Transaction Processing

Aims:

At the end of this group of two lectures you should be able to understand the concept of transactions, the purpose of concurrency control and recovery.

Reading: Elmasri & Navathe, 6th ed., Chapters 21&22

7th ed., Chapters 20-22

Overview

- Introduction to transactions
- Concurrent execution of transactions
  - Lost update problem
  - Temporary update problem
  - Incorrect summary problem
- ACID properties of transactions
- Concurrency control and recovery
- Transaction support in SQL

Multiuser systems

- **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.

Interleaved vs. Parallel Processing

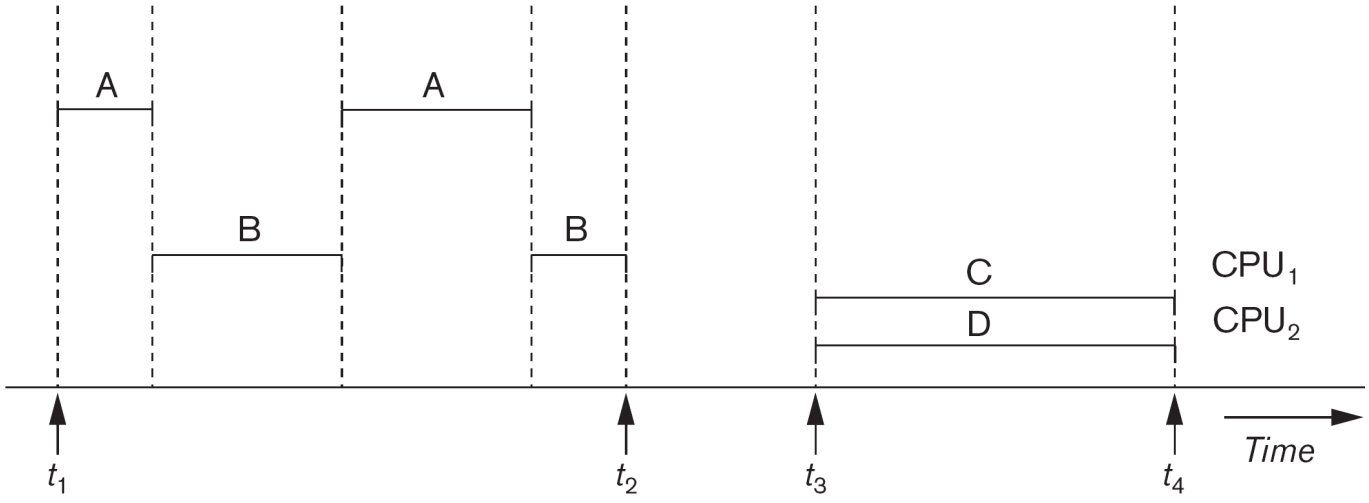


Figure 21.1
Interleaved processing versus parallel processing of concurrent transactions.

TRANSACTIONS

- A transaction is a logical unit of work
- Could be one or more statements
- An **application program** may contain several transactions separated by the *Begin* and *End* transaction boundaries
- The effects of transactions only become permanent when the transaction is completed successfully
- Functions of multi-user DBMSs
  - Transaction support
  - Concurrency control
  - Recovery

Transaction Processing

- **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

- **read_item(X)**
  - 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 the main memory)
  - Copy item X from the buffer to the program variable named X.
- **write_item(X)**
  - 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 the main memory)
  - 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

