Module:6 – Transaction management

Dr. Parimala M, SCORE

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

1 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 (2)

- **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 (3)

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.

Introduction to Transaction Processing (4)

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.

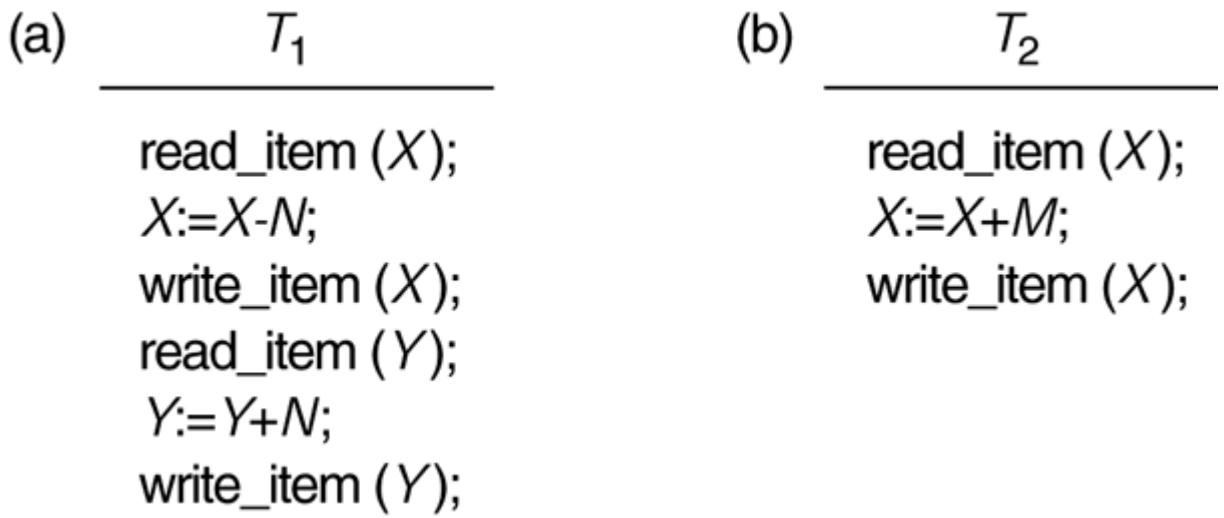
Introduction to Transaction Processing (5)

READ AND WRITE OPERATIONS (cont.):

- **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).

FIGURE 17.2
Two sample transactions. (a) Transaction T_1. (b) Transaction T_2.



Introduction to Transaction Processing (7)

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.

Introduction to Transaction Processing (8)

Why Concurrency Control is needed (cont.):

- **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.

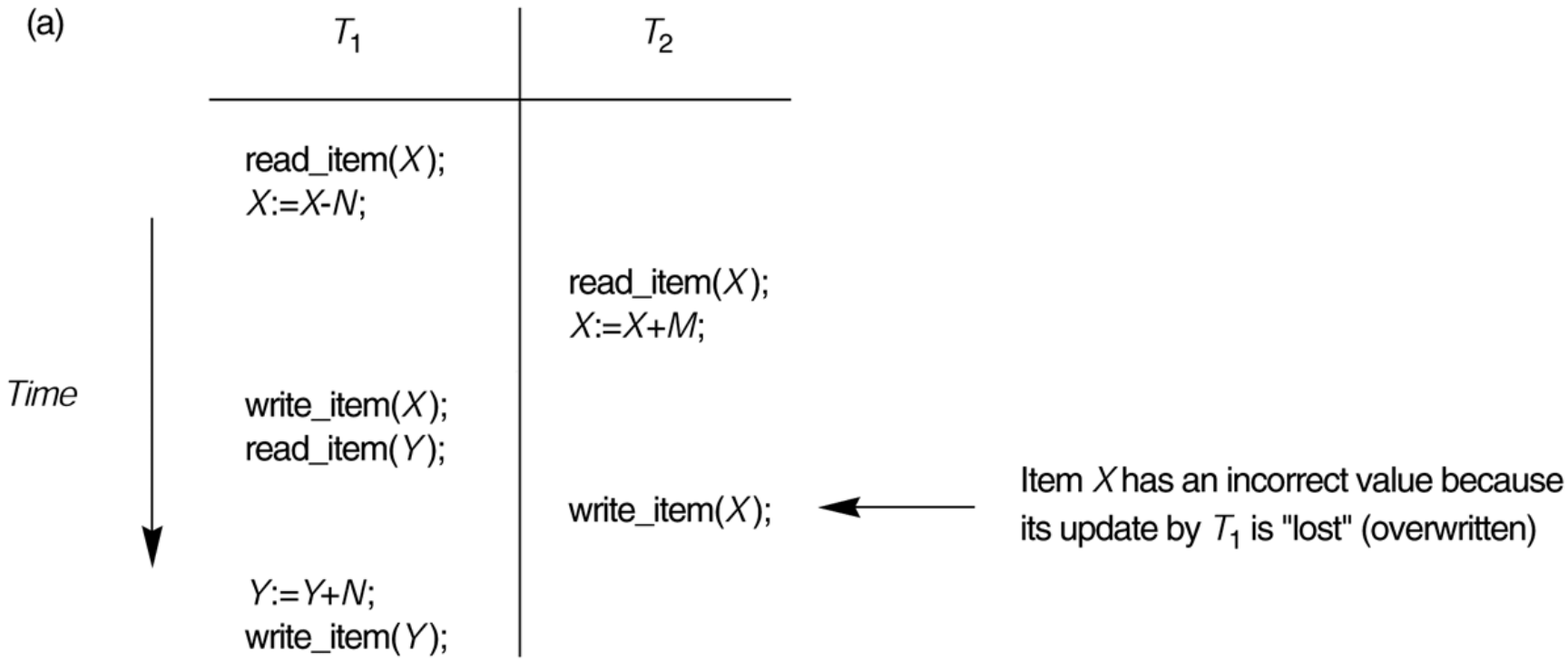


FIGURE 17.3 (continued)
Some problems that occur when concurrent execution is uncontrolled. (b) The temporary update problem.

