(A)	
		read(A)

If T_{10} fails, T_{11} and T_{12} must also be rolled back.

- Can lead to the undoing of a significant amount of work

… -- Recoverability

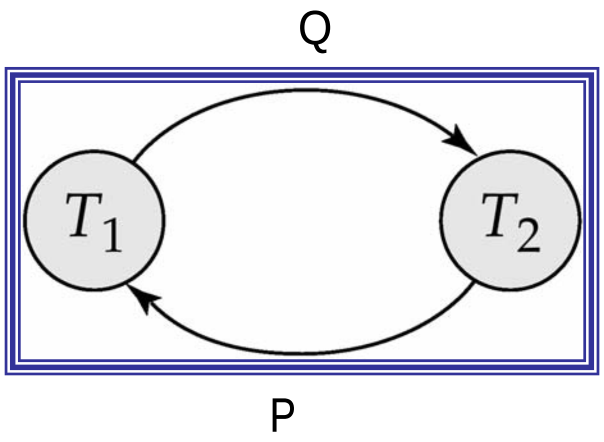
- **Cascadeless schedules** — cascading rollbacks cannot occur; for each pair of transactions T_i and T_j such that T_j reads a data item previously written by T_i, the commit operation of T_i appears before the read operation of T_j.
- Every cascadeless schedule is also recoverable
- Schedules must be conflict serializable and recoverable, for the sake of database consistency, and preferably cascadeless.
- Concurrency-control schemes tradeoff between the amount of concurrency they allow and the amount of overhead that they incur.
  - For example: A policy in which only one transaction can execute at a time generates serial schedules of less overhead, but provides a poor degree of concurrency.

-- Testing for Serializability ...

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

… -- Testing for Serializability

T_1	T_2
read(Q)	
	write(Q)
	Read(P**)**
Write(P)	



-- Example Schedule (Schedule A)

T_1	T_2	T_3	T_4	T_5
	read(X)			
read(Y)				
read(Z)				
				read(V)
				read(W)
				read(W)
	read(Y)			
	write(Y)			
		write(Z)		
read(U)				
			read(Y)	
			write(Y)	
			read(Z)	
			write(Z)	
read(U)				
write(U)				

--- Precedence Graph for Schedule A

$$\begin{tikzcd}
T_1 \arrow[r, bend left] \arrow[d, bend right] & T_2 \arrow[d, bend left] \\
T_3 \arrow[r, bend right] & T_4
\end{tikzcd}$$

-- Test for Conflict Serializability

- A schedule is conflict serializable if and only if its precedence graph is acyclic.

- Cycle-detection algorithms exist which take order n^2 time, where n is the number of vertices in the graph. (Better algorithms take order $n + e$ where e is the number of edges.)

- If precedence graph is acyclic, the serializability order can be obtained by a *topological sorting* of the graph. This is a linear order consistent with the partial order of the graph.
  For example, a serializability order for Schedule A would be
  $T_5 \rightarrow T_1 \rightarrow T_3 \rightarrow T_2 \rightarrow T_4$.

-- Concurrency Control vs. Serializability Tests

- Testing a schedule for serializability *after* it has executed is a little too late!
- Goal – to develop concurrency control protocols that will assure serializability. They will generally not examine the precedence graph as it is being created; instead a protocol will impose a discipline that avoids nonseralizable schedules.
  Will study such protocols in Chapter 18.
- Tests for serializability help understand why a concurrency control protocol is correct.

End