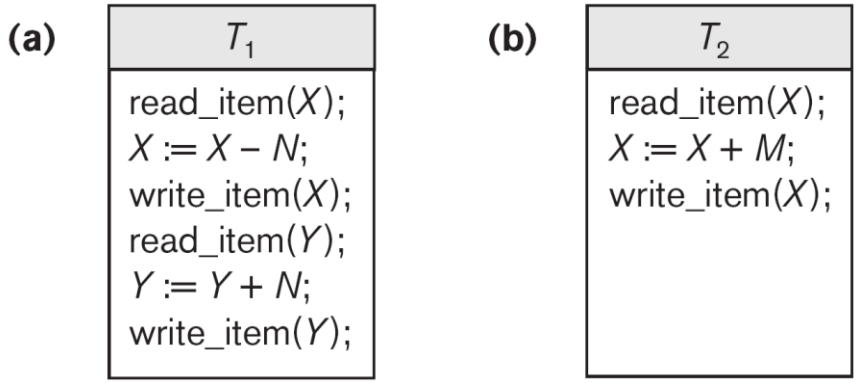


Figure 21.2
Two sample transactions. (a) Transaction T_1. (b) Transaction T_2.

Why is Concurrency Control 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.

THE LOST UPDATE PROBLEM.

(a)

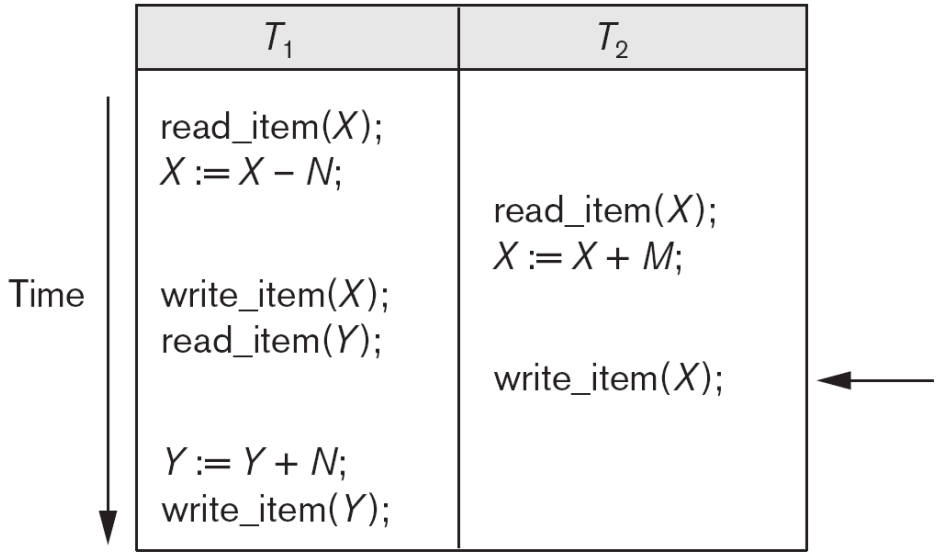
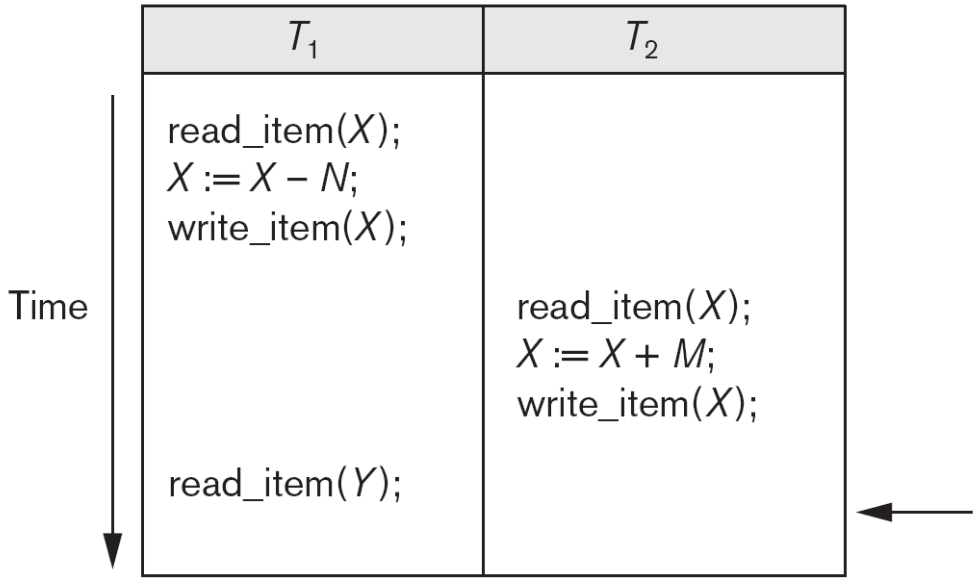


Figure 21.3
Some problems that occur when concurrent execution is uncontrolled. (a) The lost update problem. (b) The temporary update problem. (c) The incorrect summary problem.

