Ho Chi Minh City University of Technology
Faculty of Computer Science and Engineering

Chapter 4: Introduction to Transaction Processing Concepts and Theory

Database Management Systems (CO3021)

Computer Science Program

Dr. Võ Thị Ngọc Châu

(chauvtn@hcmut.edu.vn)

Semester 1 – 2018-2019

Course outline

- Overall Introduction to Database Management Systems
- Chapter 1. Disk Storage and Basic File Structures
- Chapter 2. Indexing Structures for Files
- Chapter 3. Algorithms for Query Processing and Optimization
- **Chapter 4. Introduction to Transaction Processing Concepts and Theory**
- Chapter 5. Concurrency Control Techniques
- Chapter 6. Database Recovery Techniques

References

- [1] R. Elmasri, S. R. Navathe, Fundamentals of Database Systems- 6th Edition, Pearson- Addison Wesley, 2011.
  - ***R. Elmasri, S. R. Navathe, Fundamentals of Database Systems- 7th Edition, Pearson, 2016.***
- [2] H. G. Molina, J. D. Ullman, J. Widom, Database System Implementation, Prentice-Hall, 2000.
- [3] H. G. Molina, J. D. Ullman, J. Widom, Database Systems: The Complete Book, Prentice-Hall, 2002
- [4] A. Silberschatz, H. F. Korth, S. Sudarshan, Database System Concepts –3rd Edition, McGraw-Hill, 1999.
- [Internet] …

Content

4.1. Introduction to Transaction Processing

- **Transaction processing systems** are systems with large databases and hundreds of *concurrent* users executing database *transactions*.
  - High availability and fast response time for hundreds of concurrent users
  - Examples: airline reservations, banking, credit card processing, online retail purchasing, stock markets, supermarket checkouts, etc.

→ the concept of a ***transaction***, which is used to represent a logical unit of database processing that must be completed in its entirety to ensure correctness

4.1. Introduction to Transaction Processing

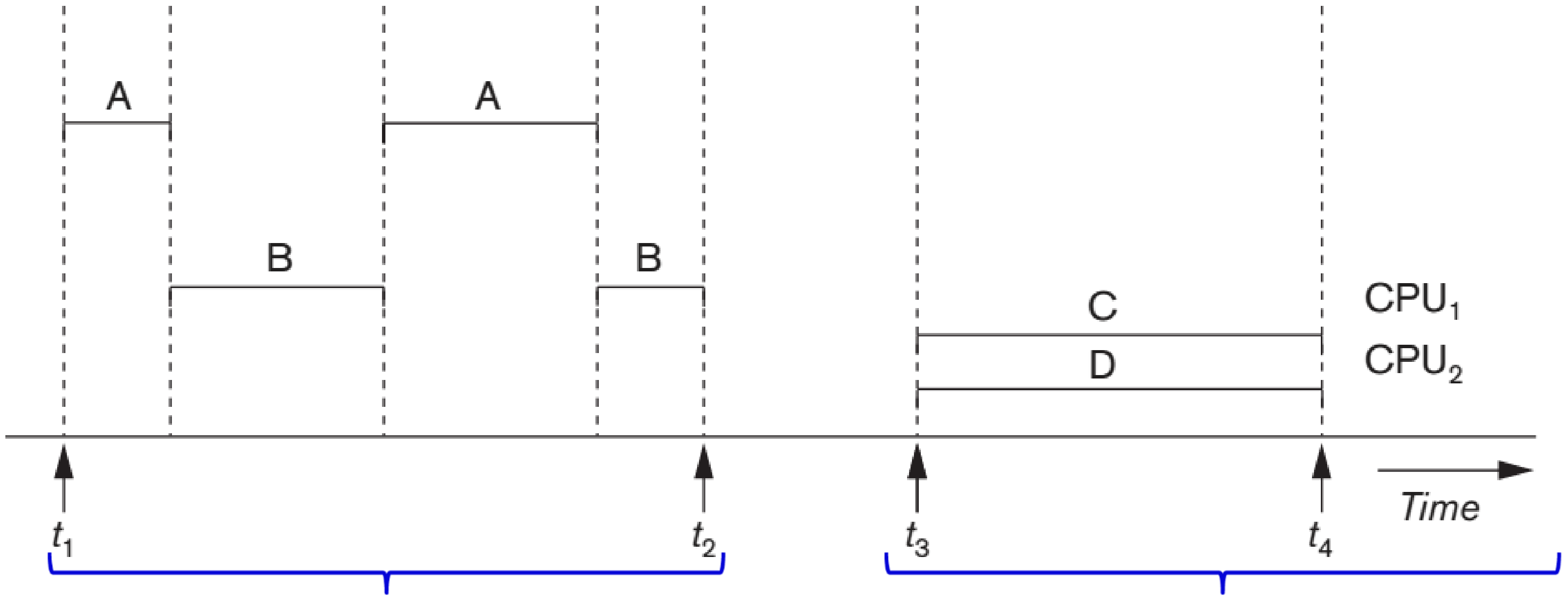
- Single-user vs. multi-user database systems
  - Concurrent execution of transactions in a multi-user system
  - Recovery from transaction failures when transactions fail while executing

→ The number of users who can use the system concurrently-that is, at the same time

  - Single-user if at most one user at a time can use the system
  - Multiuser if many users can use the system - and hence access the database - concurrently

4.1. Introduction to Transaction Processing

Interleaved processing versus parallel processing of concurrent transactions
Figure 20.1, [1], pp. 747



Concurrent execution of processes is actually interleaved. Interleaving keeps the CPU busy when a process requires an input or output (I/O) operation, such as reading a block from disk. The CPU is switched to execute another process rather than remaining idle during I/O time. Interleaving also prevents a long process from delaying other processes.

If the computer system has multiple hardware processors (CPUs), *parallel processing* of multiple processes is possible.

4.1. Introduction to Transaction Processing

- **A Transaction:** logical unit of database processing that includes one or more database access operations (read – retrieval, write – insert, update, delete).
  - Typically implemented by a computer program that includes database commands
  - The database operations that form a transaction can either be embedded within an application program or they can be specified interactively via a high-level query language such as SQL.
- **Transaction boundaries**: Begin and End transaction statements.
- An **application program** may contain several transactions separated by the transaction boundaries.

4.1. Introduction to Transaction Processing

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

- **A database -** collection of named data items
- **Granularity of data** - a field, a record , or a whole disk block (Concepts are independent of granularity)
- Basic operations are **read** and **write**
  - **read_item(X)**: Reads a database item named *X* into a program variable. To simplify our notation, we assume that *the program variable is also named X.*
  - **write_item(X)**: Writes the value of program variable *X* into the database item named *X*.

4.1. Introduction to Transaction Processing

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*.

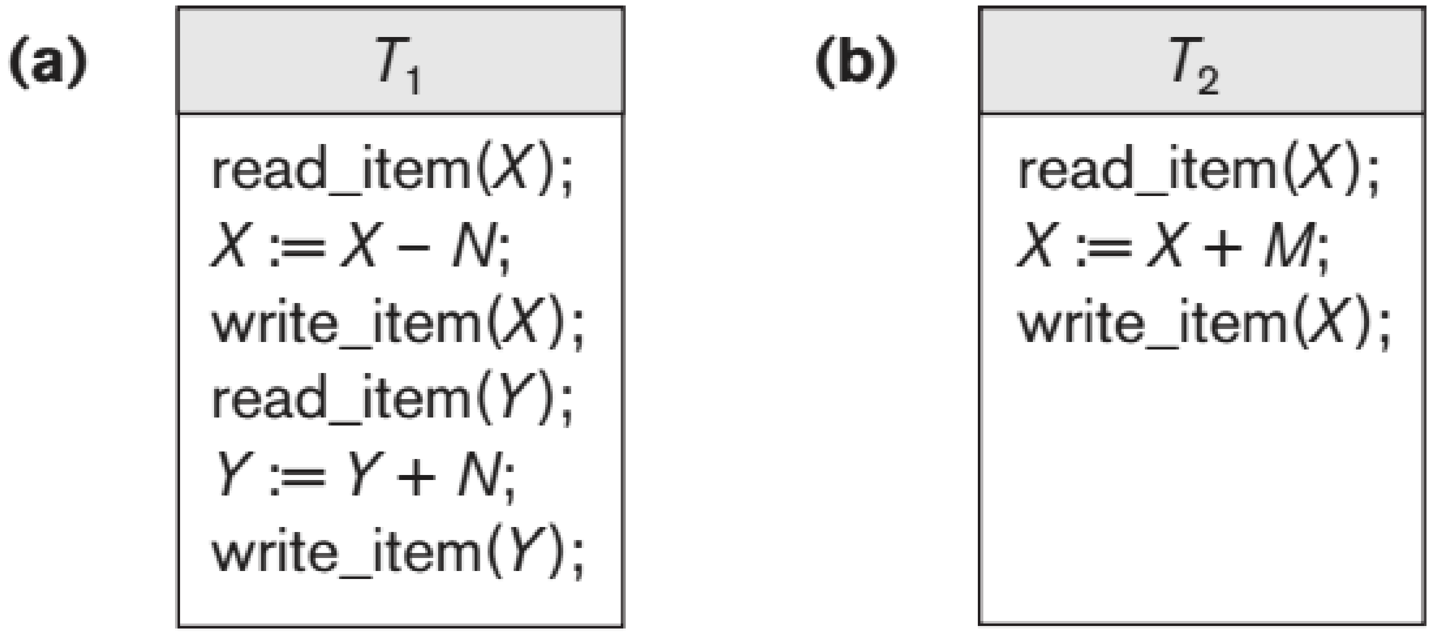
4.1. Introduction to Transaction Processing

READ AND WRITE OPERATIONS:

- **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).