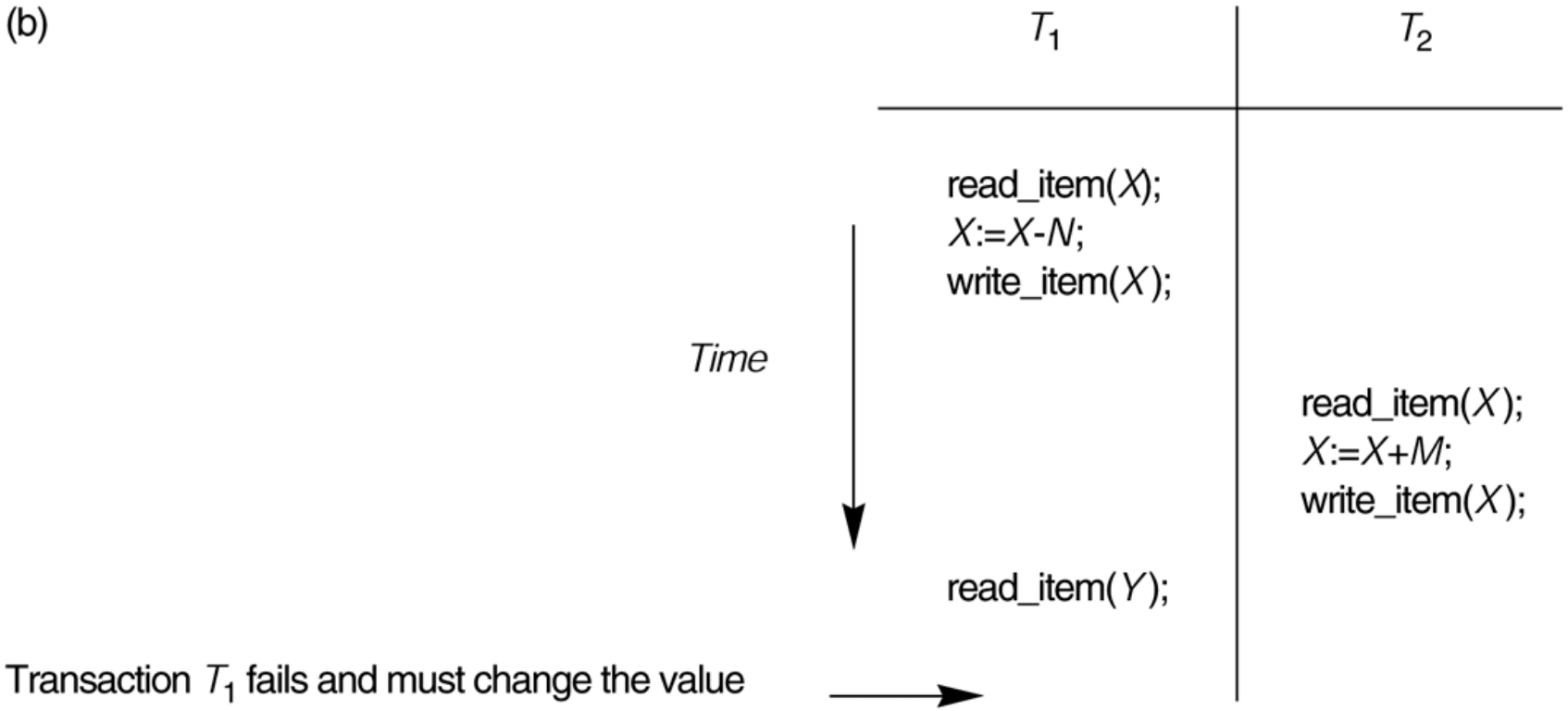
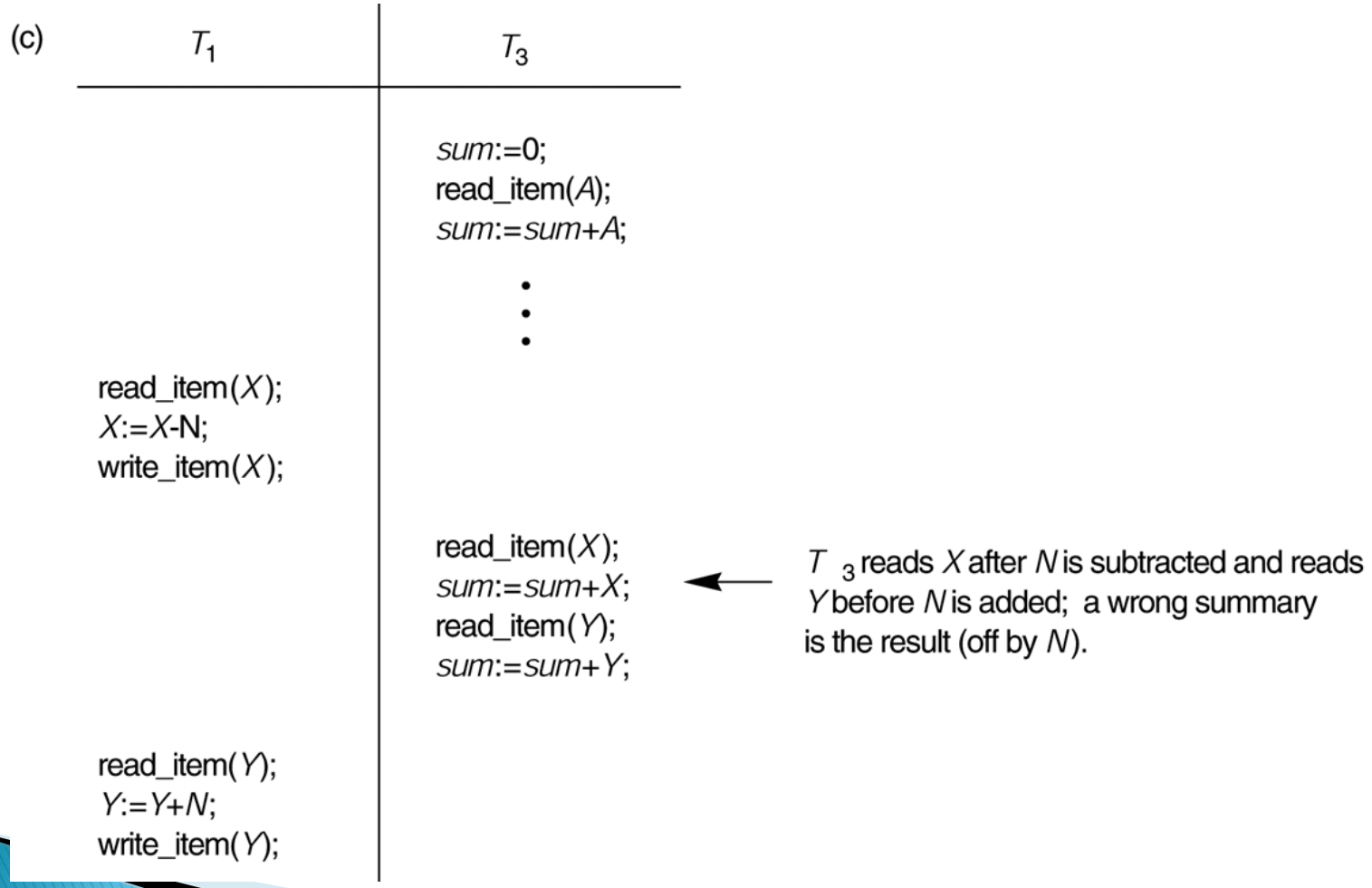


FIGURE 17.3 (continued)
Some problems that occur when concurrent execution is uncontrolled. (c) The incorrect summary problem.



Introduction to Transaction Processing (11)

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.

Introduction to Transaction Processing (12)

Why recovery is needed (cont.):

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).

Introduction to Transaction Processing (13)

Why recovery is needed (cont.):

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.

