Unit 4

Part-1

Transaction Processing Concepts and Theory

Original Content: Ramez Elmasri and Shamkant B. Navathe

Chapter Outline

1 Introduction to Transaction Processing

2 Transaction and System Concepts

3 Desirable Properties of Transactions

4 Characterizing Schedules based on Recoverability

5 Characterizing Schedules based on Serializability

6 Transaction Support in SQL

Introduction to Transaction Processing

- **Single-User System**:
  - At most one user at a time can use the system.
- **Multiuser System**:
  - Many users can access the system concurrently.
- **Concurrency**
  - **Interleaved processing**:
    - Concurrent execution of processes is interleaved in a single CPU
  - **Parallel processing**:
    - Processes are concurrently executed in multiple CPUs.

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** is a 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 AND WRITE OPERATIONS:

- 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:
  - 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.

Introduction to Transaction Processing

READ AND WRITE OPERATIONS (contd.):

- **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

(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:

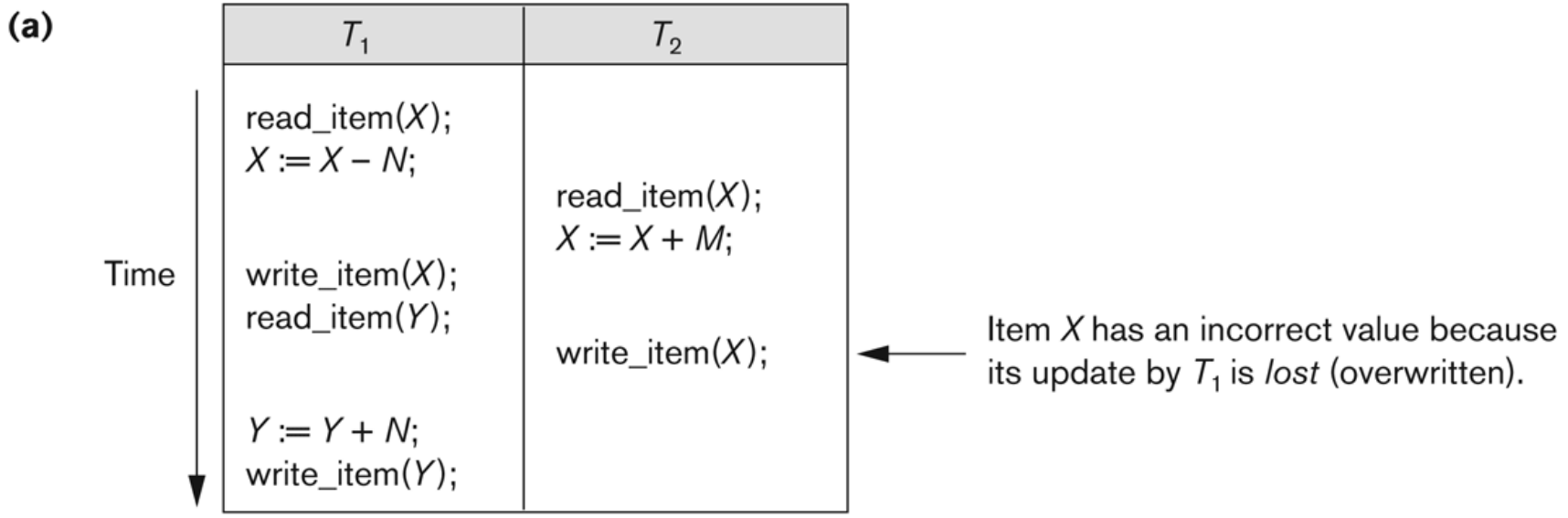
- **The Lost Update Problem**
  - This occurs when two transactions that access the same database items have their operations interleaved in a way that makes the value of some database item incorrect.

- **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.

Figure 17.3
Some problems that occur when concurrent execution is uncontrolled. (a) The lost update problem. (b) The temporary update problem. (c) The incorrect summary problem.