4.1. Introduction to Transaction Processing



(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($X$);
```

Two sample transactions.

(a) Transaction T_1.

(b) Transaction T_2.

Figure 20.2, [1], pp. 750.

4.1. Introduction to Transaction Processing

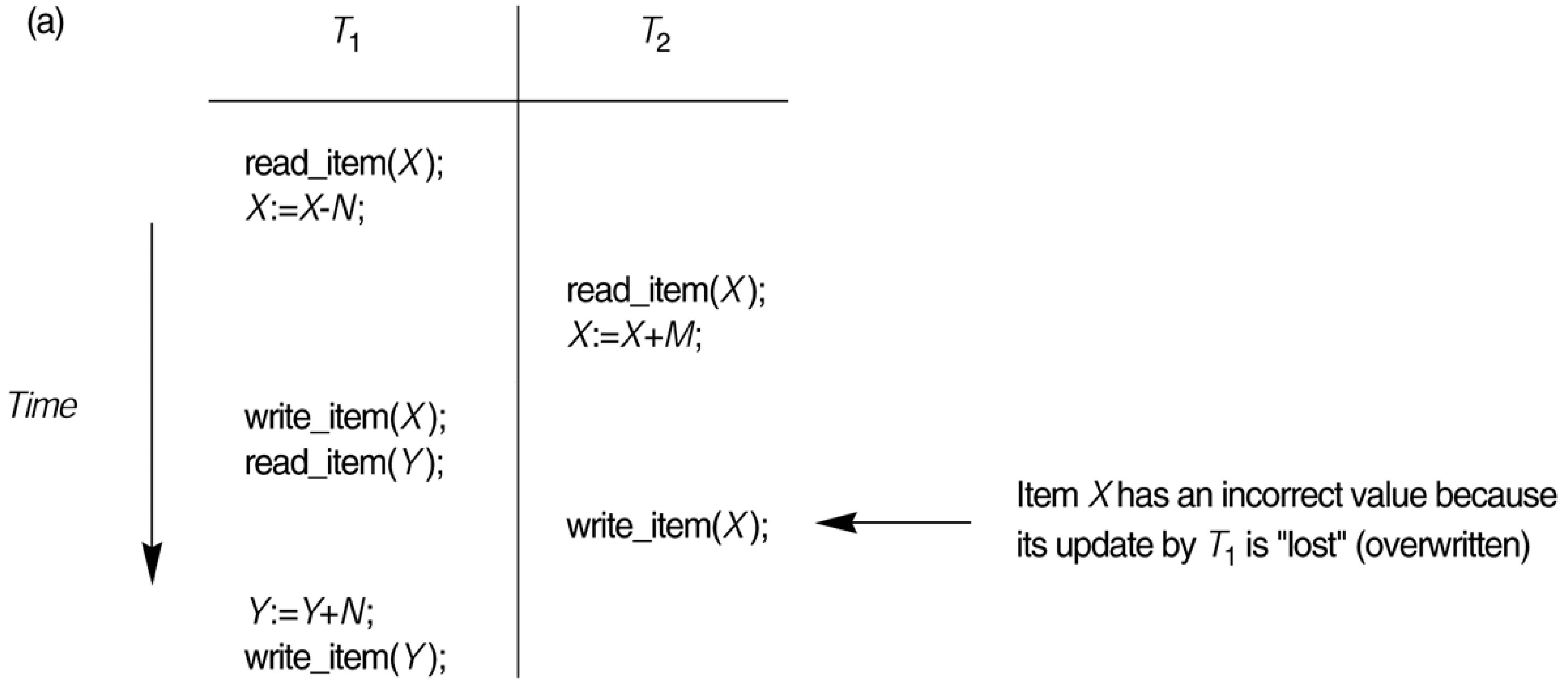
- Why concurrency control is needed

→ The problems encountered with the transactions running concurrently if uncontrolled

  - The lost update problem
  - The temporary update (or dirty read) problem
  - The incorrect summary problem
  - The unrepeatable problem
  - The phantom problem

4.1. Introduction to Transaction Processing

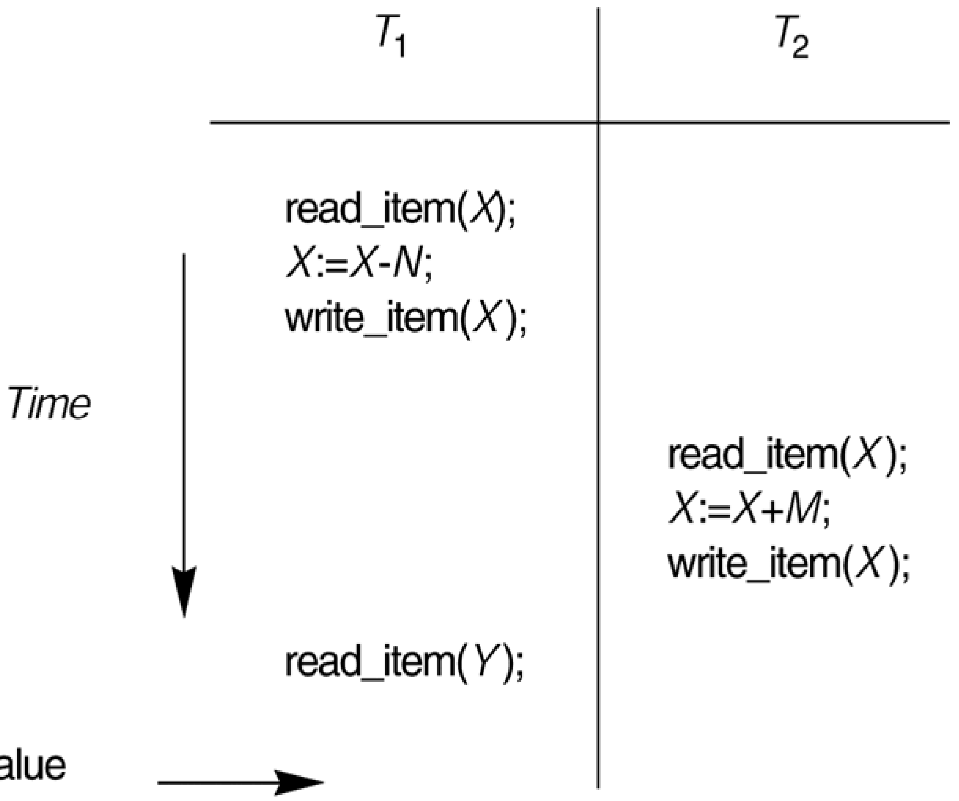
- The *lost update* problem
  - This problem occurs when two transactions that access the same database items have their operations interleaved in a way that makes the value of some database items incorrect.



4.1. Introduction to Transaction Processing

- The *temporary update* (or *dirty read*) problem
  - This problem 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.

(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.

(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*).

- 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.

4.1. Introduction to Transaction Processing

- The *unrepeatable (nonrepeatable)* problem
  - A transaction T reads an item twice and the item is changed by another transaction T' between the two reads. Hence, T receives *different values* for its two reads of the same item.

4.1. Introduction to Transaction Processing

- The *phantom* problem
  - 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 used in T1, into the table used by T1. If T1 is repeated, then T1 will see a phantom, a row that previously did not exist.

4.1. Introduction to Transaction Processing

- Why recovery is needed
  - Whenever a transaction is submitted to a DBMS for execution, the system is responsible for making sure:
    - either (1) all the operations in the transaction are completed successfully and their effect is recorded permanently in the database,
    - or (2) the transaction has no effect whatsoever on the database or on any other transactions.
  - The DBMS must not permit some operations of a transaction T to be applied to the database while other operations of T are not.
  - This may happen if a transaction fails after executing some of its operations but before executing all of them.

4.1. Introduction to Transaction Processing

- Several possible reasons for a transaction to fail in the middle of execution:
  - 1. A *computer failure (system crash)*
  - 2. A *transaction* or *system error*
  - 3. *Local errors* or *exception conditions detected by the transaction*
  - 4. *Concurrency control enforcement*
  - 5. *Disk failure*
  - 6. *Physical problems and catastrophes*

→ Recovery and backup

4.1. Introduction to Transaction Processing

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.

4.1. Introduction to Transaction Processing

Why recovery is needed:

3. **Local errors or exception conditions** detected by the transaction:

   - certain conditions necessitate cancellation of the transaction. For example, data for the transaction may not be found. A condition, such as insufficient account balance in a banking database, may cause a transaction, such as a fund withdrawal from that account, to be canceled.

   - a programmed abort in the transaction causes it to fail.

4. **Concurrency control enforcement:** The concurrency control method may decide to abort the transaction, to be restarted later, because it violates serializability or because several transactions are in a state of deadlock.

4.1. Introduction to Transaction Processing

Why recovery is needed:

5. **Disk failure:** Some disk blocks may lose their data because of a read or write malfunction or because of a disk read/write head crash. This may happen during a read or a write operation of the transaction.

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

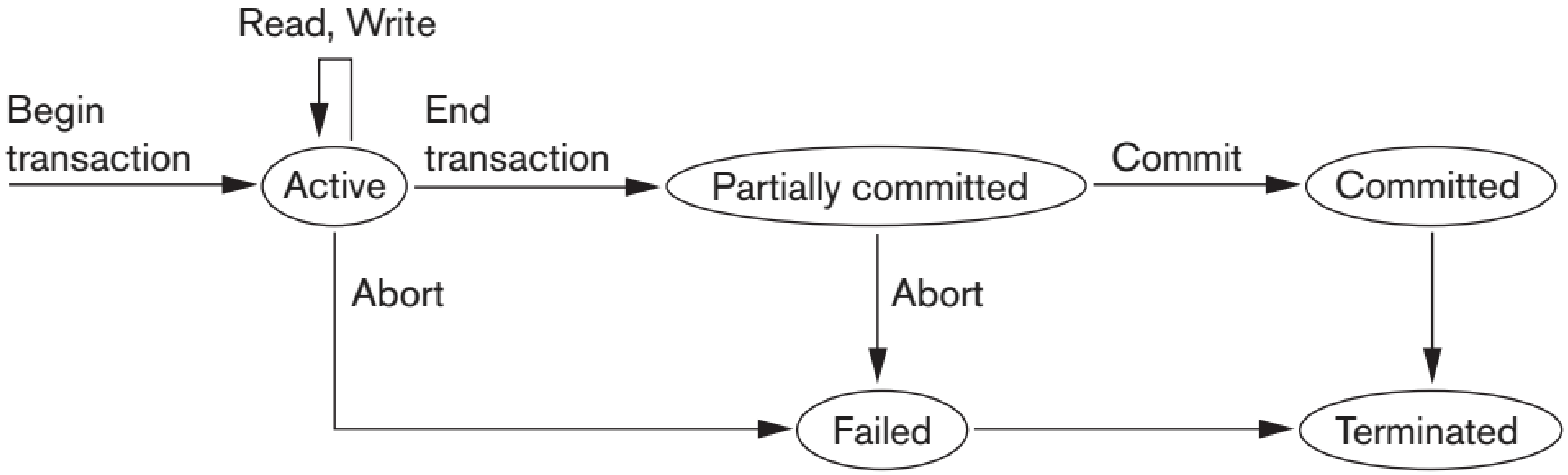
4.2. Transaction and System Concepts

A **transaction** is an atomic (*logical*) unit of work that is either completed in its entirety or not done at all.

Transaction states:

- Active state
- Partially committed state
- Committed state
- Failed state
- Terminated State

4.2. Transaction and System Concepts



State transition diagram illustrating the states for transaction execution.

Figure 20.4, [1], pp. 754.

4.2. Transaction and System Concepts

- Transaction states
  - *Active state*: a transaction goes into an active state immediately after it starts execution, where it can issue READ and WRITE operations.
  - *Partially committed*: when the transaction ends, some recovery protocols need to ensure that a system failure will not result in an inability to record the changes of the transaction permanently (usually by recording changes in the system log).