2. Transaction and System Concepts (1)

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 (2)

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 (3)

Recovery manager keeps track of the following operations (cont):

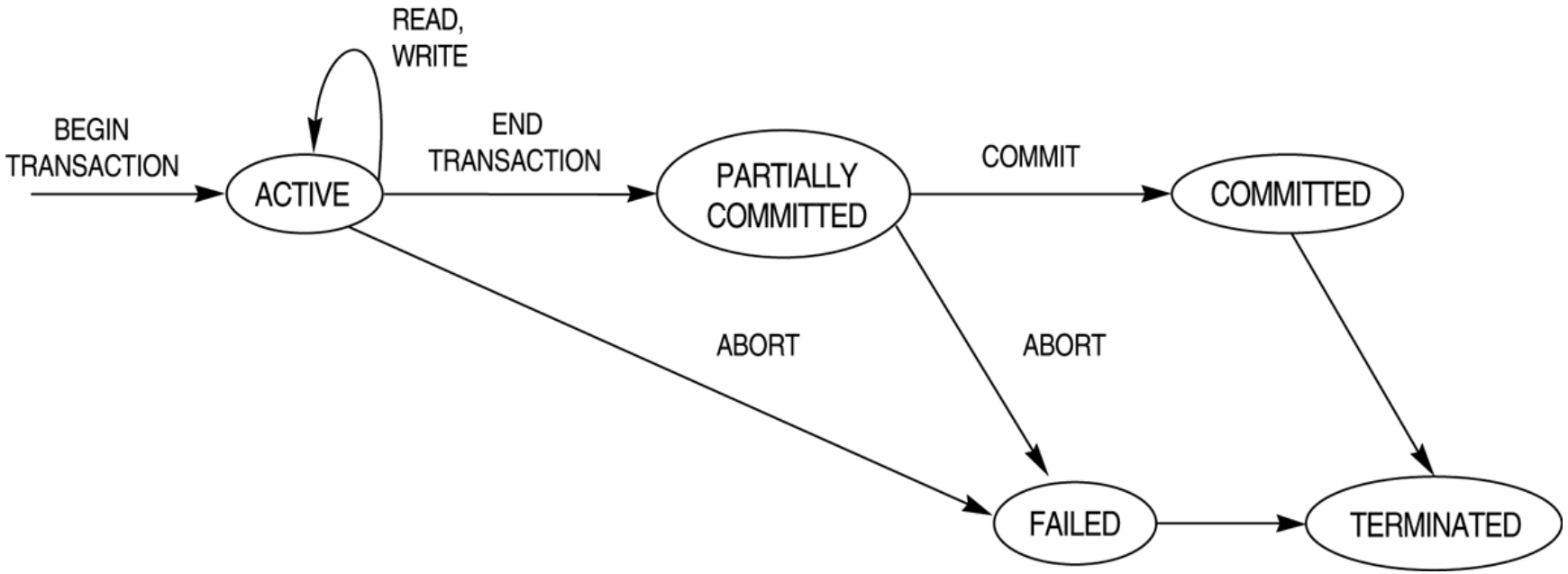
- **commit_transaction:** This signals a *successful end* of the transaction so that any changes (updates) executed by the transaction can be safely **committed** to the database and will not be undone.
- **rollback (or abort):** This signals that the transaction has *ended unsuccessfully,* so that any changes or effects that the transaction may have applied to the database must be *undone.*

Transaction and System Concepts (4)

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.

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



Transaction and System Concepts (6)

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 (7)

The System Log (cont):

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 (8)

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

Transaction and System Concepts (9)

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 (10)

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 (11)

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.

3. Desirable Properties of Transactions (1)

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.

Desirable Properties of Transactions (2)

ACID properties (cont.):

- **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.

4 Characterizing Schedules based on Recoverability (1)

- **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 (2)

Schedules classified on recoverability:

- **Recoverable schedule:** One where no transaction needs to be rolled back.

  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.
- **Cascadeless schedule**: One where every transaction reads only the items that are written by committed transactions.

  Schedules requiring cascaded rollback: A schedule in which uncommitted transactions that read an item from a failed transaction must be rolled back.

Characterizing Schedules based on Recoverability (3)

Schedules classified on recoverability (cont.):

- **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.

5 Characterizing Schedules based on Serializability (1)

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

Characterizing Schedules based on Serializability (2)

- **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'.

Characterizing Schedules based on Serializability (3)

- 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.

Characterizing Schedules based on Serializability (4)

- 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 (5)

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

Characterizing Schedules based on Serializability (6)

- **View equivalence**: A less restrictive definition of equivalence of schedules

- **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 (7)

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 (8)

The premise behind view equivalence:

- As long as each read operation of a transaction reads the result of *the same write operation* in both schedules, the write operations of each transaction musr produce the same results.
- **"The view"**: the read operations are said to see the *the same view* in both schedules.

Characterizing Schedules based on Serializability (9)

Relationship between view and conflict equivalence:

- The two are same under **constrained write assumption** which assumes that if T writes X, it is constrained by the value of X it read; i.e., new X = f(old X)
- Conflict serializability is **stricter** than view serializability. With unconstrained write (or blind write), a schedule that is view serializable is not necessarily conflict serialiable.
- Any conflict serializable schedule is also view serializable, but not vice versa.

Characterizing Schedules based on Serializability (10)

Relationship between view and conflict equivalence (cont):

Consider the following schedule of three transactions
T1: r1(X), w1(X); T2: w2(X); and T3: w3(X):
Schedule Sa: r1(X); w2(X); w1(X); w3(X); c1; c2; c3;

In Sa, the operations w2(X) and w3(X) are blind writes, since T1 and T3 do not read the value of X.

Sa is **view serializable**, since it is view equivalent to the serial schedule T1, T2, T3. However, Sa is **not conflict serializable**, since it is not conflict equivalent to any serial schedule.

Characterizing Schedules based on Serializability (11)

Testing for conflict serializability

Algorithm 17.1:

1. Looks at only read_Item (X) and write_Item (X) operations
2. Constructs a precedence graph (serialization graph) – a graph with directed edges
3. An edge is created from T_i to T_j if one of the operations in T_i appears before a conflicting operation in T_j
4. The schedule is serializable if and only if the precedence graph has no cycles.

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*).