The temporary update problem.

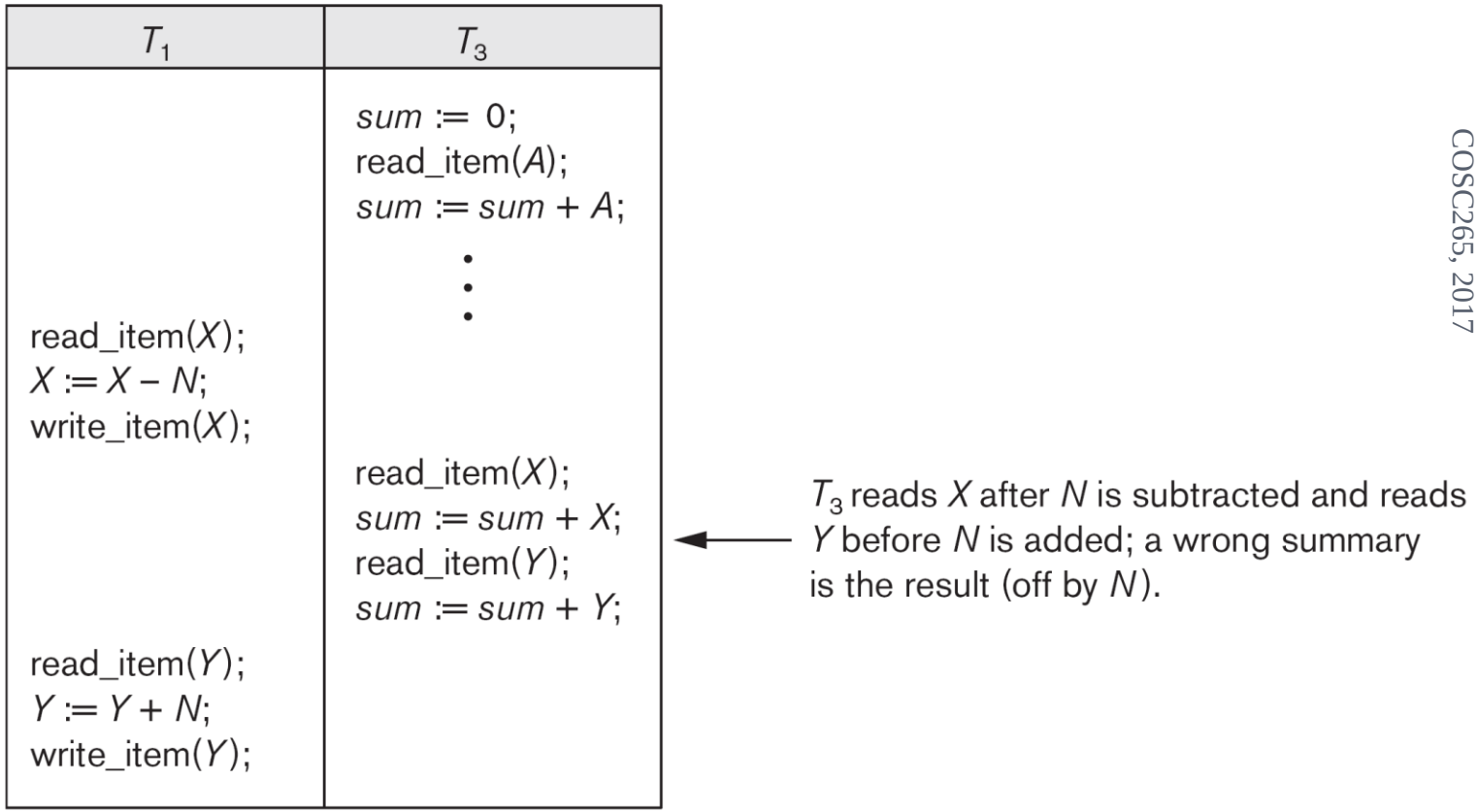
(b)



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

The incorrect summary problem

(c)



What causes transactions 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.