  - *Committed*: once this check is successful, the transaction is said to have reached its commit point. Once a transaction is committed, it has concluded its execution successfully and all its changes must be recorded permanently in the database.
  - *Failed*: if one of the checks fails or if the transaction is aborted during its active state. The transaction may then have to be rolled back to undo the effect of its WRITE operations on the database. Failed or aborted transactions may be *restarted* later - either automatically or after being resubmitted by the user-as brand new transactions.
  - *Terminated*: the transaction leaving the system. The transaction information that is maintained in system tables while the transaction has been running is removed when the transaction terminates.

4.2. Transaction and System Concepts

For ***recovery*** purposes, the system needs to keep track of when the transaction starts, terminates, and commits or aborts.

- **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.

4.2. Transaction and System Concepts

For ***recovery*** purposes, the system needs to keep track of when the transaction starts, terminates, and commits or aborts.

- **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*.

4.2. Transaction and System Concepts

For ***recovery*** purposes, the system needs to keep track of when the transaction starts, terminates, and commits or aborts.

- **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.

4.2. Transaction and System Concepts

The System Log

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

4.2. Transaction and System Concepts

The System Log

Types of log record:

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

4.2. Transaction and System Concepts

The System Log

- Protocols for recovery that <u>avoid cascading rollbacks do not require that READ operations be written to the system log</u>, whereas other protocols require these entries for recovery.
  - Cascading rollback is the case such that the rolling-back of transaction T1 makes transaction T2 rollback and so on.
- <u>Strict</u> protocols require simpler WRITE entries that do <u>not include *new_value*</u>.
  - Strict implies no cascading rollbacks and further no committed values used in any read/write operation.

4.2. Transaction and System Concepts

The System Log

- Given the following executions, write the content of the system log. Supposed that there are originally X = 10 and Y = 5 in the database.

T1	T2
read_item(X);	
	read_item(X);
X := X+2;	
write_item(X);	
read_item(Y);	
	X := X-1;
	write_item(X);
	commit;
Y := Y+2;	
write_item(Y);	
commit;	

[start_transaction, T1]
[read_item, T1, X]
[start_transaction, T2]
[read_item, T2, X]
[write_item, T1, X, 10, 12]
[read_item, T1, Y]
[write_item, T2, X, 10, 9]
[commit, T2]
[write_item, T1, Y, 5, 7]
[commit, T1]

Log content

4.2. Transaction and System Concepts

The System Log

- Given the following executions, write the content of the system log. Supposed that there are originally X = 10 and Y = 5 in the database.

T1	T2
read_item(X);	
X := X+2;	
write_item(X);	
	read_item(X);
read_item(Y);	
Y := Y+2;	
write_item(Y);	
abort;	
	X := X-1;
	write_item(X);
	abort;

Cascading rollback:

T1 aborts and rollbacks.

Then, T2 aborts and rollbacks.

4.2. Transaction and System Concepts

The System Log

- Given the following executions, write the content of the system log. Supposed that there are originally X = 10 and Y = 5 in the database.

T1	T2
read_item(X);	
	read_item(X);
X := X+2;	
write_item(X);	
read_item(Y);	
	X := X-1;
	write_item(X);
Y := Y+2;	
write_item(Y);	
abort;	
	commit;

//no cascading rollback

//no record for read operations

[start_transaction, T1]

[start_transaction, T2]

[write_item, T1, X, 10, 12]

[write_item, T2, X, 10, 9]

[write_item, T1, Y, 5, 7]

[abort, T1]

[commit, T2]

Log content

4.2. Transaction and System Concepts

The System Log

- Given the following executions, write the content of the system log. Supposed that there are originally X = 10 and Y = 5 in the database.

T1	T2
read_item(X);	
	read_item(X);
X := X+2;	
write_item(X);	
read_item(Y);	
Y := Y+2;	
write_item(Y);	
commit;	
	X := X-1;
	write_item(X);
	commit;

//strict: read & write committed data items

//no record for read operations

//no new value in write records

[start_transaction, T1]
[start_transaction, T2]
[write_item, T1, X, 10]
[write_item, T1, Y, 5]
[commit, T1]
[write_item, T2, X, 10]
[commit, T2]

Log content

4.2. Transaction and System Concepts

- Recovery from a transaction failure
  - Undoing or redoing transaction operations individually from the log
    - If the system crashes, recover to a consistent database state by examining the log and using a recovery technique.