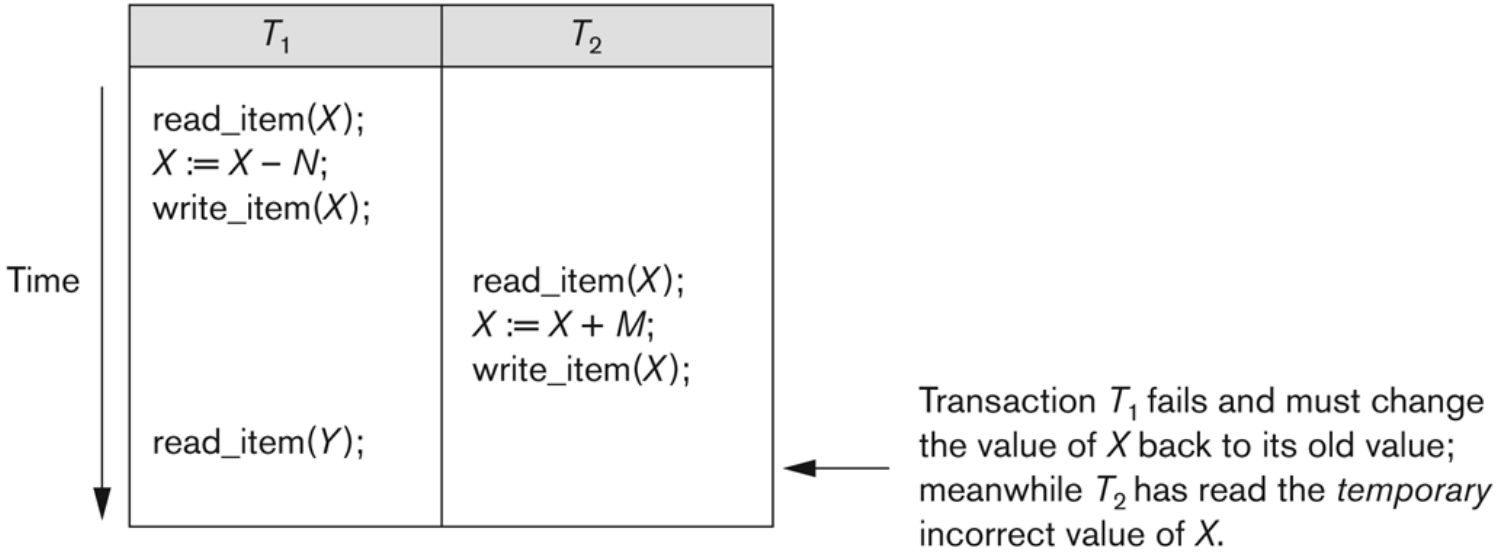


Concurrent execution is uncontrolled: (b) The temporary update problem.

Figure 17.3

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

(b)



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

Figure 17.3

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*).

Why recovery is needed:

(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.

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

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

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 (cont):
  - **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.

- 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 transition diagram

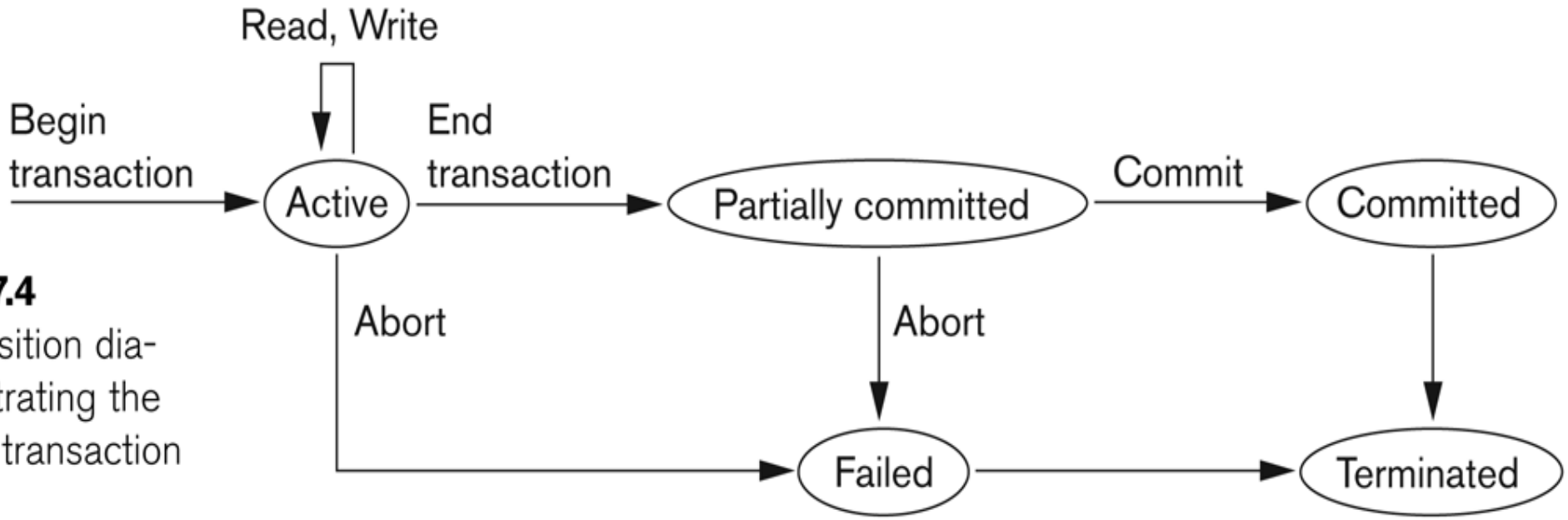


Figure 17.4
State transition diagram illustrating the states for transaction execution.

- 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.

- The System Log (cont):
  - 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:
  - Types of log record:
    - [start_transaction,T]: Records that transaction T has started execution.
    - [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.
    - [read_item,T,X]: Records that transaction T has read the value of database item X.
    - [commit,T]: Records that transaction T has completed successfully, and affirms that its effect can be committed (recorded permanently) to the database.
    - [abort,T]: Records that transaction T has been aborted.

- The System Log (cont):
  - Protocols for recovery that *avoid cascading rollbacks do not require that read operations be written to the system log*, whereas other protocols require these entries for recovery.
  - Strict protocols require simpler write entries that do not include new_value

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.

Commit Point of a Transaction:

- **Definition a Commit Point:**
  - 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.

Commit Point of a Transaction (cont):

- **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.