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.

What causes transactions 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

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:

Power or air-conditioning failure, fire, theft, sabotage, overwriting disks or tapes by mistake, and mounting of a wrong tape by the operator.

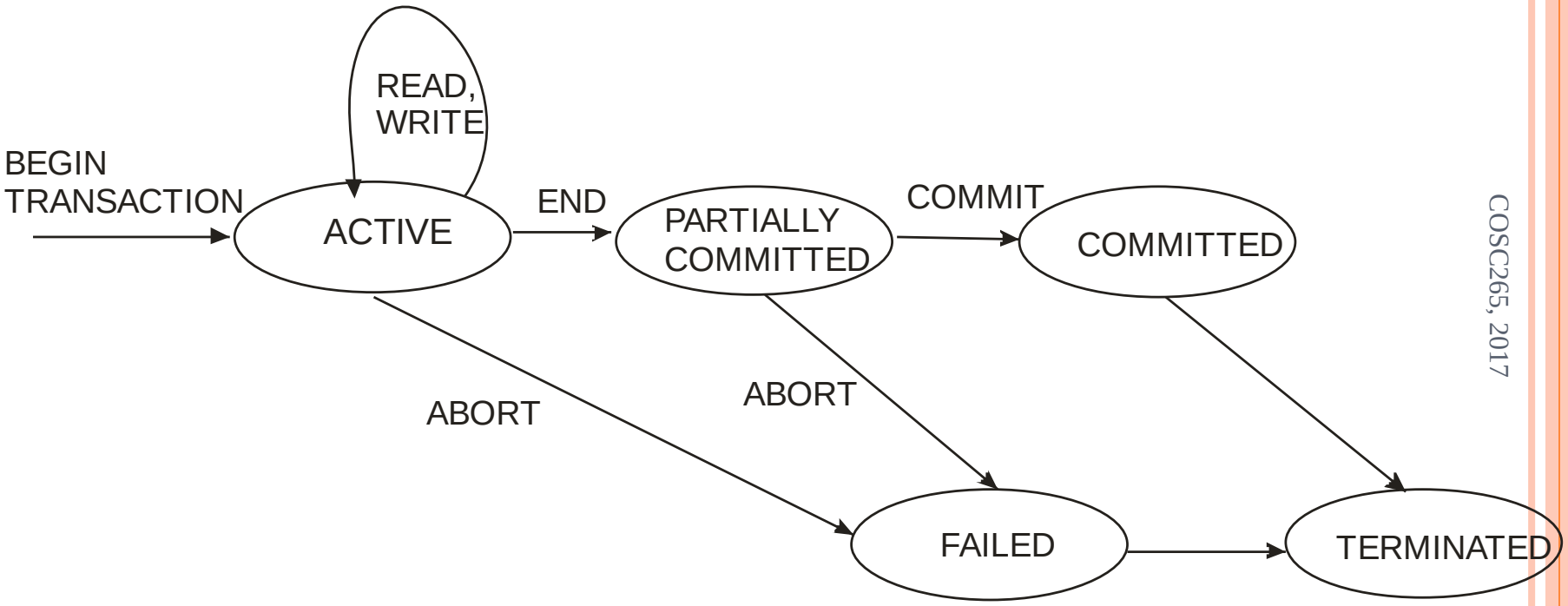
Transaction States

- 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**
  - **read** or **write**
  - **end_transaction**
  - **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 STATES



Transaction and System Concepts

- Recovery techniques use the following operators:
  - **undo**: Similar to rollback except that it applies to a single operation rather than to a whole transaction.
  - **redo**: This specifies that certain *transaction operations* must be *redone* to ensure that all the operations of a committed transaction have been applied successfully to the database.

