<u>Module 6</u>
Transaction Processing Concepts

Transaction

- The concept of transaction provides a mechanism for describing logical units of database processing.
  Example : Transfer amount from one account to another.

- **Transaction processing systems are systems with large databases and hundreds of** concurrent users executing database transactions.

- Examples of such systems include **airline reservations, banking, credit card processing, on-line retail purchasing, stock markets, supermarket checkouts**, and many other applications.

- These systems require **high availability and fast response time** for hundreds of concurrent users.

Introduction to Transaction Processing

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

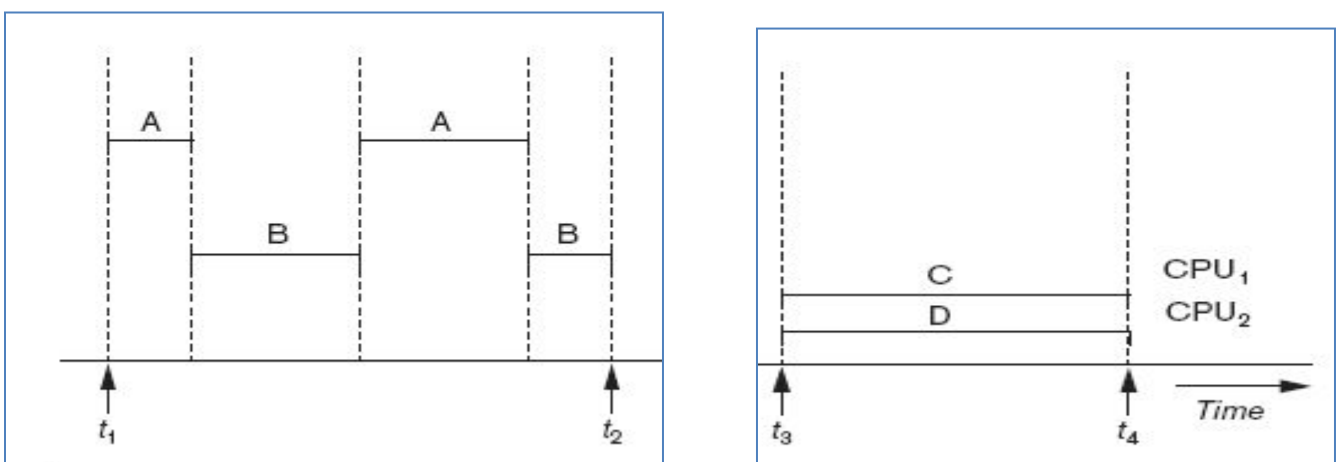


Figure 21.1
Interleaved processing versus parallel processing of concurrent transactions.

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 :**size of data item- a field, a record , or a whole disk block

Basic operations are **read** and **write**

read_item(X): Reads a database item named X into a program variable. To simplify our variable is also named X.

write_item(X): the program 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**:

  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.

- READ AND WRITE OPERATIONS (contd.):

- **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).

Why Concurrency Control is needed ?

Two sample transactions

FIGURE 17.2 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?

1. 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.

For example, if *X = 80 at the start (originally* there were 80 reservations on the flight),*N = 5 (T1 transfers 5 seat reservations from* the flight corresponding to *X to the flight corresponding to Y), and M = 4 (T2* reserves 4 seats on *X), the final result should be X =*

Why Concurrency Control is needed?

1. 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.

For example, if *X = 80 at the start (originally* there were 80 reservations on the flight),*N = 5 (T1 transfers 5 seat reservations from* the flight corresponding to *X to the flight corresponding to Y), and M = 4 (T2* reserves 4 seats on *X), the final result should be X = 79.*

Introduction to Transaction Processing

Why Concurrency Control is needed?

1. 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.

For example, if *X = 80 at the start (originally* there were 80 reservations on the flight),*N = 5 (T1 transfers 5 seat reservations from* the flight corresponding to *X to the flight corresponding to Y), and M = 4 (T2* reserves 4 seats on *X), the final result should be X = 79.*

Figure 17.3
Some problems that occur when concurrent execution is uncontrolled. (a) The lost update problem. (

(a)

Time

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

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

Why Concurrency Control is needed?

1. 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.