    - Undoing 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
    - Redoing the operations of a transaction T by *tracing forward* through the log and setting all items changed by a WRITE operation of T to their new_values

4.2. Transaction and System Concepts

Commit Point of a Transaction

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

4.2. Transaction and System Concepts

Commit Point of a Transaction

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

4.3. Desirable Properties of Transactions

- Transactions possess the ACID properties enforced by the concurrency control and recovery methods.
  - Atomicity
    - A transaction is an atomic unit of processing; it is either performed in its entirety or not performed at all.
  - Consistency preservation
    - A transaction is consistency preserving if its complete execution take(s) the database from one consistent state to another.
  - Isolation
    - A transaction should appear as though it is being executed in isolation from other transactions. That is, the execution of a transaction should not be interfered with by any other transactions executing concurrently.
  - Durability (or permanency)
    - The changes applied to the database by a committed transaction must persist in the database. These changes must not be lost because of any failure.

4.4. Characterizing Schedules based on Recoverability

- Schedule (or history)
  - The *order of execution of operations* from the various transactions when transactions are executing concurrently in an interleaved fashion
  - Types of schedules
    - Based on *recoverability*
      - Schedules facilitate recovery from transaction failures occur.
    - Based on *serializability*
      - Schedules with the interference of participating transactions

4.4. Characterizing Schedules based on Recoverability

- Constraint on schedules
  - Consider a schedule (or history) *S* of *n* transactions T_1, T_2, …, T_n
  - For each transaction T_i that participates in *S*, the operations of T_i in *S* must appear in the same order in which they occur in T_i. However, operations from other transactions T_j can be interleaved with the operations of T_i in *S*.

4.4. Characterizing Schedules based on Recoverability

TIME ↓

T1	T2	Interleaved Execution
read_item(X);		$r_1(X)$
X:=X-N;		
write_item(X);		$w_1(X)$
	read_item(X);	$r_2(X)$
	X:=X+M;	
	write_item(X);	$w_2(X)$
read_item(Y);		$r_1(Y)$
abort		a_1
	abort	a_2

Schedule S_b: $r_1(X)$; $w_1(X)$; $r_2(X)$; $w_2(X)$; $r_1(Y)$; a_1; a_2

$r_1(X)$ = T_1 reads X.

$w_1(X)$ = T_1 writes X.

a_1 = abort of T_1

c_3 = commit of T_3

4.4. Characterizing Schedules based on Recoverability

$S_a: r_1(X); r_2(X); w_1(X); r_1(Y); w_2(X); w_1(Y);$

Schedule S_a

write (w) operation of transaction T_1

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);	

TIME ↓

Shorthand notation:

b = *begin_transaction*,

r = *read_item*,

w = *write_item*,

e = *end_transaction*,

c = *commit*,

a = *abort*.

4.4. Characterizing Schedules based on Recoverability

- *Recoverable* schedules
  - *Recoverable* schedule: once a transaction T is committed, it should never be necessary to roll back T.
    - No transaction T in a schedule *commits* until all transactions T' that have written an item that T reads have committed.
  - *Cascadeless* schedule (to avoid cascading rollback): if every transaction in the schedule *reads* only items that were written by committed transactions.
  - *Strict* schedule: transactions can neither *read* nor *write* an item X until the last transaction that wrote X has committed (or aborted).
- *Nonrecoverable* schedules
  - *Nonrecoverable* schedule: once a transaction T is *committed*, T is asked to be *rolled-back*.

4.4. Characterizing Schedules based on Recoverability

S_a': $r_1(X); r_2(X); w_1(X); r_1(Y); w_2(X); c_2; w_1(Y); c_1;$

S_a' is **recoverable**, even though it suffers from the *lost update problem*. → Why?

S_c: $r_1(X); w_1(X); r_2(X); r_1(Y); w_2(X); c_2; a_1;$

S_c is **not recoverable**. → Why?

S_d: $r_1(X); w_1(X); r_2(X); r_1(Y); w_2(X); w_1(Y); c_1; c_2;$

S_d is **recoverable**. → Why?

S_e: $r_1(X); w_1(X); r_2(X); r_1(Y); w_2(X); w_1(Y); a_1; a_2;$

S_e is **recoverable**; but **not cascadeless**. → Why?

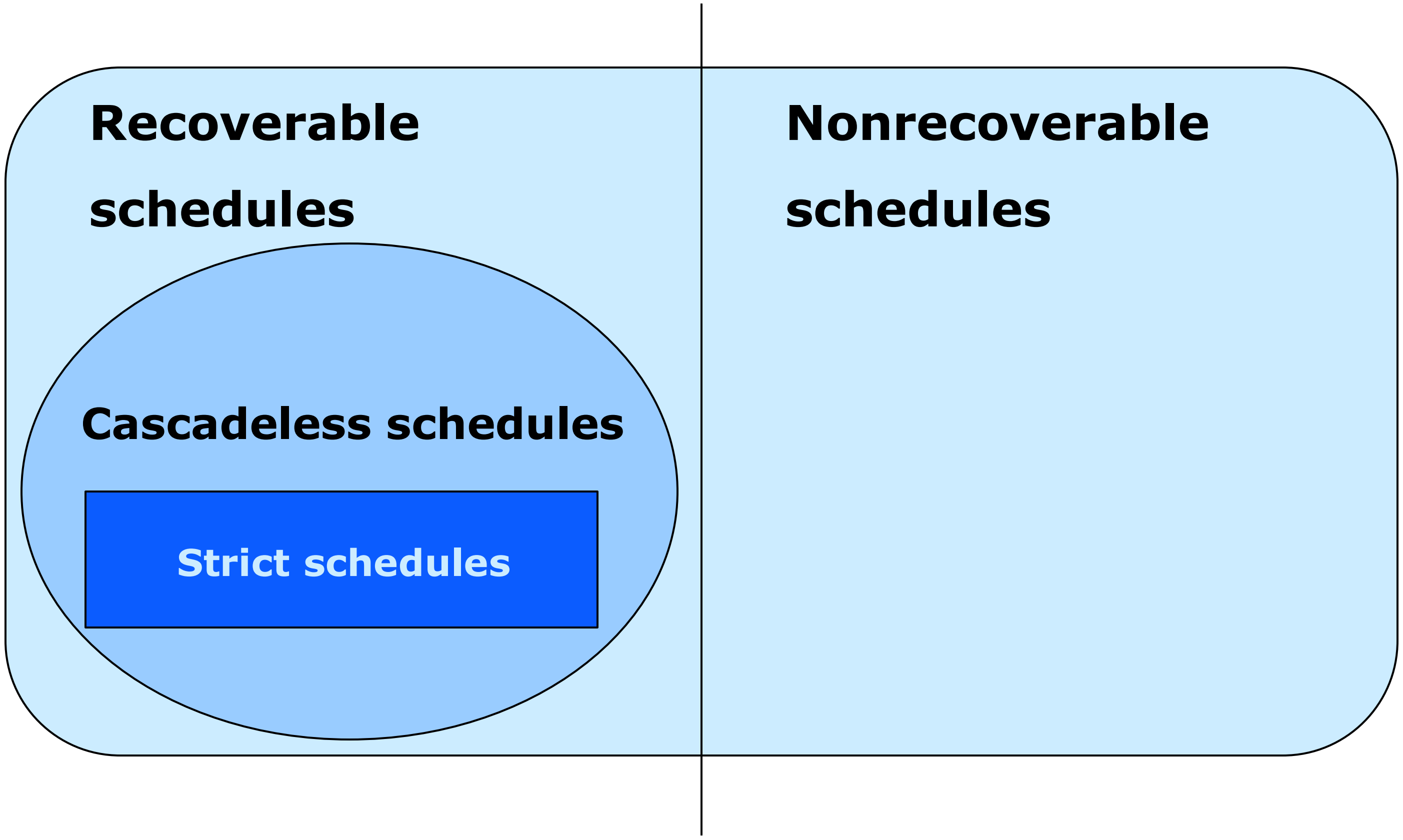
S_f: $r_1(X); w_1(X); r_1(Y); w_1(Y); c_1; r_2(X); w_2(X); c_2;$

S_f is **cascadeless**. → Why?

S_g: $r_1(X); r_2(Z); r_1(Z); r_3(X); r_3(Y); w_1(X); c_1; w_3(Y); c_3; r_2(Y); w_2(Z); r_2(Y); c_2;$

S_g is **strict**. → Why?

4.4. Characterizing Schedules based on Recoverability



Relationships between schedules based on recoverability

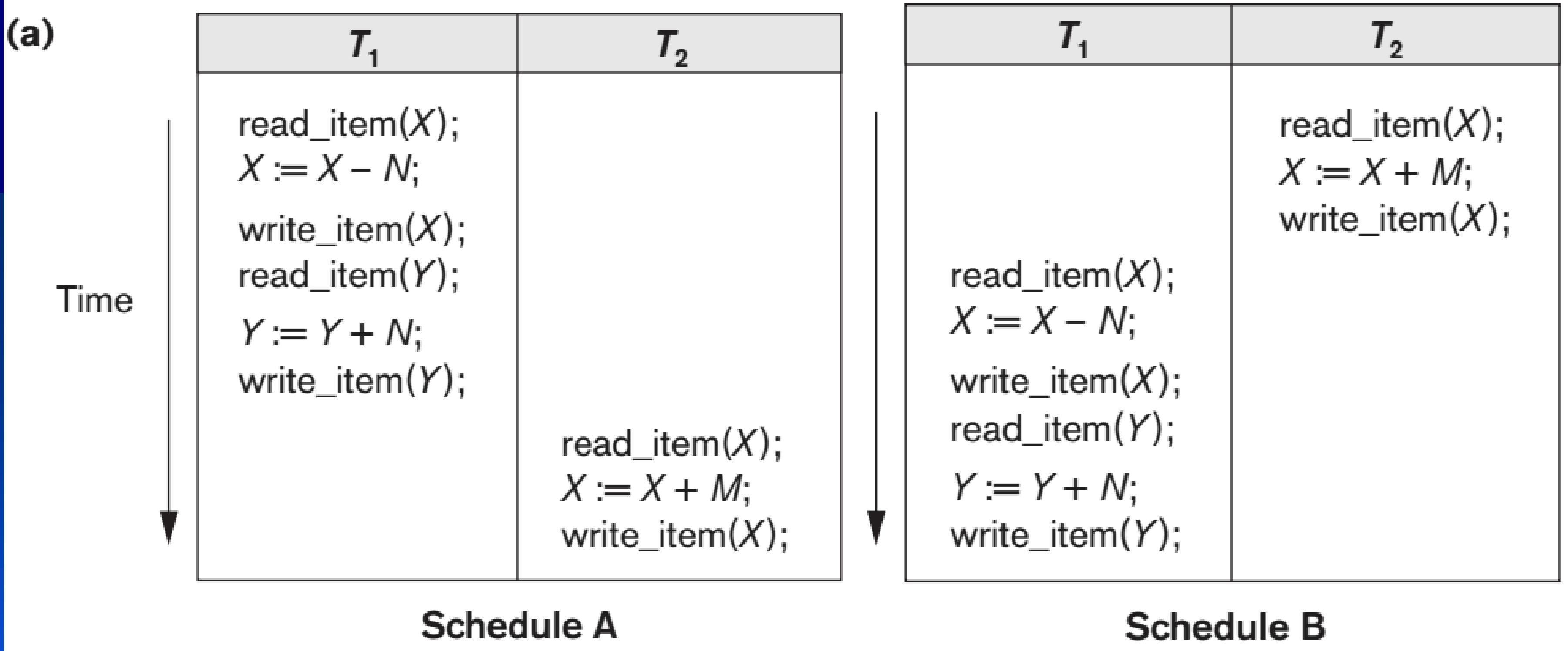
All strict schedules are cascadeless.

All cascadeless schedules are recoverable.

4.4. Characterizing Schedules based on Recoverability

- *Recoverable* schedules
  - *Recoverable* schedule