System 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.
- The last part of the log file is in the main memory buffers
- In addition, the log is periodically backed up to archival storage to guard against such catastrophic failures.

System Log

- T is a unique **transaction-id** that is generated automatically by the system 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.

Recovery Using System Log

If the system crashes, we can recover to a consistent database state by examining the log

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

- **A**ll 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

- **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.

Checkpoint record

- Written to the log when the system writes out to the database on disk the effects of all write operations of committed transactions.
- This is done periodically (every *m* minutes or after *t* committed transactions)
- The steps:
  - Suspend execution of all transactions
  - Force-write all main memory buffers that have been modified to disk
  - Write a [checkpoint] record to the log, and force-write the log to disk
  - Resume executing transactions

Desirable Properties of Transactions

ACID properties:

- **Atomicity (all or nothing property)**: 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 (independence)**: 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
- **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

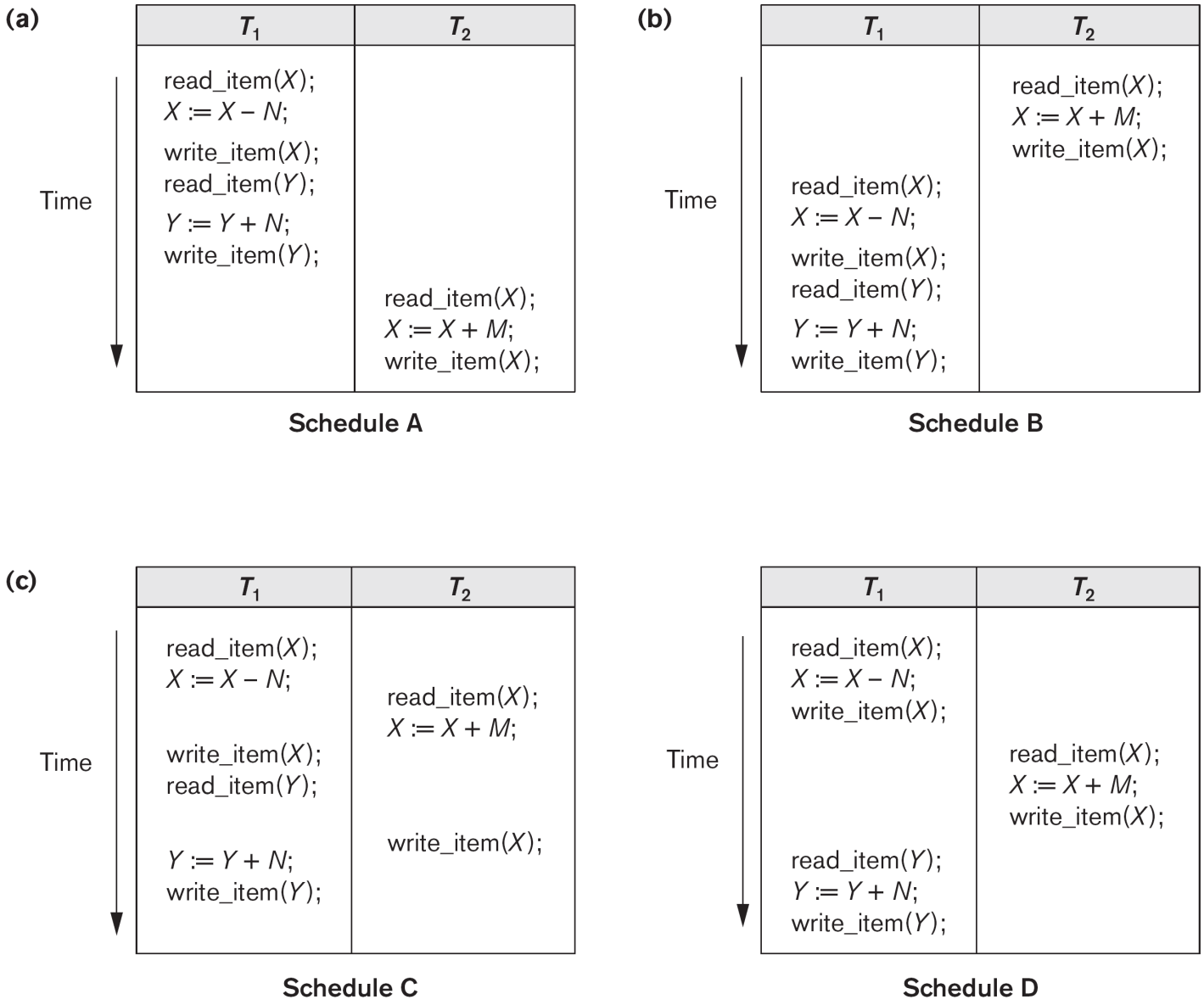
- **Transaction schedule or history**: the order of execution of operations from the various transactions
- **Serial schedule**: all operations of each transaction are executed consecutively in the schedule. Every serial schedule is correct.
- **Non-serial schedule:** an ordering of the operations of the transactions such that the operations of T1 in S must appear in the same order in which they occur in T1. Operations from other transactions Tj can be interleaved with the operations of Ti in S.
- **Serializable schedule**: equivalent to some serial schedule of the same *n* transactions.