For example, if *X = 80 at the start (originally* there were 80 reservations on the flight),*N = 5 (T1 transfers 5 seat reservations from* the flight corresponding to *X to the flight corresponding to Y), and M = 4 (T2* reserves 4 seats on *X), the final result should be X = 79.*

However, in the interleaving of operations shown in Figure 17.3(a), it is X = 84 because the update in T1 that removed the five seats from X was lost.

Figure 17.3
Some problems that occur when concurrent execution is uncontrolled. (a) The lost update problem. (

(a)

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); ← Item X has an incorrect value because its update by T_1 is *lost* (overwritten).
$Y := Y + N$; write_item(Y);	

Introduction to Transaction Processing

Why Concurrency Control is needed:

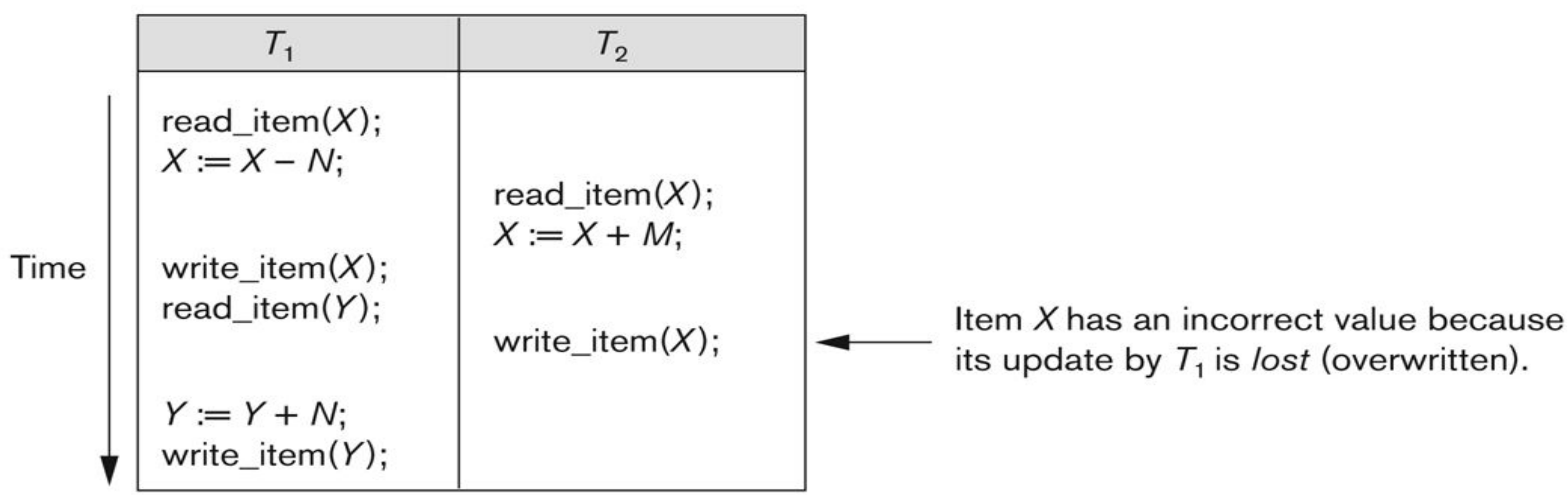
1. The Lost Update Problem

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.

For example, if *X = 80 at the start (originally* there were 80 reservations on the flight),*N = 5 (T1 transfers 5 seat reservations from* the flight corresponding to *X to the flight corresponding to Y), and M = 4 (T2* reserves 4 seats on *X), the final result should be X = 79.*
However, in the interleaving* of operations shown in Figure 17.3(a), it is *X = 84 because the update in T1 that* removed the five seats from *X was lost.

Figure 17.3
Some problems that occur when concurrent execution is uncontrolled. (a) The lost update problem.

(a)