  - *Cascadeless* schedule (to avoid cascading rollback): if every transaction in the schedule ***reads*** only items that were written by committed transactions.
    - No log record for READ operations in the system log
  - *Strict* schedule: transactions can neither ***read*** nor ***write*** an item X until the last transaction that wrote X has committed (or aborted).
    - Log records for WRITE operations in the system log do not include new_value of data item X.
    - Strict schedules simplify the recovery process.
- ***Nonrecoverable*** schedules

4.5. Characterizing Schedules based on Serializability



(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

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

Figure 20.5. (a) Serial schedule A: T_1 followed by T_2. (b) Serial schedule B: T_2 followed by T_1.

[1], pp. 764

Originally in DB: X = 5; Y = 10

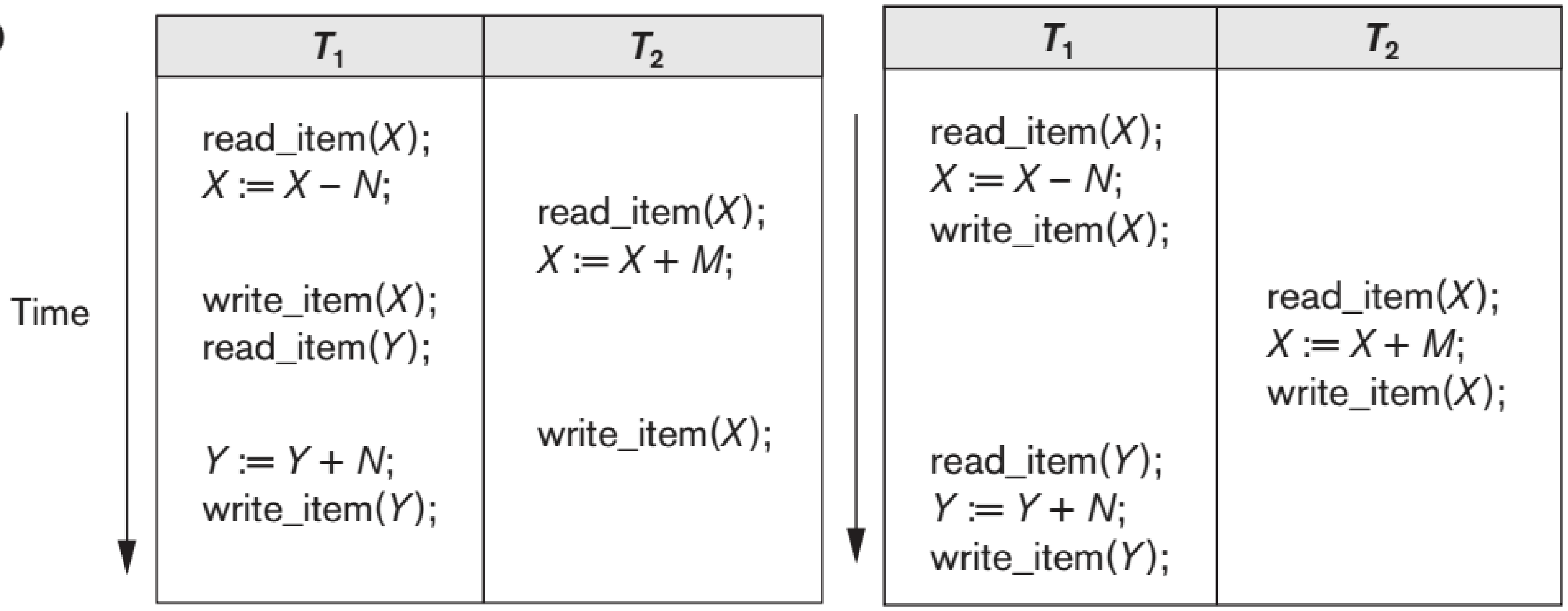
Constants: N = 1; M = 2

Schedule A: X = 6; Y = 11

Schedule B: X = 6; Y = 11

4.5. Characterizing Schedules based on Serializability

(c)



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

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

Schedule D

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

Originally in DB: X = 5; Y = 10

Constants: N = 1; M = 2

Schedule A: X = 6; Y = 11

Schedule B: X = 6; Y = 11

Which schedules are correct with interleaving of operations? Why?

Schedule C: X = 7; Y = 11

Schedule D: X = 6; Y = 11

4.5. Characterizing Schedules based on Serializability

- The concept of serializability of schedules is used to identify which schedules are correct when transaction executions have interleaving of their operations in the schedules.
- Serial schedules
  - The operations of each transaction are executed consecutively, without any interleaved operations from the other transaction.
- Nonserial schedules
  - The operations of each transaction are executed in an interleaved fashion with the operations of other transactions.

4.5. Characterizing Schedules based on Serializability

- Serial schedules
  - Only one transaction at a time is active - the commit (or abort) of the active transaction initiates execution of the next transaction.
  - No interleaving occurs in a serial schedule.
  - If we consider the transactions to be *independent,* is that every serial schedule is considered correct.
  - The problem with serial schedules is that they limit concurrency or interleaving of operations.
    - if a transaction waits for an operation to complete, we cannot switch the CPU processor to another transaction, thus wasting valuable CPU processing time.
    - If some transaction T is quite long, the other transactions must wait for T to complete all its operations before commencing.

4.5. Characterizing Schedules based on Serializability

- Nonserial schedules
  - Some nonserial schedules give the *correct* expected result.
  - We would like to determine which of the nonserial schedules always give a *correct* result and which may give *erroneous* results.
  - The concept used to characterize schedules in this manner is that of *serializability* of a schedule.

  → *Serializable schedules*

4.5. Characterizing Schedules based on Serializability

- Serializable schedules
  - A schedule S of *n* transactions is **serializable** if it is ***equivalent** to some serial schedule* of the same *n* transactions.
  - Two disjoint groups of the nonserial schedules:
    - Those that are equivalent to **one** (or **more**) of the serial schedules → **serializable** schedules
    - Those that are not equivalent to ***any*** serial schedule and hence are not serializable
  - Saying that a nonserial schedule S is serializable, is equivalent to a serial schedule, saying that it is **correct**, because a serial schedule is considered correct.

4.5. Characterizing Schedules based on Serializability

- Equivalence of schedules
  - Result equivalence
  - *Conflict equivalence*
  - View equivalence

4.5. Characterizing Schedules based on Serializability

- Equivalence of schedules
  - Conflict equivalence
    - Two schedules are said to be conflict equivalent if the order of any two *conflicting operations* is the same in both schedules.
      - Two operations in a schedule are said to *conflict* if they belong to different transactions, access the same database item, and at least one of the two operations is a write_item operation.

$r_1(X)$ vs. $r_2(X)$	X	$r_1(X)$ vs. $r_2(Y)$	X
$r_1(X)$ vs. $w_2(X)$	√	$r_1(X)$ vs. $w_2(Y)$	X
$w_1(X)$ vs. $r_2(X)$	√	$w_1(X)$ vs. $r_2(Y)$	X
$w_1(X)$ vs. $w_2(X)$	√	$w_1(X)$ vs. $w_2(Y)$	X

4.5. Characterizing Schedules based on Serializability

- Equivalence of schedules
  - Conflict equivalence

Case 1:

S_a: … $w_1(X)$ … $r_2(X)$ … $\approx$ S_b: … $w_1(X)$ … $r_2(X)$ …

S_a: … $w_1(X)$ … $r_2(X)$ … $\neq$ $S_{b'}$: … $r_2(X)$ … $w_1(X)$ …