Conflicting operations

- Belong to different transactions
- Access the same data item X
- At least one of them is write_item(X)

- Shorthand notation
  - use symbles r, w, c and a for operations read_item, write_item, commit and abort
  - Append as subscript the transaction id

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.



Equivalent schedules

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

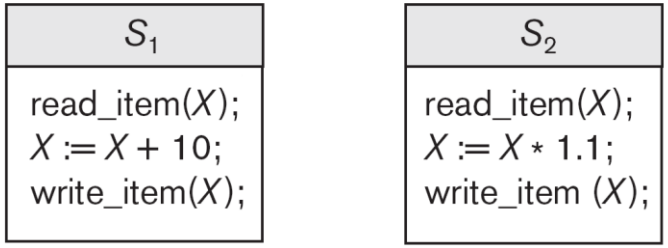


Figure 21.6
Two schedules that are result equivalent for the initial value of $X = 100$ but are not result equivalent in general.

Equivalent schedules

- **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'.
- 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

- Serializability is hard to check
  - Interleaving of operations done by OS scheduler
  - Difficult to determine beforehand how the operations in a schedule will be interleaved.
- 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

Testing for conflict serializability

Algorithm 21.1:

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

Constructing the Precedence Graphs

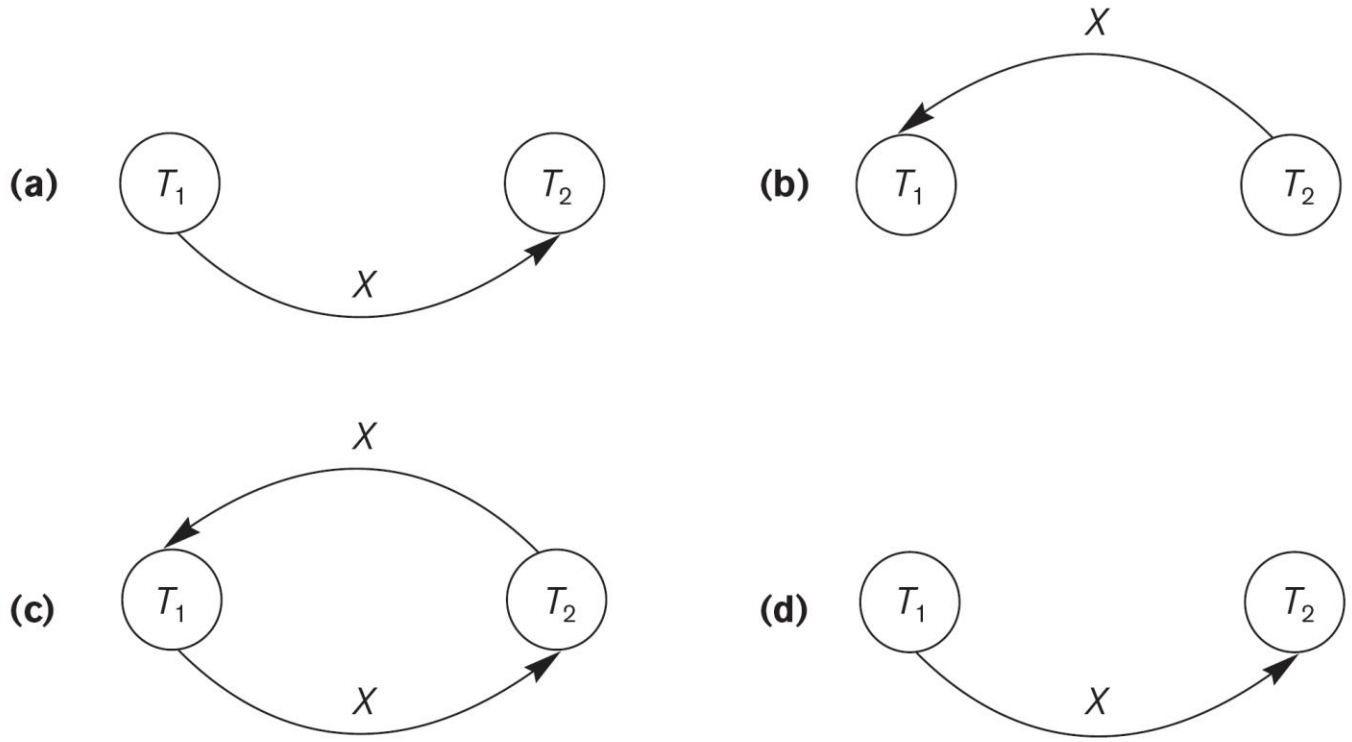


Figure 21.7
Constructing the precedence graphs for schedules A to D from Figure 21.5 to test for conflict serializability. (a) Precedence graph for serial schedule A. (b) Precedence graph for serial schedule B. (c) Precedence graph for schedule C (not serializable). (d) Precedence graph for schedule D (serializable, equivalent to schedule A).

Another Example of Serializability Testing

(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 21.8
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.

Another Example of Serializability Testing

(b)