Introduction to Transaction Processing

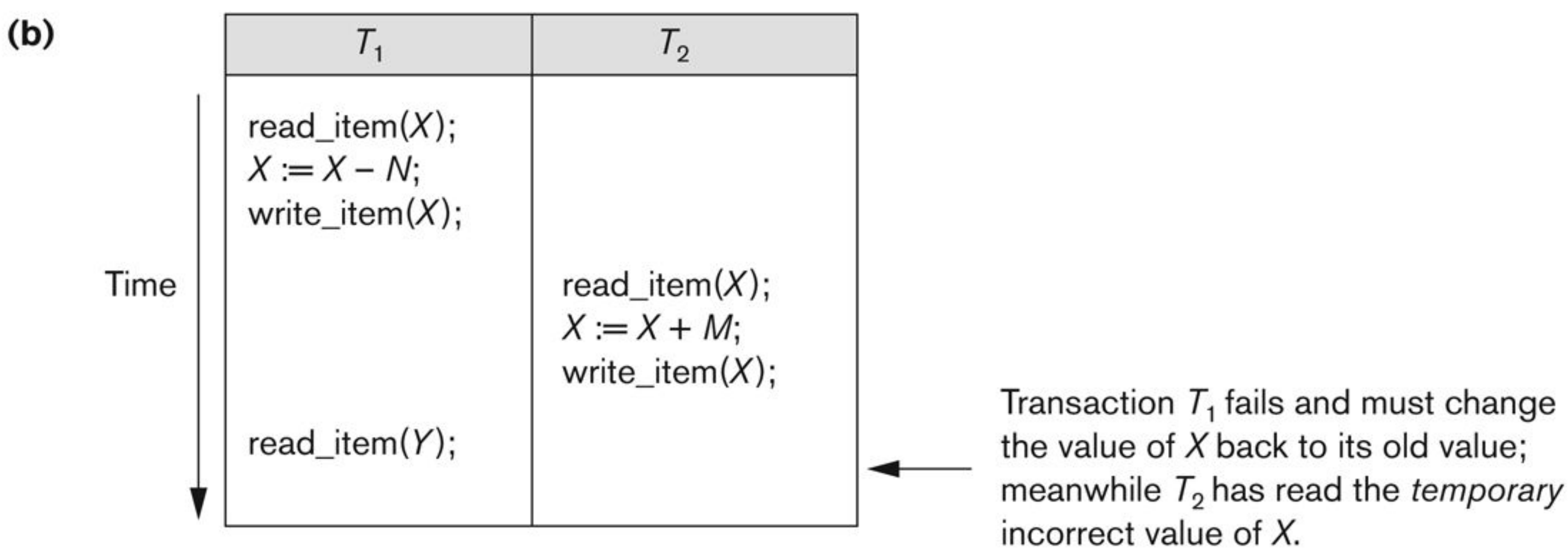
Why Concurrency Control is needed:

2. 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.

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.



Introduction to Transaction Processing

Why Concurrency Control is needed:

3. 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.

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.

For example, suppose that a transaction *T3 is calculating the total number of reservations on all the flights;*

(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 Concurrency Control is needed:

4. Unrepeatable read

unrepeatable read, where a transaction T reads the same item twice and the item is changed by another transaction T1 between the two reads.

Hence, T receives different values for its two reads of the same item.

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 (Contd.):

(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.

Why **recovery** is needed (contd.):

(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 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