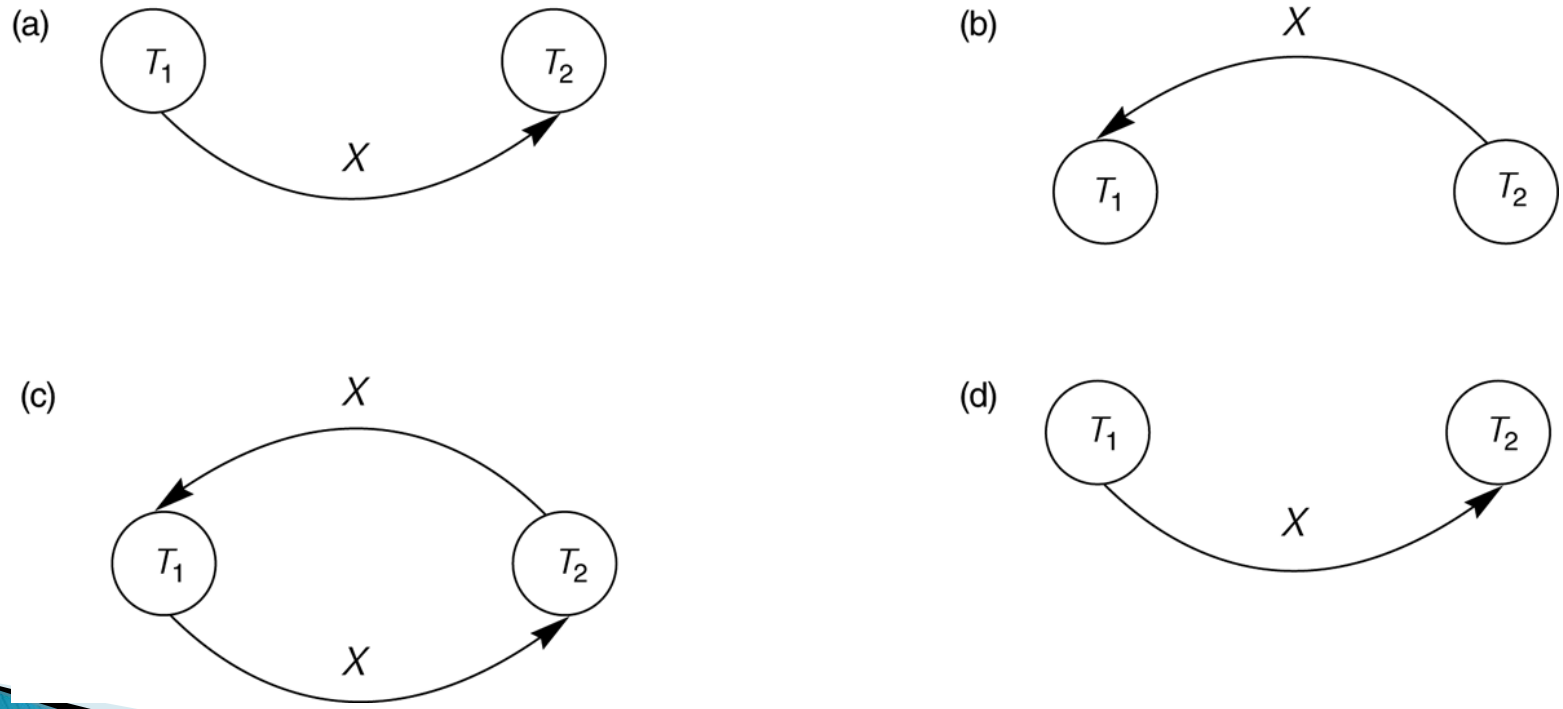


FIGURE 17.8
Another example of serializability testing. (a) The READ and WRITE operations of three transactions T_1, T_2, and T_3.

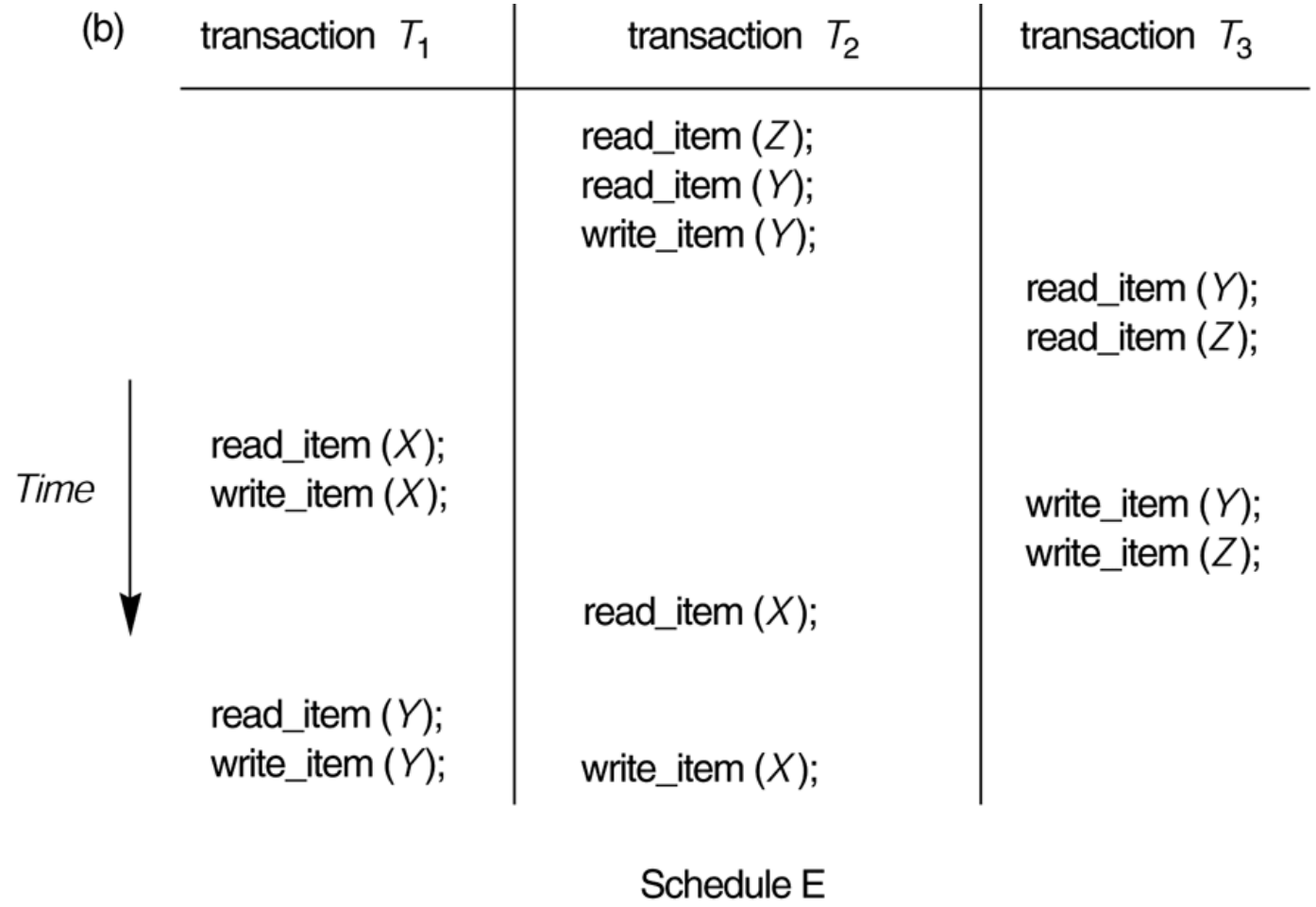
(a)

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);