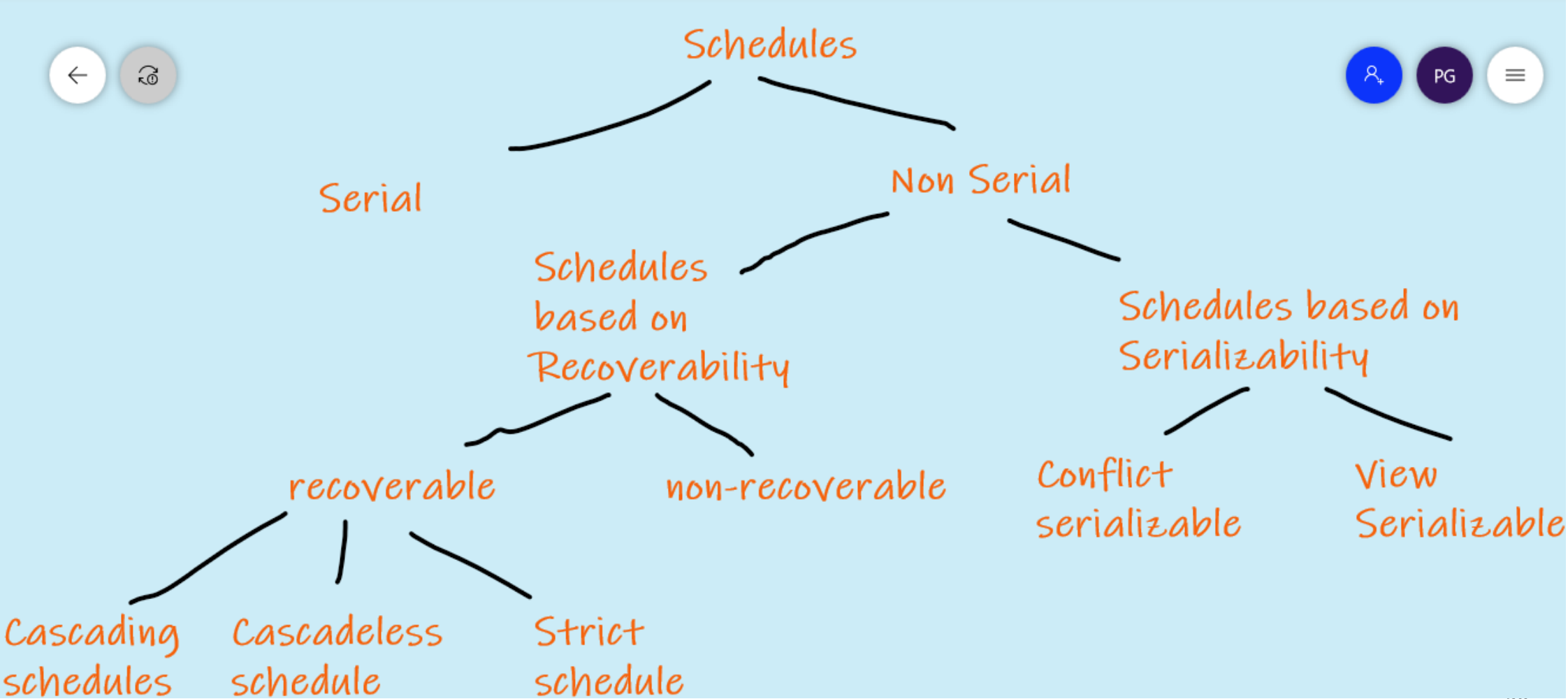
Schedules

- Schedules
- Serial Schedules and Non-serial Schedules
- Complete Schedules
- Serializable Schedules
- Classifying Schedules based on Recoverability
  - Recoverable and Non Recoverable Schedules
  - Cascading roll back schedules
  - Cascade less Schedules
  - Strict Schedules
- Classifying Schedules based on Serializability
  - Conflict serializable
  - Testing for Conflict Serializability

Schedule Definition

- **Transaction schedule or history**:
  - When transactions are executing concurrently in an interleaved fashion, the order of execution of operations from the various transactions forms (is known as) a transaction schedule (or history).
- A **schedule** (or **history**) S of n transactions T1, T2, …, Tn:
  - It is an ordering of the operations of the transactions subject to the constraint that, for each transaction Ti that participates in S, the operations of Ti in S must appear in the same order in which they occur in Ti.
  - Note, however, that operations from other transactions Tj can be interleaved with the operations of Ti in S.

Types of Schedules in Database



Serial Schedule 1

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

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

Serial Schedule 2

- A serial schedule where T_2 is followed by T_1

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

Serializable Schedule 3

- 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.

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)	
commit	
	read (B)
	$B := B + temp$
	write (B)
	commit

In Schedules 1, 2 and 3, the sum **A + B is preserved**.

Schedule 4

- The following concurrent schedule **does not preserve** the value of ***(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)	
commit	
	$B := B + temp$
	write (B)
	commit

Notations used for describing the schedules

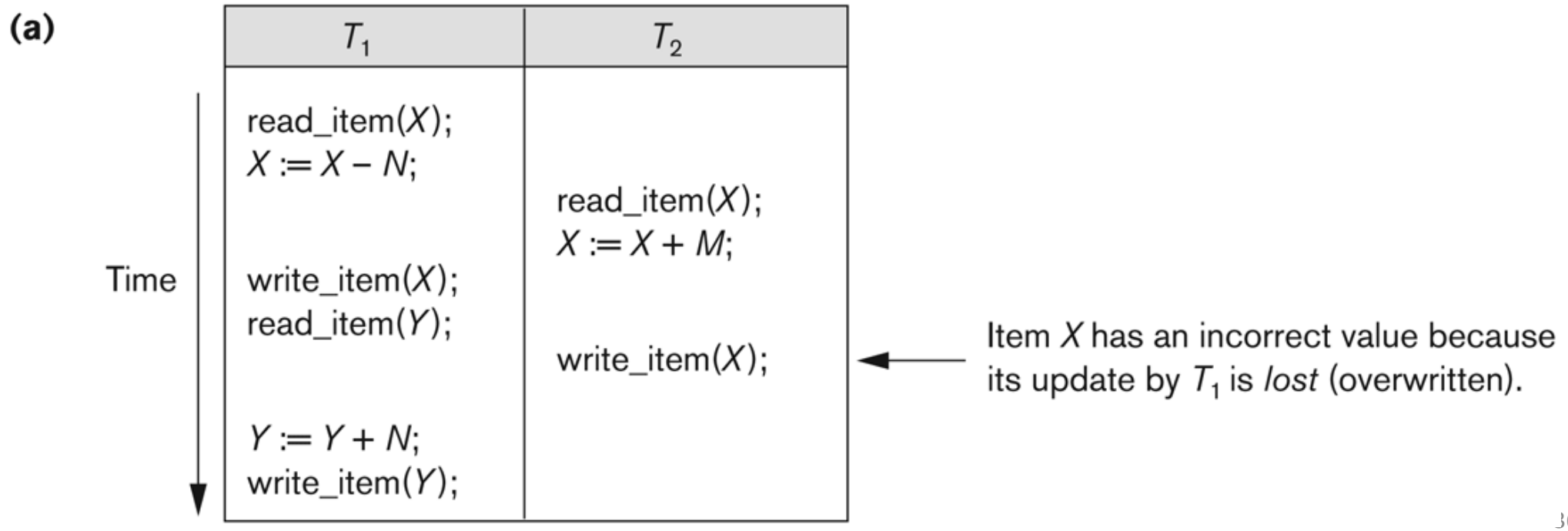
- For the purpose of recovery and concurrency control, we are mainly interested in the **read_item and write_item** operations of the transactions, as well as **the commit and abort operations**.
- A shorthand notation for describing a schedule uses the symbols *b, r, w, e, c, and a for the operations begin_transaction, read_item, write_item,* end_transaction, commit, and abort, respectively, and appends as a *subscript the* transaction id (transaction number) to each operation in the schedule.