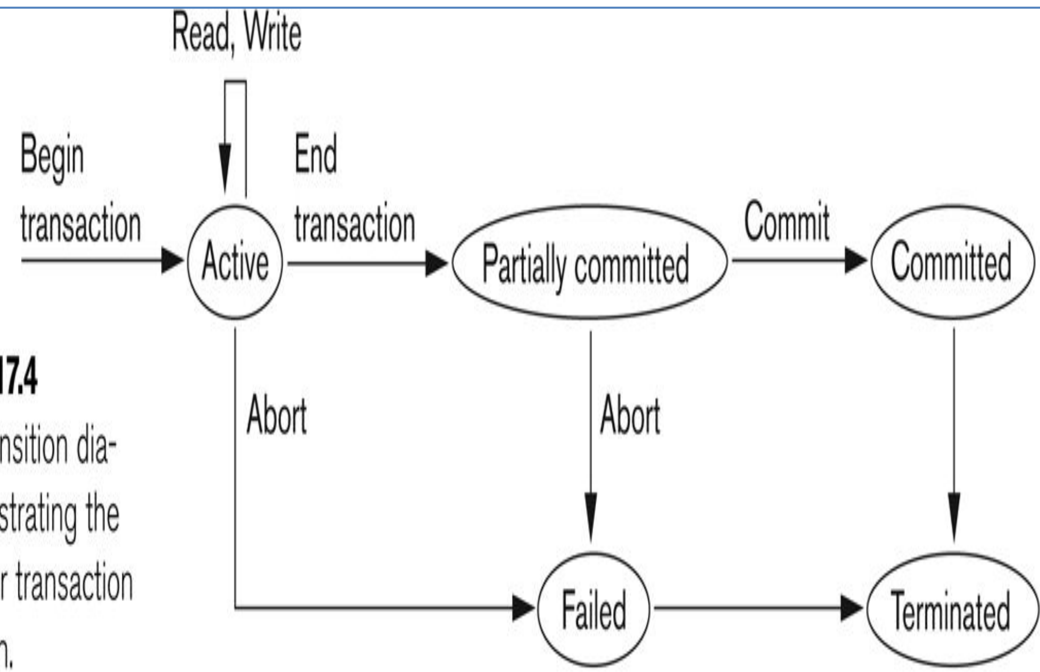


Figure 17.4
State transition diagram illustrating the states for transaction execution.

Transaction and System Concepts

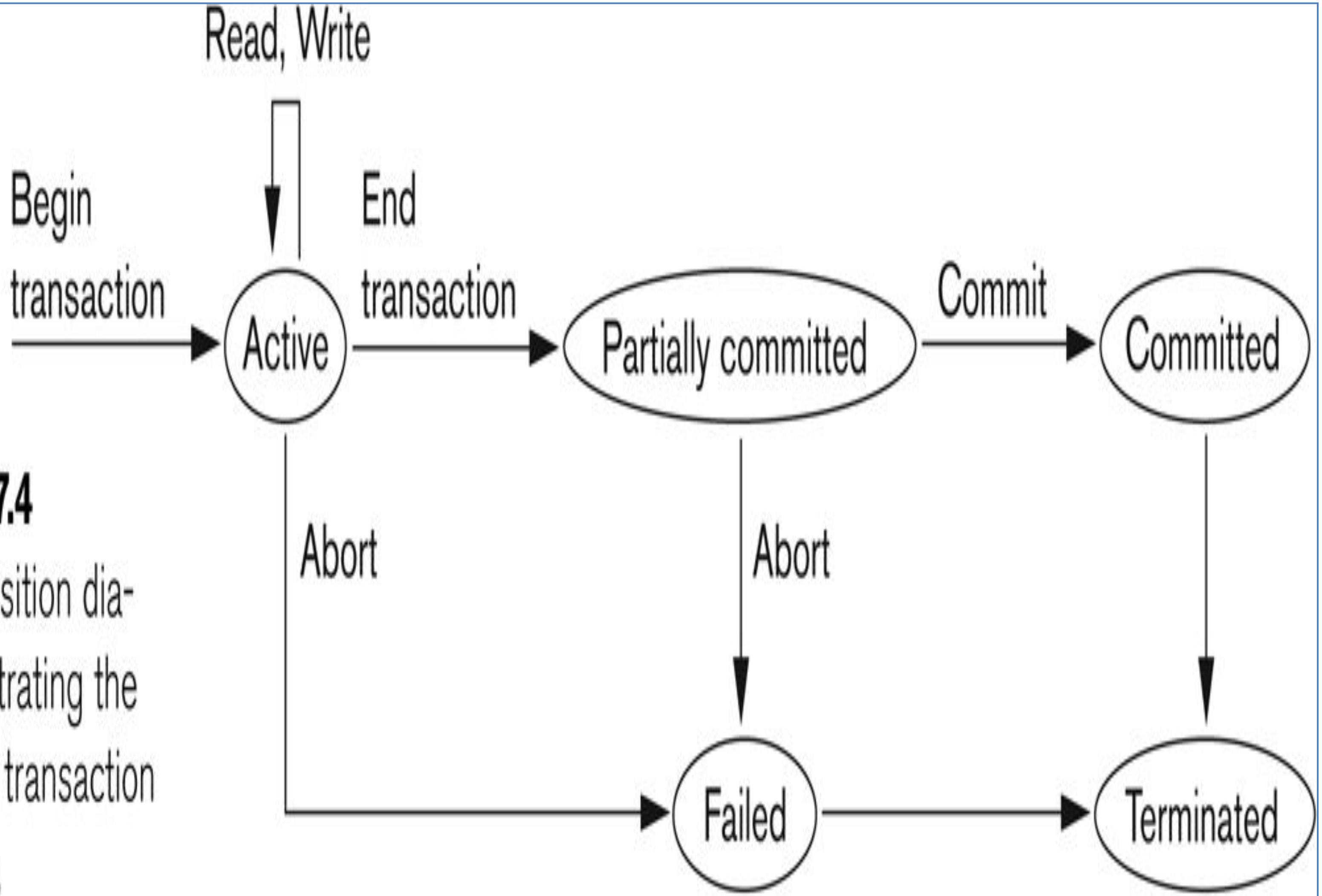


Figure 17.4
State transition diagram illustrating the states for transaction execution.