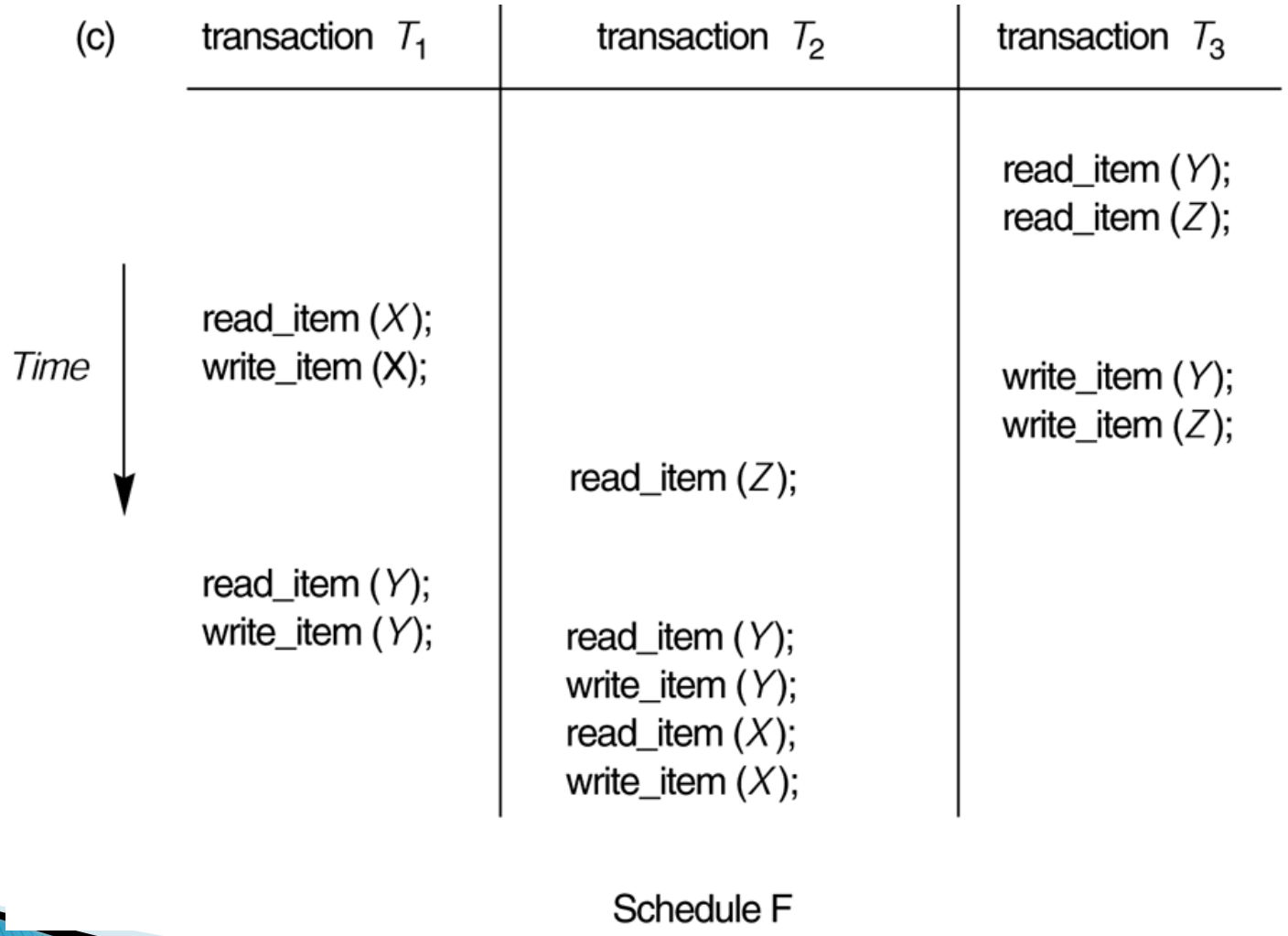
transaction T_3
read_item (Y); read_item (Z); write_item (Y); write_item (Z);

FIGURE 17.8 (continued)
Another example of serializability testing. (b) Schedule *E*.



Schedule E

FIGURE 17.8 (continued)
Another example of serializability testing. (c) Schedule *F*.



Schedule F

Chapter 18 Outline

Databases Concurrency Control

1 Purpose of Concurrency Control

2 Two-Phase locking

3 Timestamp based concurrency control algorithm

4 Concurrency technique for Multiversion

Database Concurrency Control

1 Purpose of Concurrency Control

- To enforce Isolation (through mutual exclusion) among conflicting transactions.
- To preserve database consistency through consistency preserving execution of transactions.
- To resolve read-write and write-write conflicts.

Example: In concurrent execution environment if T1 conflicts with T2 over a data item A, then the existing concurrency control decides if T1 or T2 should get the A and if the other transaction is rolled-back or waits.

Database Concurrency Control

2.Two-Phase Locking Techniques

Locking is an operation which secures (a) permission to Read or (b) permission to Write a data item for a transaction. Example: Lock (X). Data item X is locked in behalf of the requesting transaction.

Unlocking is an operation which removes these permissions from the data item. Example: Unlock (X). Data item X is made available to all other transactions.

Lock and Unlock are Atomic operations.

Database Concurrency Control

Two-Phase Locking Techniques: Essential components

Two locks modes (a) shared (read) and (b) exclusive (write).

Shared mode: shared lock (X). More than one transaction can apply share lock on X for reading its value but no write lock can be applied on X by any other transaction.

Exclusive mode: Write lock (X). Only one write lock on X can exist at any time and no shared lock can be applied by any other transaction on X.

Conflict matrix

	Read	Write
Read	Y	N
Write	N	N

Database Concurrency Control

Two-Phase Locking Techniques: Essential components

Lock Manager: Managing locks on data items.

Lock table: Lock manager uses it to store the identify of transaction locking a data item, the data item, lock mode and pointer to the next data item locked. One simple way to implement a lock table is through linked list.

Transaction ID	Data item id	lock mode	Ptr to next data item
T1	X1	Read	Next

Database Concurrency Control

Two-Phase Locking Techniques: Essential components

Database requires that all transactions should be well-formed. A transaction is well-formed if:

- It must lock the data item before it reads or writes to it.
- It must not lock an already locked data items and it must not try to unlock a free data item.

Database Concurrency Control

Two-Phase Locking Techniques: Essential components

The following code performs the lock operation:

```
B: if LOCK (X) = 0 (*item is unlocked*)
   then LOCK (X) ← 1 (*lock the item*)
   else begin
            wait (until lock (X) = 0) and
            the lock manager wakes up the transaction);
   goto B
   end;
```

Database Concurrency Control

Two-Phase Locking Techniques: Essential components

The following code performs the unlock operation:

LOCK (X) ← 0 (*unlock the item*)
if any transactions are waiting then
    wake up one of the waiting the transactions;

Database Concurrency Control

Two-Phase Locking Techniques: Essential components

The following code performs the read operation:

```
B: if LOCK (X) = "unlocked" then
        begin LOCK (X) ← "read-locked";
            no_of_reads (X) ← 1;
        end
        else if LOCK (X) ← "read-locked" then
                no_of_reads (X) ← no_of_reads (X) +1
             else begin wait (until LOCK (X) = "unlocked" and
                        the lock manager wakes up the transaction);
                        go to B
                     end;
```

Database Concurrency Control

Two-Phase Locking Techniques: Essential components

The following code performs the write lock operation:

```
B: if LOCK (X) = "unlocked" then
        begin LOCK (X) ← "read-locked";
            no_of_reads (X) ← 1;
        end
        else if LOCK (X) ← "read-locked" then
                no_of_reads (X) ← no_of_reads (X) +1
             else begin wait (until LOCK (X) = "unlocked" and
                        the lock manager wakes up the transaction);
                        go to B
                     end;
```

Database Concurrency Control

Two-Phase Locking Techniques: Essential components

The following code performs the unlock operation:

```
if LOCK (X) = "write-locked" then
        begin LOCK (X) ← "unlocked";
              wakes up one of the transactions, if any
        end
        else if LOCK (X) ← "read-locked" then
             begin
                  no_of_reads (X) ← no_of_reads (X) -1
                  if  no_of_reads (X) = 0 then
                  begin
                         LOCK (X) = "unlocked";
                        wake up one of the transactions, if any
                  end
             end;
```

Database Concurrency Control

Two-Phase Locking Techniques: Essential components

Lock conversion

Lock upgrade: existing read lock to write lock

```
if Ti has a read-lock (X) and Tj has no read-lock (X) (i ≠ j) then
    convert read-lock (X) to write-lock (X)
else
    force Ti to wait until Tj unlocks X
```

Lock downgrade: existing write lock to read lock

```
Ti has a write-lock (X)    (*no transaction can have any lock on X*)
convert write-lock (X) to read-lock (X)
```

Database Concurrency Control

Two-Phase Locking Techniques: The algorithm

Two Phases: (a) Locking (Growing) (b) Unlocking (Shrinking).