Example – Shorthand notation for Schedule

Sa: r1(X); r2(X); w1(X); r1(Y); w2(X); w1(Y);a1;

Figure 17.3

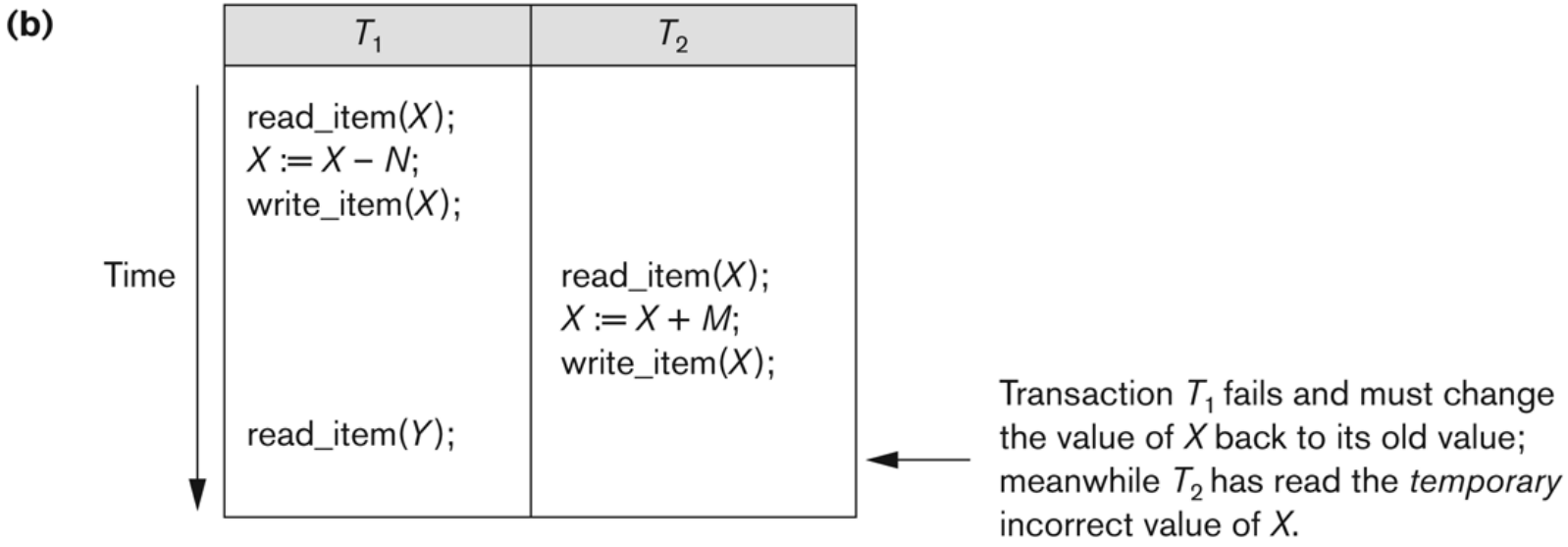
Some problems that occur when concurrent execution is uncontrolled. (a) The lost update problem. (b) The temporary update problem. (c) The incorrect summary problem.



Example – Shorthand notation for Schedule

- *Sb: r1(X); w1(X); r2(X); w2(X); r1(Y); a1;*

Figure 17.3
Some problems that occur when concurrent execution is uncontrolled. (a) The lost update problem. (b) The temporary update problem. (c) The incorrect summary problem.



Characterizing Schedules based on Recoverability

Schedules classified on recoverability:

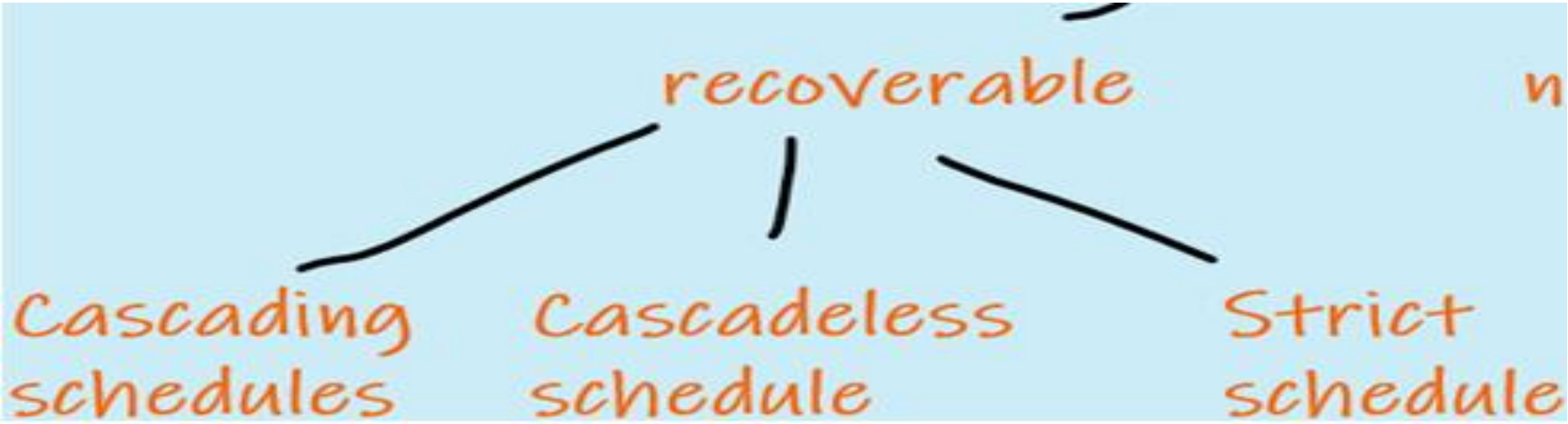
- **Recoverable schedule**:
  - Schedule where no transaction needs to be rolled back once the transaction is committed.
  - *This ensures that the durability property of* transactions is not violated.
  - A schedule where a committed transaction may have to be rolled back during recovery is called **non-recoverable** and hence should not be permitted by the DBMS.

Characterizing Schedules based on Recoverability

- **Recoverable schedule**:
  - A schedule S is recoverable if no transaction T in S commits until all transactions T' that have written an item that T reads have committed.
  - In other words, if some transaction T_j is reading value updated or written by some other transaction T_i, then the commit of T_j must occur after the commit of T_i.
  - S1: R1(x), **W1(x)**, R2(x), R1(y), R2(y), **W2(x)**, W1(y), **C1**, **C2**; (R)
  - *Sc: r1(X); w1(X); r2(X); r1(Y); w2(X); c2; a1; (NR)*

Characterizing Schedules based on Recoverability

- *Sd: r1(X); w1(X); r2(X); r1(Y); w2(X); w1(Y); c1; c2; (R)*
- *Se: r1(X); w1(X); r2(X); r1(Y); w2(X); w1(Y); a1; a2; (R but cascading Rollback)*
- *Sa': r1(X); r2(X); w1(X); r1(Y); w2(X); c2; w1(Y); c1; (R)*
- *Sb: r1(X); w1(X); r2(X); w2(X); r1(Y); a1;*



Characterizing Schedules based on Recoverability

- Because cascading rollback can be time-consuming—since numerous transactions can be rolled back it is important to characterize the schedules where this phenomenon is guaranteed not to occur.
- **Cascadeless schedule**:
  - A schedule is said to be **cascadeless, or to avoid cascading rollback, if every transaction in the schedule** reads only items that were written by committed transactions.
  - To satisfy this criterion, the *$r_2(X)$ command* in schedules *S_d and S_e must be postponed until after T_1 has committed (or* aborted), thus delaying *T_2 but ensuring no cascading rollback if T_1 aborts.*
  - *Sf : w1(X, 5); w2(X, 8); a1; (Cascadeless)*

Characterizing Schedules based on Recoverability

Schedules classified on recoverability (contd.):

- **Cascaded rollback Schedules**:
  - A schedule in which uncommitted transactions that read an item from a failed transaction must be rolled back.
  - *Se: r1(X); w1(X); r2(X); r1(Y); w2(X); w1(Y); a1; a2; (R but cascading Rollback)*
- **Strict Schedules:**
  - A schedule in which a transaction can neither read or write an item X until the last transaction that wrote X has committed.
  - The schedule *Sf is cascadeless, but it is not* a strict schedule, since it permits *T2 to write item X even though the transaction T1* that last wrote *X had not yet committed*

Characterizing Schedules based on Recoverability

- r1(a);w1(a);w2(a);c1;r2(a);c2;

Example: Consider the following schedule involving two transactions T_1 and T_2.

T_1	T_2
R(A)	
W(A)	
	W(A)
commit	
	R(A)
	commit

Characterizing Schedules based on Recoverability

- This schedule is cascadeless. Since the updated value of **A** is read by T_2 only after the updating transaction i.e. T_1 commits.

Example: Consider the following schedule involving two transactions T_1 and T_2.

T_1	T_2
R(A)	
W(A)	
	W(A)
commit	
	R(A)
	commit

Characterizing Schedules based on Recoverability

- r1(a);r2(a);w1(a);c1;w2(a);r2(a);c2

Example: Consider the following schedule involving two transactions T_1 and T_2.