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.

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.

- **Durability or permanency**: Once a transaction changes the database and the changes are committed, these changes must never be lost because of subsequent failure.

Characterizing Schedules based on Recoverability

- **Transaction Schedule or History**:

  When transactions are executing concurrently in an interleaved fashion, the order of execution of operations from the various transactions forms what 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 T1 in S must appear in the same order in which they occur in T1.

  Note, however, that operations from other transactions Tj can be interleaved with the operations of Ti in S.

Characterizing Schedules based on Recoverability

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 for above figure can be written as follows:

Characterizing Schedules based on Recoverability

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 for above figure can be written as follows:

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

Characterizing Schedules based on Recoverability

(b)

Time

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

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

Schedule for above figure can be written as follows:

Characterizing Schedules based on Recoverability

(b)

Time

T_1	T_2

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

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

read_item(Y);	

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

Schedule for above figure can be written as follows:

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

Characterizing Schedules based on Recoverability

- **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).

For example,
In schedule *Sa,*
the operations r1(X) and w2(X)
the operations r2(X) and w1(X)
the operations w1(X) and w2(X)

However,
the operations r1(X) and r2(X)

w2(X) and *w1(Y)*

the operations *r1(X) and w1(X)*

Characterizing Schedules based on Recoverability

- **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).

For example,
In schedule *Sa,*
the operations r1(X) and w2(X) – conflict
the operations r2(X) and w1(X) – conflict
the operations w1(X) and w2(X) – conflict

However,
the operations r1(X) and r2(X)

w2(X) and *w1(Y)*

the operations *r1(X) and w1(X)*

Characterizing Schedules based on Recoverability

- **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).

For example,
In schedule *Sa,*
the operations r1(X) and w2(X) – conflict
the operations r2(X) and w1(X) – conflict
the operations w1(X) and w2(X) – conflict

However,
the operations r1(X) and r2(X)
do not conflict, -- since they are both read operations;

w2(X) and *w1(Y)*
do not conflict -- because they operate on distinct data items X and Y;

the operations *r1(X) and w1(X)*
do not conflict -- because they belong to the same transaction.

Two operations are conflicting if changing their order can result in a different outcome. For example

if we change the order of the two operations
r1(X);w2(X)
to
w2(X); r1(X), then ….

If we change the order of two operations
w1(X); w2(X)
to
w2(X); w1(X)

Two operations are conflicting if changing their order can result in a different outcome. For example

if we change the order of the two operations
r1(X);w2(X)
to
w2(X); r1(X), then ….

read-write conflict

If we change the order of two operations
w1(X); w2(X)
to
w2(X); w1(X)

write-write conflict

Characterizing Schedules based on Recoverability

Schedules classified on recoverability:

Recoverable schedule:

Once Transaction T is committed, it should never be necessary to rollback.

“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”.

Example: T1 reads the content written by T2 & T3. So Unless and until T2 & T3 not committed, T1 can not commit. i.e. T1 can commit after T2 & T3 committed.

Cascadeless schedule:

One where every transaction reads only the items that are written by committed transactions.

Characterizing Schedules based on Recoverability

Schedules classified on recoverability (contd.):

Schedules requiring cascaded rollback:

A schedule in which uncommitted transactions that read an item from a failed transaction must be rolled back.

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.

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 "non serial schedule".

Serializable schedule:

A schedule S is serializable if it is equivalent to some serial schedule of the same n transactions.

Figure 21.5
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.

(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

(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

Characterizing Schedules based on Serializability

Result equivalent:

Two schedules are called result equivalent if they produce the same final state of the database.

Example: Schedule D result equivalent to A and B

Characterizing Schedules based on Serializability

Result equivalent:

Two schedules are called result equivalent if they produce the same final state of the database.

Example: Schedule D result equivalent to A and B

Problem:

S1	S2
read_item(X);	read_item(X);
X := X + 10;	X := X * 1.1;
write_item(X);	write_item(X);

Characterizing Schedules based on Serializability

Result equivalent:

Two schedules are called result equivalent if they produce the same final state of the database.