Locking (Growing) Phase: A transaction applies locks (read or write) on desired data items one at a time.

Unlocking (Shrinking) Phase: A transaction unlocks its locked data items one at a time.

Requirement: For a transaction these two phases must be mutually exclusively, that is, during locking phase unlocking phase must not start and during unlocking phase locking phase must not begin.

Database Concurrency Control

Two-Phase Locking Techniques: The algorithm

T1	T2	Result
read_lock (Y);	read_lock (X);	Initial values: X=20; Y=30
read_item (Y);	read_item (X);	Result of serial execution
unlock (Y);	unlock (X);	T1 followed by T2
write_lock (X);	Write_lock (Y);	X=50, Y=80.
read_item (X);	read_item (Y);	Result of serial execution
X:=X+Y;	Y:=X+Y;	T2 followed by T1
write_item (X);	write_item (Y);	X=70, Y=50
unlock (X);	unlock (Y);	

Database Concurrency Control

Two-Phase Locking Techniques: The algorithm

T1	T2	Result
read_lock (Y);		X=50; Y=50
read_item (Y);		Nonserializable because it.
unlock (Y);		violated two-phase policy.
	read_lock (X);	
	read_item (X);	
	unlock (X);	
	write_lock (Y);	
	read_item (Y);	
	Y:=X+Y;	
	write_item (Y);	
	unlock (Y);	
write_lock (X);		
read_item (X);		
X:=X+Y;		
write_item (X);		
unlock (X);		

Time

Database Concurrency Control

Two-Phase Locking Techniques: The algorithm

T'1	T'2	
read_lock (Y);	read_lock (X);	T1 and T2 follow two-phase policy but they are subject to deadlock, which must be dealt with.
read_item (Y);	read_item (X);	
write_lock (X);	Write_lock (Y);	
unlock (Y);	unlock (X);	
read_item (X);	read_item (Y);	
X:=X+Y;	Y:=X+Y;	
write_item (X);	write_item (Y);	
unlock (X);	unlock (Y);	

Database Concurrency Control

Two-Phase Locking Techniques: The algorithm

Two-phase policy generates two locking algorithms (a) Basic and (b) Conservative.

Conservative: Prevents deadlock by locking all desired data items before transaction begins execution.

Basic: Transaction locks data items incrementally. This may cause deadlock which is dealt with.

Strict: A more stricter version of Basic algorithm where unlocking is performed after a transaction terminates (commits or aborts and rolled-back). This is the most commonly used two-phase locking algorithm.

Database Concurrency Control

3. Timestamp based concurrency control algorithm

Timestamp

A monotonically increasing variable (integer) indicating the age of an operation or a transaction. A larger timestamp value indicates a more recent event or operation.

Timestamp based algorithm uses timestamp to serialize the execution of concurrent transactions.

Database Concurrency Control

Timestamp based concurrency control algorithm

Basic Timestamp Ordering

1. Transaction T issues a write_item(X) operation:
   a. If read_TS(X) > TS(T) or if write_TS(X) > TS(T), then an younger transaction has already read the data item so abort and roll-back T and reject the operation.
   b. If the condition in part (a) does not exist, then execute write_item(X) of T and set write_TS(X) to TS(T).
2. Transaction T issues a read_item(X) operation:
   a. If write_TS(X) > TS(T), then an younger transaction has already written to the data item so abort and roll-back T and reject the operation.
   b. If write_TS(X) ≤ TS(T), then execute read_item(X) of T and set read_TS(X) to the larger of TS(T) and the current read_TS(X).

Database Concurrency Control

Timestamp based concurrency control algorithm

Strict Timestamp Ordering

1. Transaction T issues a write_item(X) operation:
   a. If TS(T) > read_TS(X), then delay T until the transaction T' that wrote or read X has terminated (committed or aborted).
2. Transaction T issues a read_item(X) operation:
   a. If TS(T) > write_TS(X), then delay T until the transaction T' that wrote or read X has terminated (committed or aborted).

Database Concurrency Control

Timestamp based concurrency control algorithm

Thomas's Write Rule

1. If read_TS(X) > TS(T) then abort and roll-back T and reject the operation.
2. If write_TS(X) > TS(T), then just ignore the write operation and continue execution. This is because the most recent writes counts in case of two consecutive writes.
3. If the conditions given in 1 and 2 above do not occur, then execute write_item(X) of T and set write_TS(X) to TS(T).

Database Concurrency Control

4. Multiversion concurrency control techniques

This approach maintains a number of versions of a data item and allocates the right version to a read operation of a transaction. Thus unlike other mechanisms a read operation in this mechanism is never rejected.

Side effect: Significantly more storage (RAM and disk) is required to maintain multiple versions. To check unlimited growth of versions, a garbage collection is run when some criteria is satisfied.

Database Concurrency Control

Multiversion technique based on timestamp ordering

This approach maintains a number of versions of a data item and allocates the right version to a read operation of a transaction. Thus unlike other mechanisms a read operation in this mechanism is never rejected.

Side effects: Significantly more storage (RAM and disk) is required to maintain multiple versions. To check unlimited growth of versions, a garbage collection is run when some criteria is satisfied.

Database Concurrency Control

Multiversion technique based on timestamp ordering

Assume X1, X2, …, Xn are the version of a data item X created by a write operation of transactions. With each Xi a read_TS (read timestamp) and a write_TS (write timestamp) are associated.

read_TS(Xi): The read timestamp of Xi is the largest of all the timestamps of transactions that have successfully read version Xi.

write_TS(Xi): The write timestamp of Xi that wrote the value of version Xi.

A new version of Xi is created only by a write operation.

Database Concurrency Control

Multiversion technique based on timestamp ordering

To ensure serializability, the following two rules are used.

If transaction T issues write_item (X) and version i of X has the highest write_TS(Xi) of all versions of X that is also less than or equal to TS(T), and read _TS(Xi) > TS(T), then abort and roll-back T; otherwise create a new version Xi and read_TS(X) = write_TS(Xj) = TS(T).

If transaction T issues read_item (X), find the version i of X that has the highest write_TS(Xi) of all versions of X that is also less than or equal to TS(T), then return the value of Xi to T, and set the value of read _TS(Xi) to the largest of TS(T) and the current read_TS(Xi).

Database Concurrency Control

Multiversion technique based on timestamp ordering

To ensure serializability, the following two rules are used.

1. If transaction T issues write_item (X) and version i of X has the highest write_TS(Xi) of all versions of X that is also less than or equal to TS(T), and read _TS(Xi) > TS(T), then abort and roll-back T; otherwise create a new version Xi and read_TS(X) = write_TS(Xj) = TS(T).

2. If transaction T issues read_item (X), find the version i of X that has the highest write_TS(Xi) of all versions of X that is also less than or equal to TS(T), then return the value of Xi to T, and set the value of read _TS(Xi) to the largest of TS(T) and the current read_TS(Xi).

Rule 2 guarantees that a read will never be rejected.