T_1	T_2
R(A)	
	R(A)
W(A)	
commit	
	W(A)
	R(A)
	commit

Characterizing Schedules based on Recoverability

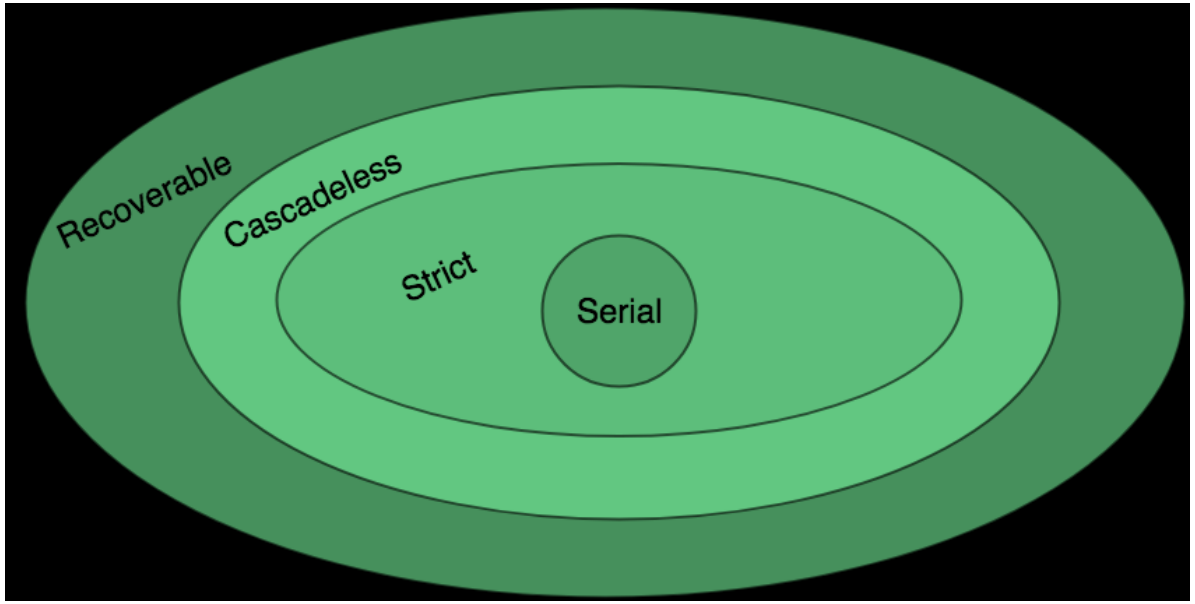
- This is a strict schedule since T_2 reads and writes A which is written by T_1 only after the commit of T_1.

Example: Consider the following schedule involving two transactions T_1 and T_2.

T_1	T_2
R(A)	
	R(A)
W(A)	
commit	
	W(A)
	R(A)
	commit

Characterizing Schedules based on Recoverability

- It is important to note that any strict schedule is also cascadeless, and any cascadeless schedule is also recoverable.
- The cascadeless schedules will be a subset of the recoverable schedules, and the strict schedules will be a subset of the cascadeless schedules. Thus, all strict schedules are cascadeless, and all cascadeless schedules are recoverable.



Characterizing Schedules based 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. Otherwise, the schedule is called nonserial schedule.
  - Therefore, in a serial schedule, only one transaction at a time is active—the commit (or abort) of the active transaction initiates execution of the next transaction. No interleaving occurs in a serial schedule.

Characterizing Schedules based on Serializability

- Problems with Serial Schedules
  - The problem with serial schedules is that they limit concurrency by prohibiting interleaving of operations. In a serial schedule, if a transaction waits for an I/O operation to complete, we cannot switch the CPU processor to another transaction, thus wasting valuable CPU processing time.
  - Additionally, if some transaction *T is* long, the other transactions must wait for *T to complete all its operations before* starting.
  - Hence, serial schedules are *unacceptable in practice. However, if we can* determine which other schedules are *equivalent to a serial schedule, we can allow* these schedules to occur.
- Serializable schedule:
  - A schedule S is serializable if it is equivalent to some serial schedule of the same n transactions

Concept of *equivalence of schedules*

- Result equivalent:
  - Two schedules are called result equivalent if they produce the same final state of the database.
- Conflict equivalent:
  - Two schedules are said to be conflict equivalent if the order of any two conflicting operations is the same in both schedules.
- Conflict serializable:
  - A schedule S is said to be conflict serializable if it is conflict equivalent to some serial schedule S'.
  - In such a case, we can reorder the *nonconflicting operations in S until we form the* equivalent serial schedule *S'.*

Conflicting Operations in a Schedule

- Two operations in a schedule are said to **conflict if they satisfy all three of the following conditions:**

(1) they belong to *different transactions;*

(2) *they access the same item X; and*

(3) *at least one of the operations* is a write_item(*X).*

Conflicting Operations in a Schedule

- For example, in a schedule S, (Examples of Conflicting Operations)
  - *the operations r1(X) and w2(X)* conflict,
  - the operations *r2(X) and w1(X),* ***(Read-Write conflict)***
  - *the operations w1(X) and w2(X).* ***(Write-Write conflict)***
- However, the examples of non-conflicting operations
  - *r1(X) and r2(X) do not conflict, since they are both read* operations;
  - the operations *w2(X) and w1(Y) do not conflict because they operate on* distinct data items *X and Y; and*
  - *the operations r1(X) and w1(X) do not conflict* because they belong to the same transaction.
- **Intuitively, two operations are conflicting if changing their order can result in a different outcome.**

Serializable Schedules

- Being serializable is not the same as being serial
- Being serializable implies that the schedule is a correct schedule.
  - It will leave the database in a consistent state.
  - The interleaving is appropriate and will result in a state as if the transactions were serially executed, yet will achieve efficiency due to concurrent execution.
- Serializability is hard to check.
  - Interleaving of operations occurs in an operating system through some scheduler
  - Difficult to determine beforehand how the operations in a schedule will be interleaved.