Example: Schedule D result equivalent to A and B

Problem: true for X=100 but not in general

S1	S2
read_item(X);	read_item(X);
X := X + 10;	X := X * 1.1;
write_item(X);	write_item(X);

Characterizing Schedules based on Serializability

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’.

Example: Schedule D is conflict serializable to Schedule A.

Characterizing Schedules based on Serializability

* Being serializable is <u>not</u> the same as being serial
* Being serializable implies that the schedule is a <u>correct</u> 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.

Characterizing Schedules based on Serializability

Serializability is hard to check.

Interleaving of operations occurs in an operating system through some scheduler

-Difficult to determine beforehand how the operations in a schedule will be interleaved.

Characterizing Schedules based on Serializability

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

View equivalence:

A less restrictive definition of equivalence of schedules.

- Two schedules are said to be view equivalent if the following three conditions hold:

1. The same set of transactions participates in S and S', and S and S' include the same operations of those transactions.

2. For any operation Ri(X) of Ti in S, if the value of X read by the operation has been written by an operation Wj(X) of Tj (or if it is the original value of X before the schedule started), the same condition must hold for the value of X read by operation Ri(X) of Ti in S'.

3. If the operation Wk(Y) of Tk is the last operation to write item Y in S, then Wk(Y) of Tk must also be the last operation to write item Y in S'.

Characterizing Schedules based on Serializability

View serializability:

Definition of serializability based on view equivalence.

A schedule is *view serializable* if it is *view equivalent* to a serial schedule.

Characterizing Schedules based on Serializability

Testing for Conflict Serializability of a Schedule.

Algorithm - Testing Conflict Serializability of a Schedule *S*

1. For each transaction *Ti* participating in schedule *S,* create a node labeled *Ti* in the precedence graph.

2. For each case in *S* where *Tj* executes a read_item(*X*) after *Ti* executes a write_item(*X*), create an edge (*Ti*→ *Tj*) in the precedence graph.

3. For each case in *S* where *Tj* executes a write_item(*X*) after *Ti* executes a read_item(*X*), create an edge (*Ti*→*Tj*) in the precedence graph.

4. For each case in *S* where *Tj* executes a write_item(*X*) after *Ti* executes a write_item(*X*), create an edge (*Ti*→ *Tj*) in the precedence graph.

5. The schedule *S* is serializable if and only if the precedence graph has no cycles.

Figure 21.5
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.