Case 2:

S_a: … $r_1(X)$ … $w_2(X)$ … $\approx$ S_b: … $r_1(X)$ … $w_2(X)$ …

S_a: … $r_1(X)$ … $w_2(X)$ … $\neq$ $S_{b'}$: … $w_2(X)$ … $r_1(X)$ …

Case 3:

S_a: … $w_1(X)$ … $w_2(X)$ … $\approx$ S_b: … $w_1(X)$ … $w_2(X)$ …

S_a: … $w_1(X)$ … $w_2(X)$ … $\neq$ $S_{b'}$: … $w_2(X)$ … $w_1(X)$ …

4.5. Characterizing Schedules based on Serializability

- Equivalence of schedules
  - Conflict equivalence
    - *Conflict serializable* schedule S if it is *conflict equivalent* to some serial schedule S'.
      - In such a case, we can reorder the *nonconflicting* operations in S until we form the equivalent serial schedule S'.
      - Schedules in Figure 20.5

S_a: $r_1(X)$; $w_1(X)$; $r_1(Y)$; $w_1(Y)$; $r_2(X)$; $w_2(X)$ → serial?

S_b: $r_2(X)$; $w_2(X)$; $r_1(X)$; $w_1(X)$; $r_1(Y)$; $w_1(Y)$ → serial?

S_c: $r_1(X)$; $r_2(X)$; $w_1(X)$; $r_1(Y)$; $w_2(X)$; $w_1(Y)$ → not serializable?

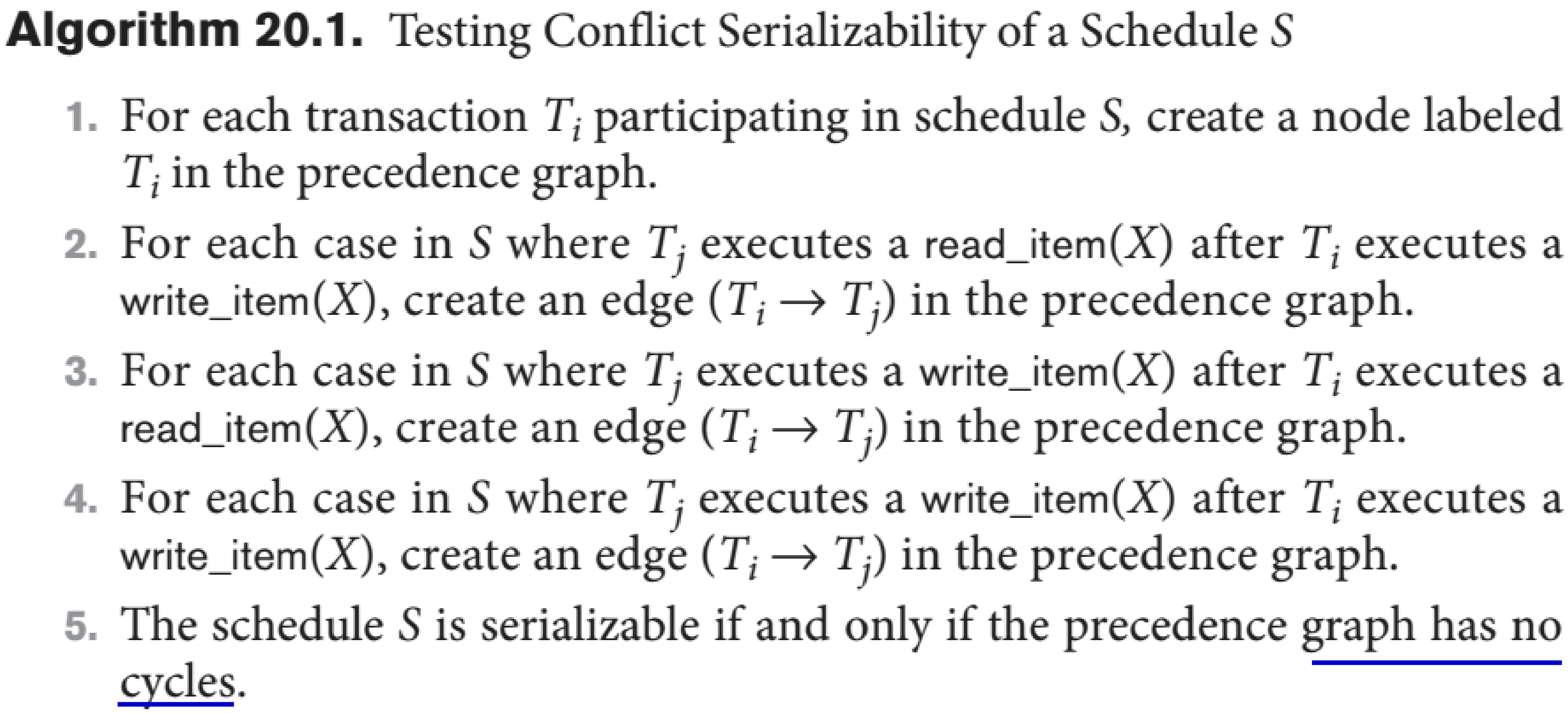
S_d: $r_1(X)$; $w_1(X)$; $r_2(X)$; $w_2(X)$; $r_1(Y)$; $w_1(Y)$ → conflict serializable?

4.5. Characterizing Schedules based on Serializability

- Equivalence of schedules
  - Conflict equivalence
    - *Conflict serializable* schedule S → **Test** for conflict serializability of a schedule
      - The algorithm looks at only the read_item and write_item operations in a schedule to construct a ***precedence graph*** (or ***serialization graph***), which is a directed graph G *(N,* E) that consists of a set of nodes N = $\{T_1, T_2, \ldots, T_n\}$ and a set of directed edges E = $\{e_1, e_2, \ldots, e_m\}$.
      - There is one node in the graph for each transaction T_i in the schedule.
      - Each edge e_i in the graph is of the form ($T_j \rightarrow T_k$), $1 \le j \le n$, $1 \le k \le n$, where T_j is the *starting node* of e_i and T_k is the *ending node* of e_i. Such an edge is created if one of the operations in T_j appears in the schedule *before* some *conflicting operation* in T_k.

4.5. Characterizing Schedules based on Serializability

- Equivalence of schedules
  - Conflict equivalence
    - *Conflict serializable* schedule S



Algorithm 20.1. Testing Conflict Serializability of a Schedule S

1. For each transaction T_i participating in schedule S, create a node labeled T_i in the precedence graph.
2. For each case in S where T_j executes a read_item(X) after T_i executes a write_item(X), create an edge ($T_i \rightarrow T_j$) in the precedence graph.
3. For each case in S where T_j executes a write_item(X) after T_i executes a read_item(X), create an edge ($T_i \rightarrow T_j$) in the precedence graph.
4. For each case in S where T_j executes a write_item(X) after T_i executes a write_item(X), create an edge ($T_i \rightarrow T_j$) in the precedence graph.
5. The schedule S is serializable if and only if the precedence graph has no cycles.

[1], pp. 767.

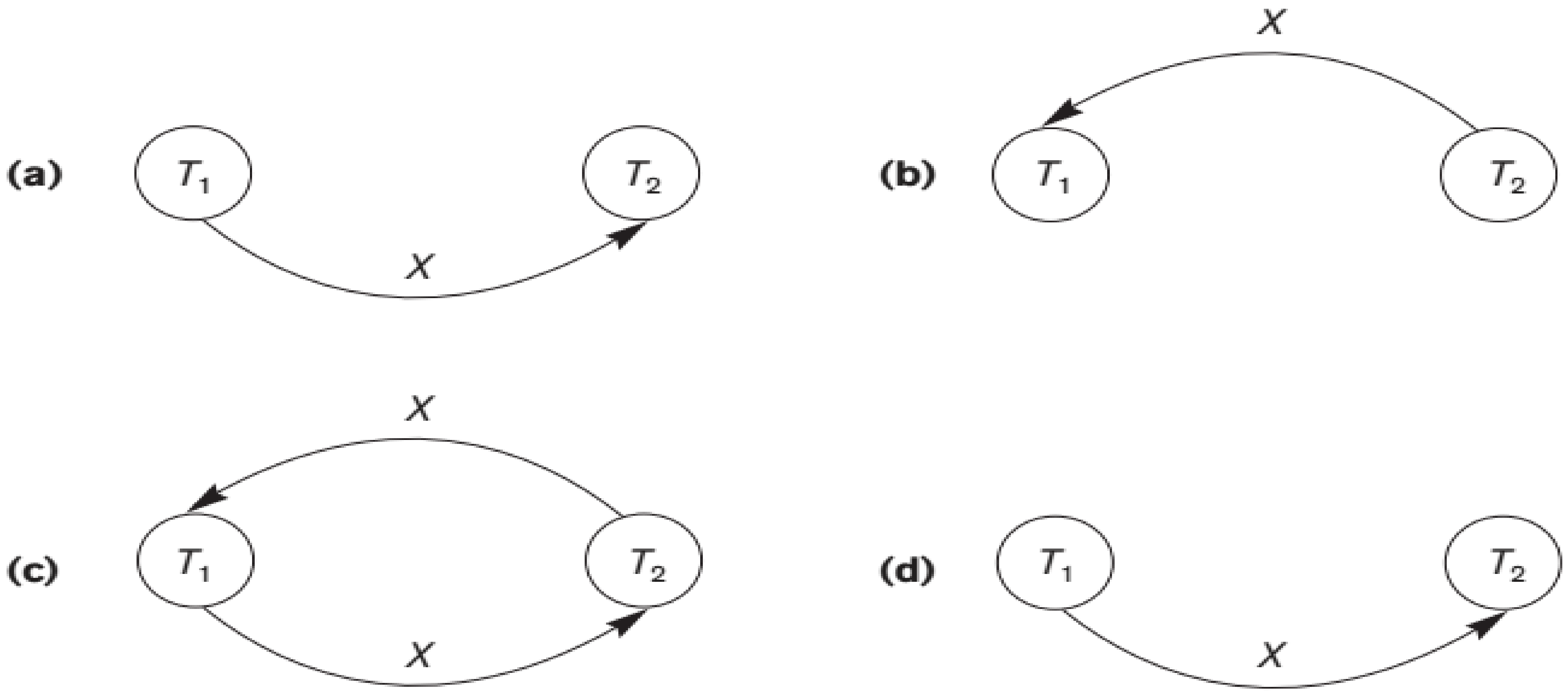


Figure 20.7
Constructing the precedence graphs for schedules A to D from Figure 20.5 to test for conflict serializability. (a) Precedence graph for serial schedule A. (b) Precedence graph for serial schedule B. (c) Precedence graph for schedule C (not serializable). (d) Precedence graph for schedule D (serializable, equivalent to schedule A).

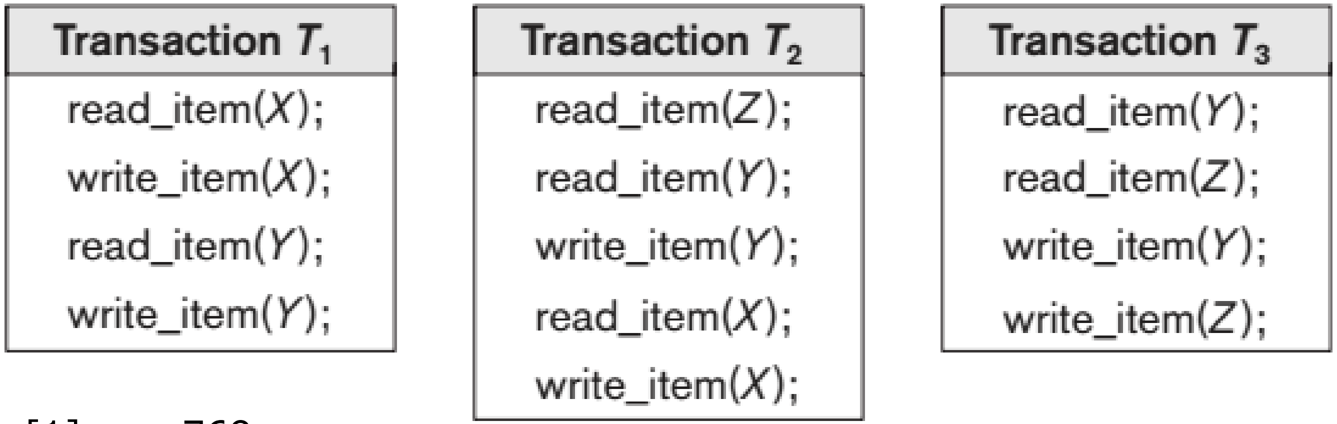
S_a: $r_1(X)$; $w_1(X)$; $r_1(Y)$; $w_1(Y)$; $r_2(X)$; $w_2(X)$ $\rightarrow$ serial

S_b: $r_2(X)$; $w_2(X)$; $r_1(X)$; $w_1(X)$; $r_1(Y)$; $w_1(Y)$ $\rightarrow$ serial

S_c: $r_1(X)$; $r_2(X)$; $w_1(X)$; $r_1(Y)$; $w_2(X)$; $w_1(Y)$ $\rightarrow$ not serializable

S_d: $r_1(X)$; $w_1(X)$; $r_2(X)$; $w_2(X)$; $r_1(Y)$; $w_1(Y)$ $\rightarrow$ conflict serializable

4.5. Characterizing Schedules based on Serializability



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);

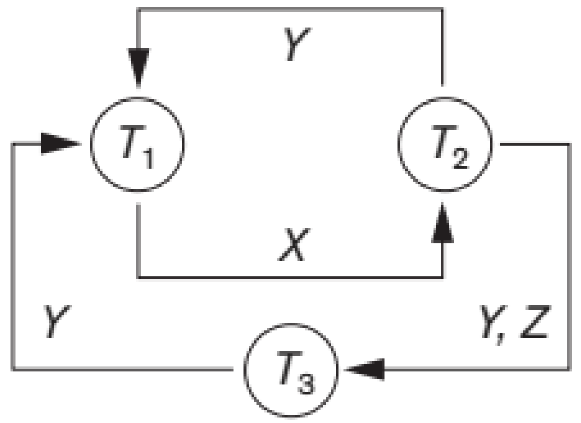
Figure 20.8, [1], pp. 769.

Another ***problem*** appears here: When transactions are submitted continuously to the system, it is *difficult to determine when a schedule begins and when it ends*. Serializability theory can be adapted to deal with this problem by considering only the committed projection of a schedule S. The *committed projection* C(S) of a schedule S includes only the operations in S that belong to committed transactions. We can theoretically define a schedule S to be serializable if its committed projection *C(S)* is equivalent to some serial schedule, since only committed transactions are guaranteed by the DBMS.

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



Equivalent serial schedules

None

Reason

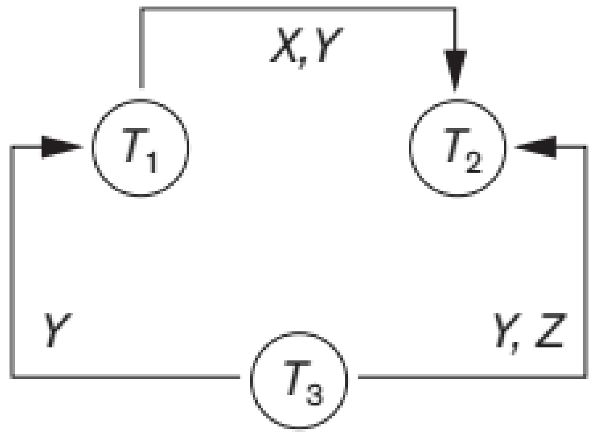
Cycle $X(T_1 \rightarrow T_2), Y(T_2 \rightarrow T_1)$

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

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



Equivalent serial schedules

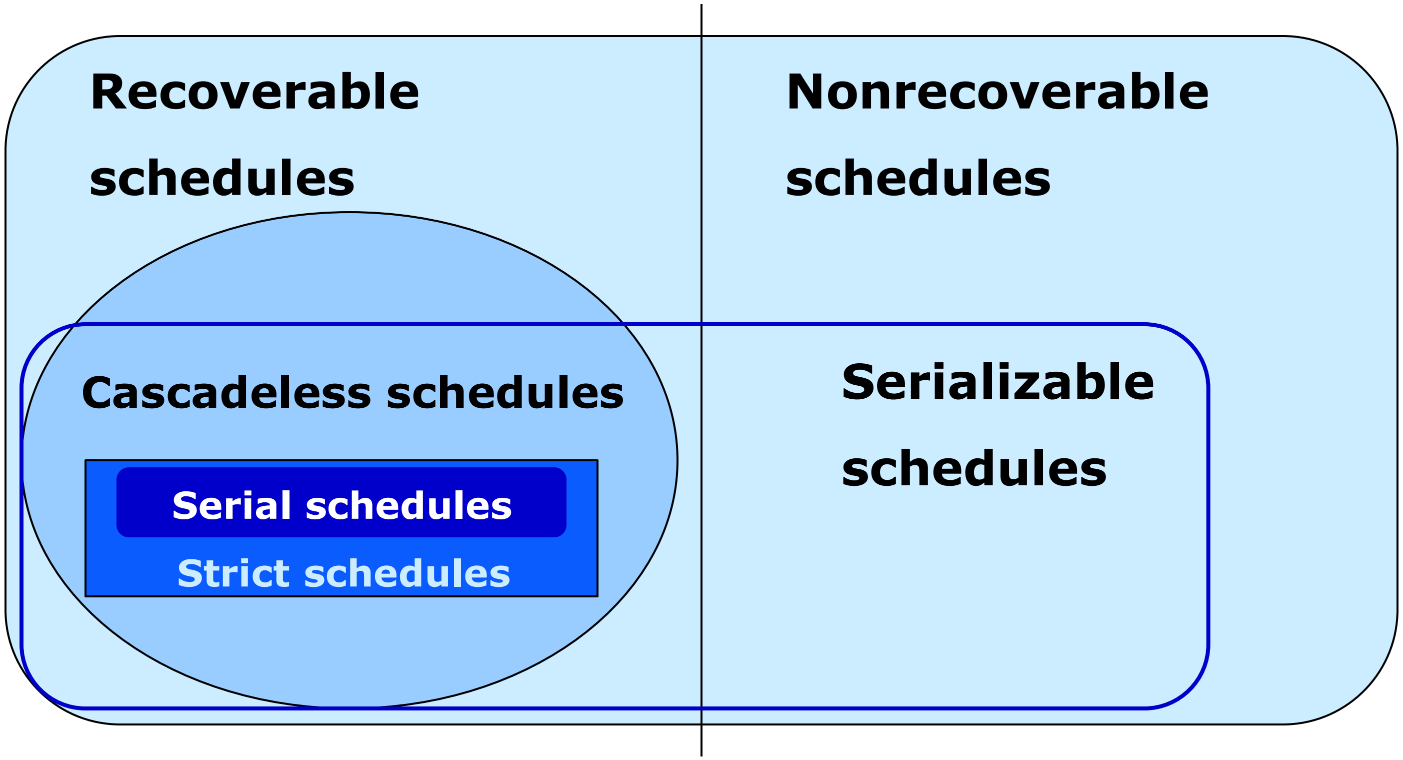