Serializable Schedules

Practical approach:

- Come up with methods (protocols) to ensure serializability.
- It's not possible to determine when a schedule begins and when it ends.
  - Hence, we reduce the problem of checking the whole schedule to checking only a **committed project** of the schedule (i.e. operations from only the committed transactions.)
- Current approach used in most DBMSs:
  - Use of locks with two phase locking

Conflict Serializability

Testing for conflict serializability: Algorithm :

- The algorithm looks at only the read_item and write_item operations in a schedule
- to construct a **precedence graph (or serialization graph), which is a directed graph**
- *G = (N, E) that consists of a set of nodes N = {T_1, T_2, … , T_n } and a set of directed* edges *E = {e_1, e_2, … , e_m }.*
- *There is one node in the graph for each transaction T_i in* the schedule. Each edge *e_i in the graph is of the form ($T_j \rightarrow T_k$), 1 ≤ j ≤ n, 1 ≤ k ≤ n,* where *T_j is the **starting node of e_i and T_k is the ending node of ei.***
- ***Such an edge*** from node *T_j to node T_k is created by the algorithm if a pair of conflicting operations* exist in *T_j and T_k and*
- *the conflicting operation in T_j appears in the schedule before* the *conflicting operation in T_k.*

Testing for conflict serializability: Algorithm

Testing for conflict serializability: Algorithm :

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.

6. Topological Sorting is used to determine the equivalent serial Schedule from the precedence graph

Example Schedules

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

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

Schedule B

(c)

Time

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

Schedule C

Time

T_1	T_2

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

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

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

Schedule D

Figure 20.5
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) Two nonserial schedules C and D with interleaving of operations.

Constructing the Precedence Graphs

- Constructing the precedence graphs for schedules A and D from Figure 17.5 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).

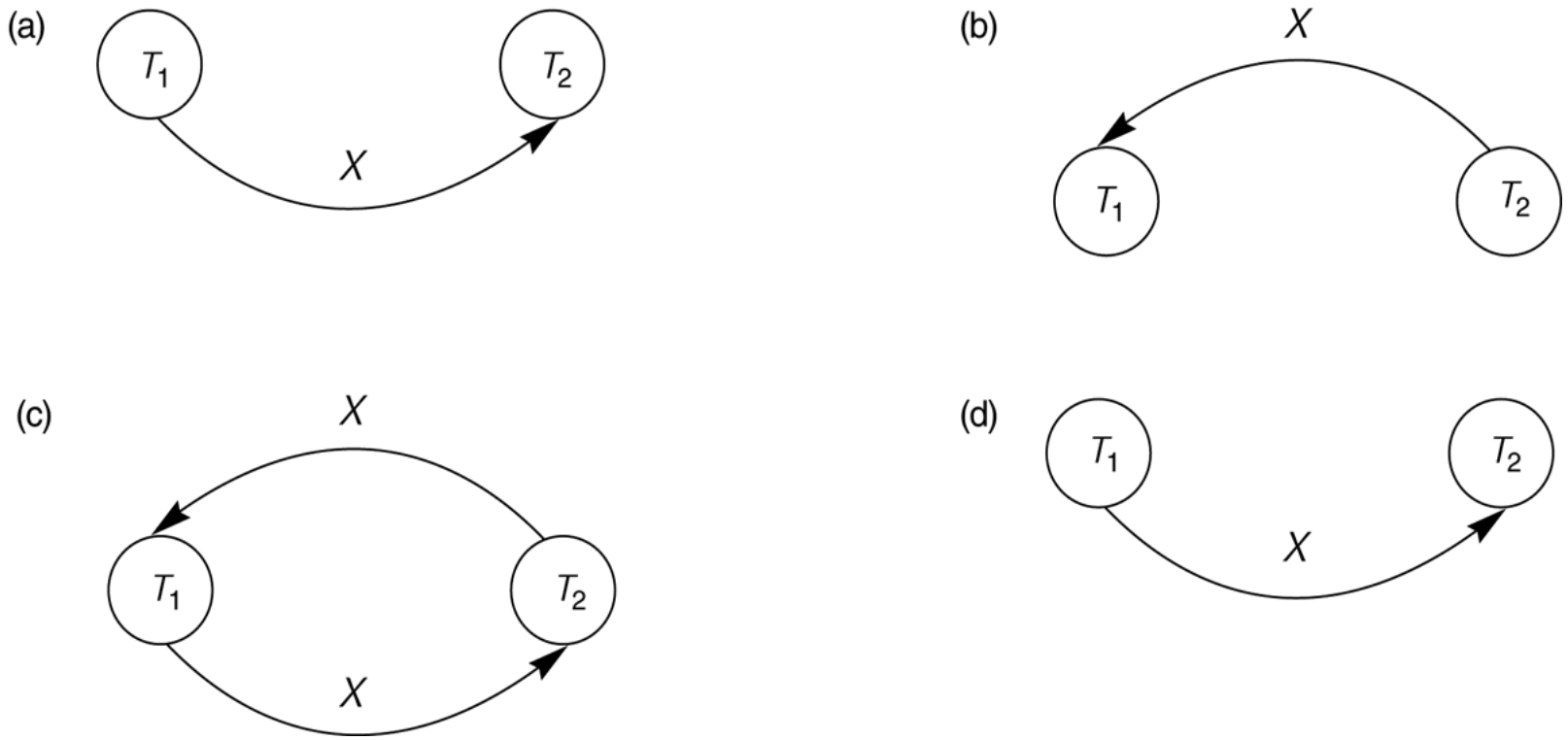


Figure 17.8
Another example of serializability testing. (a) The read and write operations of three transactions T_1, T_2, and T_3. (b) Schedule E. (c) Schedule *F*.

(a)

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

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

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

Figure 17.8
Another example of serializability testing. (a) The read and write operations of three transactions T_1, T_2, and T_3. (b) Schedule E. (c) Schedule *F*.

(b)

Time

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

Schedule E

Figure 17.8
Another example of serializability testing. (a) The read and write operations of three transactions T_1, T_2, and T_3. (b) Schedule E. (c) Schedule *F*.