Database Concurrency Control

Multiversion Two-Phase Locking Using Certify Locks

Concept

Allow a transaction T' to read a data item X while it is write locked by a conflicting transaction T.

This is accomplished by maintaining two versions of each data item X where one version must always have been written by some committed transaction. This means a write operation always creates a new version of X.

Database Concurrency Control

Multiversion Two-Phase Locking Using Certify Locks

Steps

1. X is the committed version of a data item.
2. T creates a second version X' after obtaining a write lock on X.
3. Other transactions continue to read X.
4. T is ready to commit so it obtains a certify lock on X'.
5. The committed version X becomes X'.
6. T releases its certify lock on X', which is X now.

Compatibility tables for

	Read	Write
Read	yes	no
Write	no	no

read/write locking scheme

	Read	Write	Certify
Read	yes	no	no
Write	no	no	no
Certify	no	no	no

read/write/certify locking scheme

Database Concurrency Control

Multiversion Two-Phase Locking Using Certify Locks

Note

In multiversion 2PL read and write operations from conflicting transactions can be processed concurrently. This improves concurrency but it may delay transaction commit because of obtaining certify locks on all its writes. It avoids cascading abort but like strict two phase locking scheme conflicting transactions may get deadlocked.

Chapter 19 Outline

Databases Recovery

1 Purpose of Database Recovery
2 Types of Failure
3 Transaction Log
4 Data Updates
5 Data Caching
6 Transaction Roll-back (Undo) and Roll-Forward
7 Checkpointing
8 Recovery schemes – Deferred and Immediate update

Database Recovery

1 Purpose of Database Recovery

- To bring the database into the last consistent state, which existed prior to the failure.
- To preserve transaction properties (Atomicity, Consistency, Isolation and Durability).

Example: If the system crashes before a fund transfer transaction completes its execution, then either one or both accounts may have incorrect value. Thus, the database must be restored to the state before the transaction modified any of the accounts.

Database Recovery

2 Types of Failure

The database may become unavailable for use due to

- Transaction failure: Transactions may fail because of incorrect input, deadlock, incorrect synchronization.
- System failure: System may fail because of addressing error, application error, operating system fault, RAM failure, etc.
- Media failure: Disk head crash, power disruption, etc.

Database Recovery

3 Transaction Log

For recovery from any type of failure data values prior to modification (BFIM – BeFore Image) and the new value after modification (AFIM – AFter Image) are required. These values and other information is stored in a sequential file called Transaction log. A sample log is given below. **Back P** and **Next P** point to the previous and next log records of the same transaction.

T ID	Back P	Next P	Operation	Data item	BFIM	AFIM
T1	0	1	Begin			
T1	1	4	Write	X	X = 100	X = 200
T2	0	8	Begin			
T1	2	5	W	Y	Y = 50	Y = 100
T1	4	7	R	M	M = 200	M = 200
T3	0	9	R	N	N = 400	N = 400
T1	5	nil	End			

Database Recovery

4 Data Update

- **Immediate Update**: As soon as a data item is modified in cache, the disk copy is updated.
- **Deferred Update**: All modified data items in the cache is written either after a transaction ends its execution or after a fixed number of transactions have completed their execution.
- **Shadow update**: The modified version of a data item does not overwrite its disk copy but is written at a separate disk location.
- **In-place update**: The disk version of the data item is overwritten by the cache version.

Database Recovery

5 Data Caching

Data items to be modified are first stored into database cache by the Cache Manager (CM) and after modification they are flushed (written) to the disk. The flushing is controlled by **Modified** and **Pin-Unpin** bits.

Pin-Unpin: Instructs the operating system not to flush the data item.

Modified: Indicates the AFIM of the data item.

Database Recovery

6 **Transaction Roll-back (Undo) and Roll-Forward (Redo)**

To maintain atomicity, a transaction's operations are **redone** or **undone.**

Undo: Restore all BFIMs on to disk (Remove all AFIMs).

Redo: Restore all AFIMs on to disk.

Database recovery is achieved either by performing only Undos or only Redos or by a combination of the two. These operations are recorded in the log as they happen.

Database Recovery

Roll-back

We show the process of roll-back with the help of the following three transactions T1, and T2 and T3.

T1	T2	T3
read_item (A)	read_item (B)	read_item (C)
read_item (D)	write_item (B)	write_item (B)
write_item (D)	read_item (D)	read_item (A)
	write_item (A)	write_item (A)

Database Recovery

Roll-back: One execution of T1, T2 and T3 as recorded in the log.

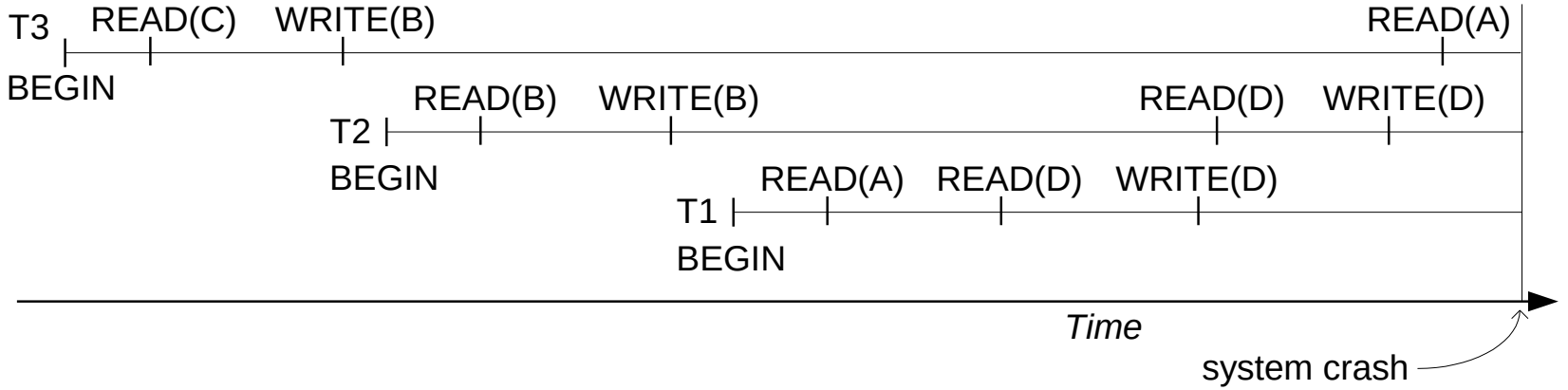
		A	B	C	D
		30	15	40	20
	[start_transaction, T3]				
	[read_item, T3, C]				
*	[write_item, T3, B, 15, 12]		12		
	[start_transaction,T2]				
	[read_item, T2, B]				
**	[write_item, T2, B, 12, 18]		18		
	[start_transaction,T1]				
	[read_item, T1, A]				
	[read_item, T1, D]				
	[write_item, T1, D, 20, 25]				25
	[read_item, T2, D]				
**	[write_item, T2, D, 25, 26]				26
	[read_item, T3, A]				
	---- system crash ----				

* T3 is rolled back because it did not reach its commit point.

** T2 is rolled back because it reads the value of item B written by T3.

Database Recovery

Roll-back: One execution of T1, T2 and T3 as recorded in the log.