Time

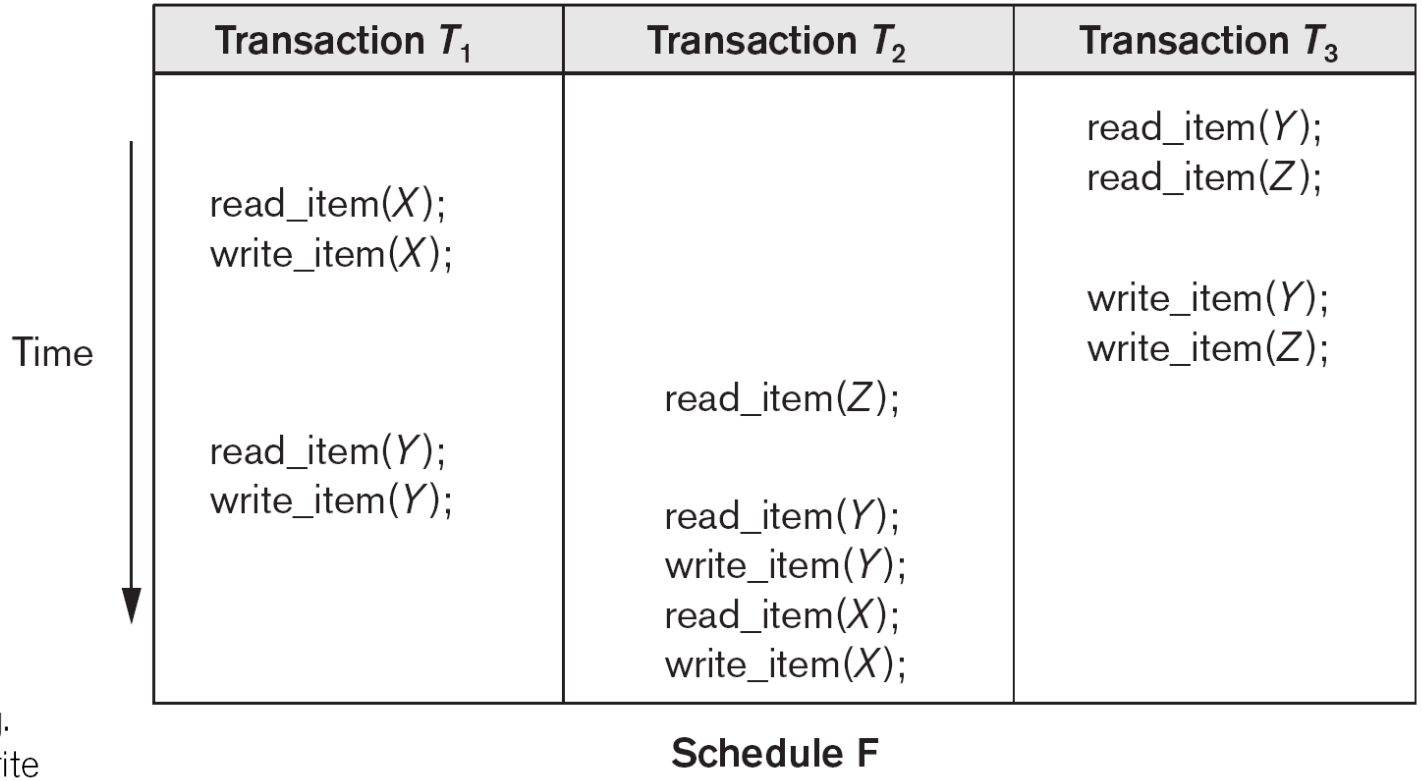
Transaction T_1	Transaction T_2	Transaction T_3
	read_item(Z);	
	read_item(Y);	
	write_item(Y);	
		read_item(Y);
		read_item(Z);
read_item(X);		
write_item(X);		
		write_item(Y);
		write_item(Z);
	read_item(X);	
read_item(Y);		
write_item(Y);		
	write_item(X);	

Schedule E

Figure 21.8
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.

Another Example of Serializability Testing

(c)



Schedule F

Figure 21.8
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.

Transactions in SQL

- The beginning of a transaction is the first SQL statement following a CONNECT, COMMIT, or ROLLBACK statement, issued by a user, program, or by the DBMS.
- Every DDL statement is a separate transaction
- The end of a transaction
  - DDL statement
  - COMMIT
  - ROLLBACK
  - Termination of the session (implicit COMMIT)
  - Abnormal termination (by a program or DBMS)

Transactions in SQL (2)

- SET TRANSACTION {{ READ {ONLY |WRITE} | ISOLATION LEVEL {SERIALIZABLE | READ COMMITTED} | USE ROLLBACK SEGMENT rollback_segment } [NAME 'text'] | NAME 'text' };
- COMMIT [WORK] [COMMENT 'text' | FORCE 'text' [, integer]] ;
- ROLLBACK [WORK] [TO [SAVEPOINT] savepoint | FORCE 'text'];
- SAVEPOINT savepoint;
- LOCK TABLE relation_name;
- SET AUTOCOMMIT {ON | IMMEDIATE | OFF | *n*}
  - SHOW AUTOCOMMIT

Transactions in SQL (3)

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.

Transactions in SQL (4)

- Potential problem with lower isolation levels (cont.):
  - 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.

Transactions in SQL (5)

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

Transactions in SQL (6)

- 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:  ...
```