(c)

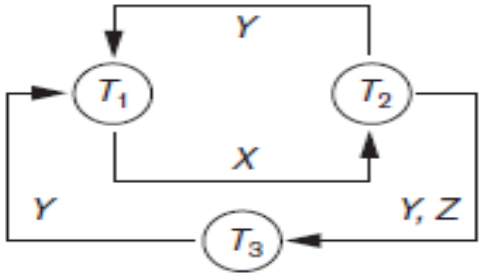
Time

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

Schedule F

Figure 20.8 (continued)
Another example of serializability testing. (d) Precedence graph for schedule E. (e) Precedence graph for schedule F. (f) Precedence graph with two equivalent serial schedules.

(d)



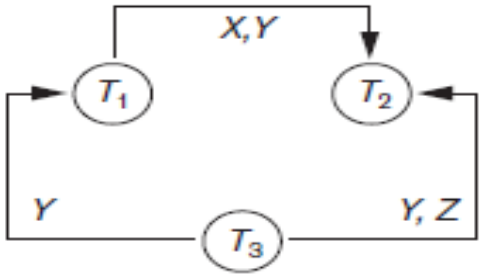
Equivalent serial schedules

None

Reason

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

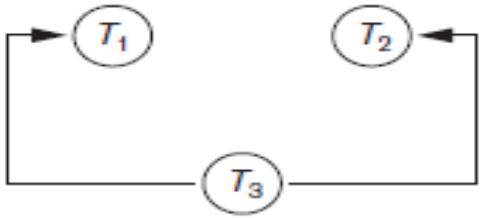
(e)



Equivalent serial schedules

$T_3 \rightarrow T_1 \rightarrow T_2$

(f)



Equivalent serial schedules

$T_3 \rightarrow T_1 \rightarrow T_2$

$T_3 \rightarrow T_2 \rightarrow T_1$

Other Types of Equivalence of Schedules

- Under special **semantic constraints**, schedules that are otherwise not conflict serializable may work correctly.
  - Using commutative operations of addition and subtraction (which can be done in any order) certain non-serializable transactions may work correctly

Other Types of Equivalence of Schedules (contd.)

- Example: bank credit / debit transactions on a given item are **separable** and **commutative**.
  - Consider the following schedule S for the two transactions:
  - Sh : r1(X); w1(X); r2(Y); w2(Y); r1(Y); w1(Y); r2(X); w2(X);
  - Using conflict serializability, it is **not serializable**.
  - However, if it came from a (read,update, write) sequence as follows:
    - r1(X); X := X – 10; w1(X); r2(Y); Y := Y – 20;r1(Y);
    - Y := Y + 10; w1(Y); r2(X); X := X + 20; (X);
  - Sequence explanation: debit, debit, credit, credit.
  - It is a *correct schedule for the given semantics*

Transaction Support in SQL2

- A **single** SQL statement is always considered to be **atomic**.
  - Either the statement completes execution without error or it fails and leaves the database unchanged.
- With SQL, there is no explicit Begin Transaction statement.
  - Transaction initiation is done implicitly when particular SQL statements are encountered.
- Every transaction must have an explicit end statement, which is either a COMMIT or ROLLBACK.

Transaction Support in SQL2

Characteristics specified by a SET TRANSACTION statement in SQL2:

- **Access mode**:
  - READ ONLY or READ WRITE.
    - The default is READ WRITE unless the isolation level of READ UNCOMITTED is specified, in which case READ ONLY is assumed.
- **Diagnostic size** n, specifies an integer value n, indicating the number of conditions that can be held simultaneously in the diagnostic area.

Transaction Support in SQL2

Characteristics specified by a SET TRANSACTION statement in SQL2 (contd.):

- **Isolation level** <isolation>, where <isolation> can be READ UNCOMMITTED, READ COMMITTED, REPEATABLE READ or SERIALIZABLE. The default is SERIALIZABLE.
  - With SERIALIZABLE: the interleaved execution of transactions will adhere to our notion of serializability.
  - However, if any transaction executes at a lower level, then serializability may be violated.

Transaction Support in SQL2

Potential problem with lower isolation levels:

- **Dirty Read**:
  - Reading a value that was written by a transaction which failed.
- **Nonrepeatable Read**:
  - Allowing another transaction to write a new value between multiple reads of one transaction.
  - A transaction T1 may read a given value from a table. If another transaction T2 later updates that value and T1 reads that value again, T1 will see a different value.
    - Consider that T1 reads the employee salary for Smith. Next, T2 updates the salary for Smith. If T1 reads Smith's salary again, then it will see a different value for Smith's salary.

Transaction Support in SQL2

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

Transaction Support in SQL2

- Sample SQL transaction:

```
EXEC SQL whenever sqlerror go to UNDO;
EXEC SQL SET TRANSACTION
          READ WRITE
          DIAGNOSTICS SIZE 5
          ISOLATION LEVEL SERIALIZABLE;
EXEC SQL INSERT
          INTO EMPLOYEE (FNAME, LNAME, SSN, DNO, SALARY)
          VALUES ('Robert','Smith','991004321',2,35000);
EXEC SQL UPDATE EMPLOYEE
          SET SALARY = SALARY * 1.1
          WHERE DNO = 2;
EXEC SQL COMMIT;
          GOTO  THE_END;
UNDO: EXEC SQL ROLLBACK;
THE_END:  ...
```

Transaction Support in SQL2

- Possible violation of serializabilty:

Isolation level	Type of Violation: Dirty read	nonrepeatable read	phantom
READ UNCOMMITTED	yes	yes	yes
READ COMMITTED	no	yes	yes
REPEATABLE READ	no	no	yes
SERIALIZABLE	no	no	no

Summary

- Transaction and System Concepts
- Desirable Properties of Transactions
- Characterizing Schedules based on Recoverability
- Characterizing Schedules based on Serializability
- Transaction Support in SQL