Illustrating cascading roll-back

Database Recovery

Write-Ahead Logging

When **in-place** update (immediate or deferred) is used then log is necessary for recovery and it must be available to recovery manager. This is achieved by **Write-Ahead Logging** (WAL) protocol. WAL states that

For Undo: Before a data item's AFIM is flushed to the database disk (overwriting the BFIM) its BFIM must be written to the log and the log must be saved on a stable store (log disk).

For Redo: Before a transaction executes its commit operation, all its AFIMs must be written to the log and the log must be saved on a stable store.

Database Recovery

7 Checkpointing

Time to time (randomly or under some criteria) the database flushes its buffer to database disk to minimize the task of recovery. The following steps defines a checkpoint operation:

1. Suspend execution of transactions temporarily.
2. Force write modified buffer data to disk.
3. Write a [checkpoint] record to the log, save the log to disk.
4. Resume normal transaction execution.

During recovery **redo** or **undo** is required to transactions appearing after [checkpoint] record.

Database Recovery

Steal/No-Steal and Force/No-Force

Possible ways for flushing database cache to database disk:

Steal: Cache can be flushed before transaction commits.

No-Steal: Cache cannot be flushed before transaction commit.

Force: Cache is immediately flushed (forced) to disk.

No-Force: Cache is deferred until transaction commits.

These give rise to four different ways for handling recovery:

Steal/No-Force (Undo/Redo), Steal/Force (Undo/No-redo), No-Steal/No-Force (Redo/No-undo) and No-Steal/Force (No-undo/No-redo).

Database Recovery

8 Recovery Scheme

Deferred Update (No Undo/Redo)

The data update goes as follows:

1. A set of transactions records their updates in the log.
2. At commit point under WAL scheme these updates are saved on database disk.

After reboot from a failure the log is used to redo all the transactions affected by this failure. No undo is required because no AFIM is flushed to the disk before a transaction commits.

Database Recovery

Deferred Update in a single-user system

There is no concurrent data sharing in a single user system. The data update goes as follows:

1. A set of transactions records their updates in the log.
2. At commit point under WAL scheme these updates are saved on database disk.

After reboot from a failure the log is used to redo all the transactions affected by this failure. No undo is required because no AFIM is flushed to the disk before a transaction commits.

Database Recovery

Deferred Update in a single-user system

(a)

T1	T2
read_item (A)	read_item (B)
read_item (D)	write_item (B)
write_item (D)	read_item (D)
	write_item (A)

(b)

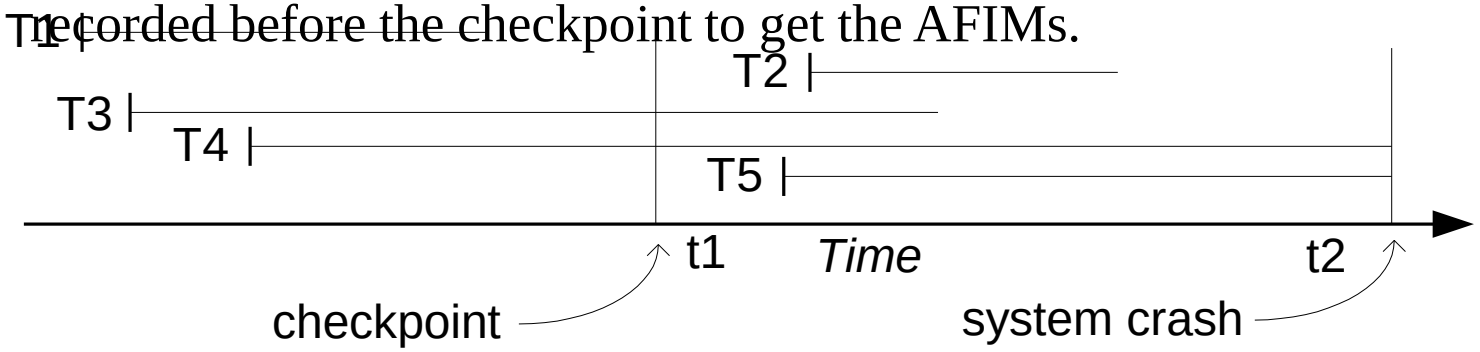
[start_transaction, T1]
[write_item, T1, D, 20]
[commit T1]
[start_transaction, T1]
[write_item, T2, B, 10]
[write_item, T2, D, 25] ← system crash

The [write_item, …] operations of T1 are redone.
T2 log entries are ignored by the recovery manager.

Database Recovery

Deferred Update with concurrent users

This environment requires some concurrency control mechanism to guarantee isolation property of transactions. In a system recovery transactions which were recorded in the log after the last checkpoint were redone. The recovery manager may scan some of the transactions recorded before the checkpoint to get the AFIMs.



Recovery in a concurrent users environment.

Database Recovery

Deferred Update with concurrent users

(a)

T1	T2	T3	T4
read_item (A)	read_item (B)	read_item (A)	read_item (B)
read_item (D)	write_item (B)	write_item (A)	write_item (B)
write_item (D)	read_item (D)	read_item (C)	read_item (A)
	write_item (D)	write_item (C)	write_item (A)

(b)
[start_transaction, T1]
[write_item, T1, D, 20]
[commit, T1]
[checkpoint]
[start_transaction, T4]
[write_item, T4, B, 15]
[write_item, T4, A, 20]
[commit, T4]
[start_transaction T2]
[write_item, T2, B, 12]
[start_transaction, T3]
[write_item, T3, A, 30]
[write_item, T2, D, 25] ← system crash

T2 and T3 are ignored because they did not reach their commit points.
T4 is redone because its commit point is after the last checkpoint.

Database Recovery

Deferred Update with concurrent users

Two tables are required for implementing this protocol:

Active table: All active transactions are entered in this table.
Commit table: Transactions to be committed are entered in this table.

During recovery, all transactions of the commit table are redone and all transactions of active tables are ignored since none of their AFIMs reached the database. It is possible that a commit table transaction may be redone twice but this does not create any inconsistency because of a redone is “idempotent”, that is, one redone for an AFIM is equivalent to multiple redone for the same AFIM.

Database Recovery

Recovery Techniques Based on Immediate Update

Undo/No-redo Algorithm

In this algorithm AFIMs of a transaction are flushed to the database disk under WAL before it commits. For this reason the recovery manager undoes all transactions during recovery. No transaction is redone. It is possible that a transaction might have completed execution and ready to commit but this transaction is also undone.

Database Recovery

Recovery Techniques Based on Immediate Update

Undo/Redo Algorithm (Single-user environment)

Recovery schemes of this category apply undo and also redo for recovery. In a single-user environment no concurrency control is required but a log is maintained under WAL. Note that at any time there will be one transaction in the system and it will be either in the commit table or in the active table. The recovery manager performs:

1. Undo of a transaction if it is in the active table.
2. Redo of a transaction if it is in the commit table.

Database Recovery

Recovery Techniques Based on Immediate Update

Undo/Redo Algorithm (Concurrent execution)

Recovery schemes of this category applies undo and also redo to recover the database from failure. In concurrent execution environment a concurrency control is required and log is maintained under WAL. Commit table records transactions to be committed and active table records active transactions. To minimize the work of the recovery manager checkpointing is used. The recovery performs:

1. Undo of a transaction if it is in the active table.
2. Redo of a transaction if it is in the commit table.

Refrences

- Ramez Elmasri and B. Navathe, "Fundamentals of Database Systems", 2016, 7th Edition, Pearson Education