$T_3 \rightarrow T_1 \rightarrow T_2$

4.5. Characterizing Schedules based on Serializability

- Check conflict-serializability of each following schedule. Determine their equivalent serial schedules. Your turn!
  - S_1: $w_1(Y); w_2(Y); w_2(X); w_1(X); w_3(X);$
  - S_2: $r_2(A); r_1(B); w_2(A); r_3(A); w_1(B); w_3(A); r_2(B); w_2(B);$
  - S_3: $r_2(A); r_1(B); w_2(A); r_2(B); r_3(A); w_1(B); w_3(A); w_2(B);$
  - S_4: $r_1(A); r_2(A); r_3(B); w_1(A); r_2(C); r_2(B); w_2(B); w_1(C);$
  - S_5: $r_1(A); w_1(B); r_2(B); w_2(C); r_3(C); w_3(A);$
  - S_6: $w_3(A); r_1(A); w_1(B); r_2(B); w_2(C); r_3(C);$
  - S_7: $r_1(A); r_2(A); w_1(B); w_2(B); r_1(B); r_2(B); w_2(C); w_1(D);$
  - S_8: $r_1(A); r_2(A); r_1(B); r_2(B); r_3(A); r_4(B); w_1(A); w_2(B);$

[3], pp. 890-897

4.5. Characterizing Schedules based on Recoverability and Serializability



Relationships between schedules based on recoverability and serializability

All strict schedules are cascadeless, serializable.

All cascadeless schedules are recoverable.

4.5. Characterizing Schedules based on Serializability

- Equivalence of schedules
  - Result equivalence
    - Two schedules are called ***result equivalent*** if they produce the ***same*** final state of the database.
      - Two different schedules may **accidentally** produce the same final state.
      - Result equivalence alone cannot be used to define equivalence of schedules.
      - → The safest and most general approach to defining schedule equivalence is not to make any assumption about the types of operations included in the transactions.
      - → For two schedules to be equivalent, the operations applied to each data item affected by the schedules should be applied to that item in both schedules in *the same order*.
  - Conflict equivalence
  - View equivalence

4.5. Characterizing Schedules based on Serializability

- Equivalence of schedules
  - Result equivalence
  - Conflict equivalence
  - View equivalence
    - Another *less restrictive* definition of equivalence of schedules
    - Two schedules S and S' 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 $r_i(X)$ of T_i in S, if the value of X read by the operation has been written by an operation $w_j(X)$ of T_j (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 $r_i(X)$ of T_i in *S'*.
      - 3. If the operation $w_k(Y)$ of T_k is the last operation to write item *Y* in S, then $w_k(Y)$ *of* T_k must also be the last operation to write item *Y* in *S'*.

5.4. Characterizing transaction schedules based on serializability

- Equivalence of schedules
  - Result equivalence
  - Conflict equivalence
  - View equivalence
    - The *idea behind view equivalence* is that, 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 must produce the same results.
    - The read operations are hence said to *see the same view* in both schedules.
    - Condition 3 ensures that the final write operation on each data item is the same in both schedules, so the database state should be the same at the end of both schedules.
    - *View serializable*: a schedule S is said to be *view serializable* if it is *view equivalent* to a <u>serial</u> schedule.

4.6. Transaction Support in SQL

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

4.6. Transaction Support in SQL

Characteristics specified by a SET TRANSACTION statement in SQL:

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

4.6. Transaction Support in SQL

Characteristics specified by a SET TRANSACTION statement in SQL:

- **Isolation level** <isolation>, where <isolation> can be READ UNCOMMITTED, READ COMMITTED, REPEATABLE READ or SERIALIZABLE. The default is SERIALIZABLE.

  With SERIALIZABLE: the interleaved execution of transactions will adhere to our notion of serializability. However, if any transaction executes at a lower level, then serializability may be violated.

4.6. Transaction Support in SQL

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 T_1 may read a given value from a table. If another transaction T_2 later updates that value and T_1 reads that value again, T_1 will see a different value. Consider that T_1 reads the employee salary for Smith. Next, T_2 updates the salary for Smith. If T_1 reads Smith's salary again, then it will see a different value for Smith's salary.

4.6. Transaction Support in SQL

Potential problem with lower isolation levels:

- **Phantoms**: New rows being read using the same read with a condition.

  A transaction T_1 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 T_2 inserts a new row that also satisfies the WHERE clause condition of T_1, into the table used by T_1. If T_1 is repeated, then T_1 will see a row that previously did not exist, called a **phantom**.

4.6. Transaction Support in SQL

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:  ...
```