(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

Constructing the Precedence Graphs

(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

Constructing the Precedence Graphs

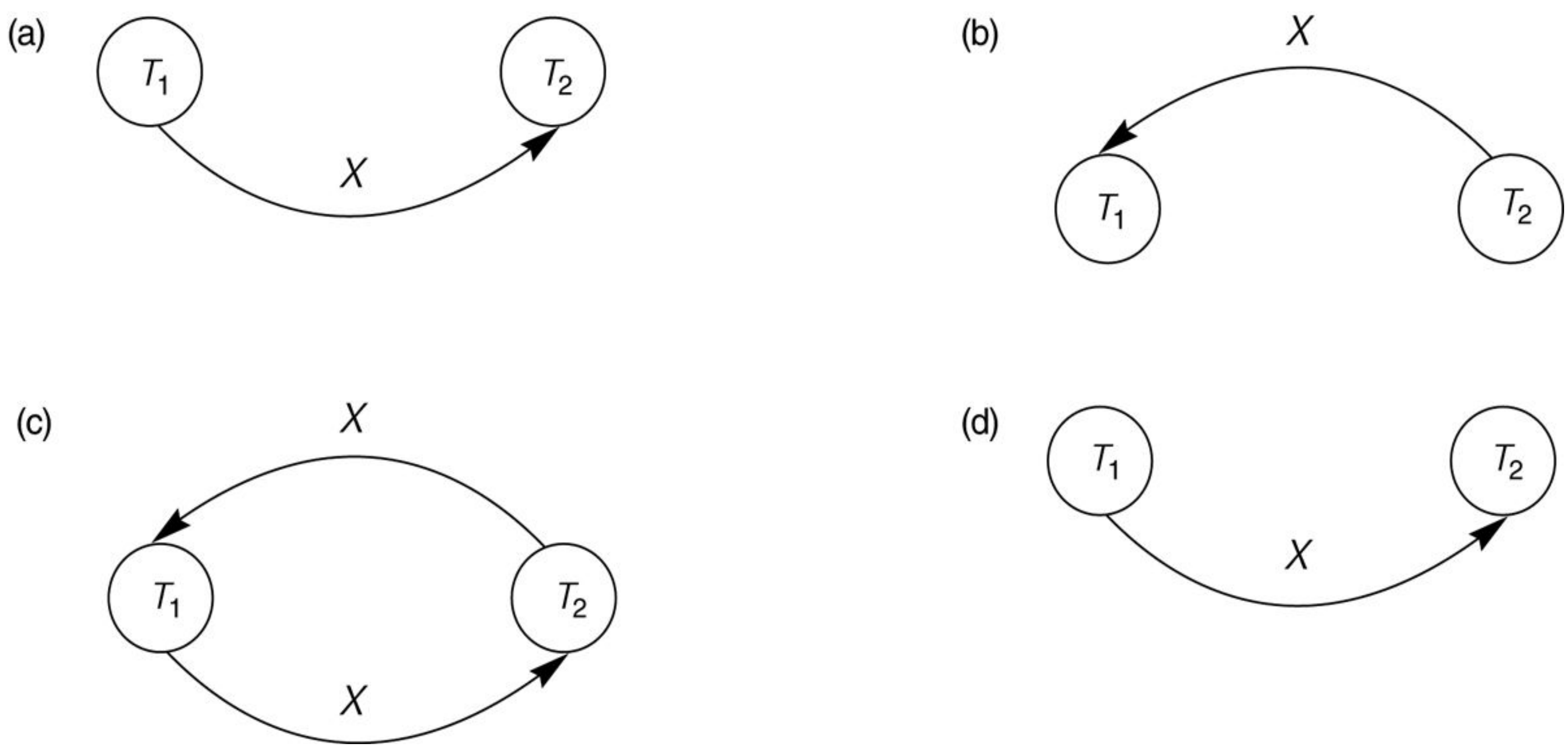
FIGURE 17.7 Constructing the precedence graphs for schedules A and D from Figure 17.5 to test for conflict serializability.

(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).



Another example of Serializability Testing

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);

Another Example of Serializability Testing

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

Another Example of Serializability Testing

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 ↓

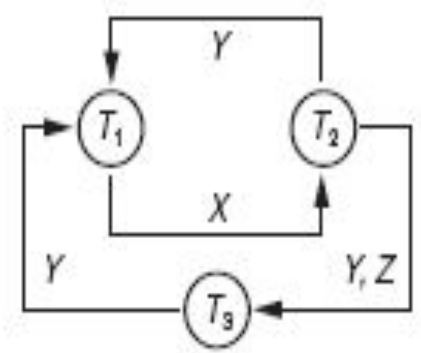
Transaction T_1	Transaction T_2	Transaction T_3

	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

(d)



Equivalent serial schedules

None

Reason

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

Another Example of Serializability Testing

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

Another Example of Serializability Testing

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)

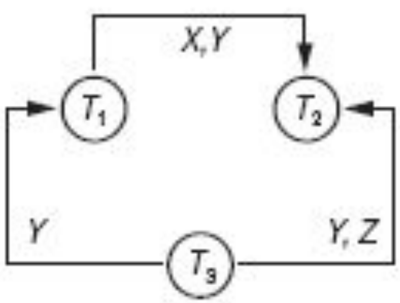
Time

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

(e)



Equivalent serial schedules

$T_3 \rightarrow T_1 \rightarrow T_2$

Characterizing Schedules based on Serializability

Other Types of Equivalence of Schedules (contd.)

Example: bank credit / debit transactions on a given item are **separable** and **commutative**.

(Consider the following transactions, each of which may be used to transfer an amount of money between two bank accounts):

$T1$: $r1(X)$; $X := X - 10$; $w1(X)$; $r1(Y)$; $Y := Y + 10$; $w1(Y)$;
$T2$: $r2(Y)$; $Y := Y - 20$; $w2(Y)$; $r2(X)$; $X := X + 20$; $w2(X)$;

Consider the following non-serializable schedule *Sh* for the two transactions:
Sh: $r1(X)$; $w1(X)$; $r2(Y)$; $w2(Y)$; $r1(Y)$; $w1(Y)$; $r2(X)$; $w2(X)$

Summary

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

Thank You