4.6. Transaction Support in SQL

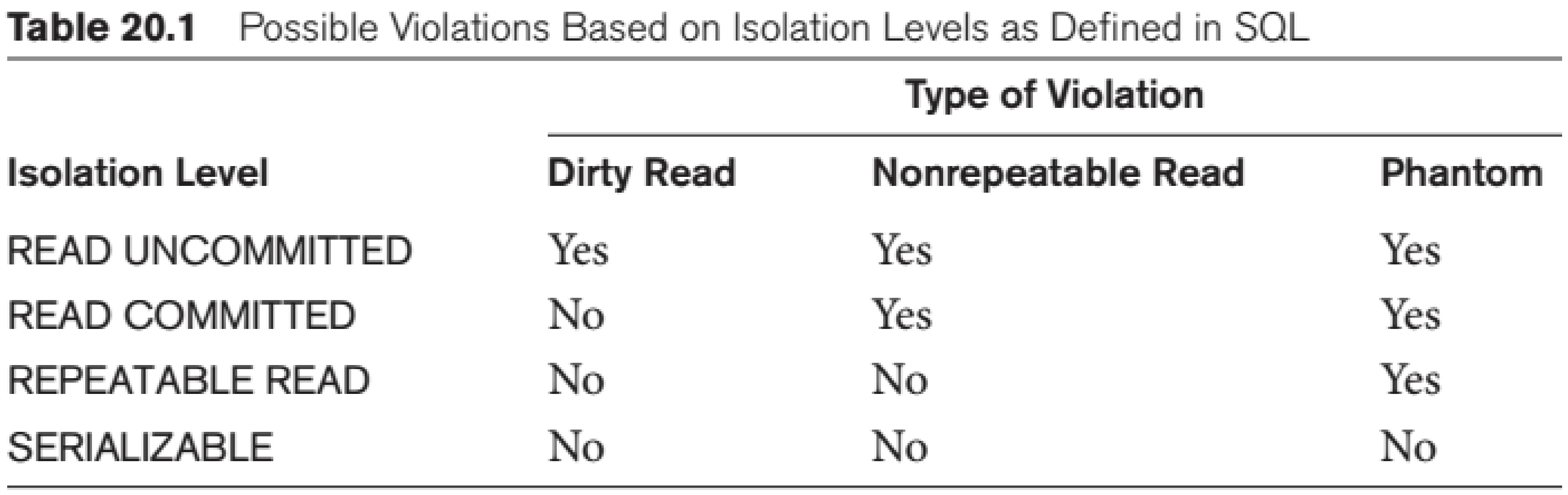


Table 20.1 Possible Violations Based on Isolation Levels as Defined in SQL

	Type of Violation		
Isolation Level	**Dirty Read**	**Nonrepeatable Read**	**Phantom**
READ UNCOMMITTED	Yes	Yes	Yes
READ COMMITTED	No	Yes	Yes
REPEATABLE READ	No	No	Yes
SERIALIZABLE	No	No	No

[1], pp. 775

Summary

- Single-user vs. multiuser systems
- Problems with uncontrolled execution of concurrency in multiuser systems:
  - Lost update, temporary read, incorrect summary, non-repeatable read, phantom
- Failures & recovery issues & the system log
- Transaction: a logical unit of database processing (read, write)
  - ACID properties
- Commit point

Summary

- Schedule (or history) as an execution sequence of the operations of several transactions with interleaving
  - characterized schedules in terms of their recoverability
    - Recoverable, Cascadeless, Strict
    - Non-recoverable
  - characterized schedules in terms of their serializability
    - Conflict-serializable
    - Precedence graph for testing its conflict-serializability
- Transaction support in SQL: isolation levels

Chapter 4: Introduction to Transaction Processing Concepts and Theory

Check for Understandings

- 4.1. Differentiate multiuser systems from single-user systems. Give their examples.
- 4.2. Describe different types of failures and give their examples.
- 4.3. What is a transaction? Give 3 examples.
- 4.4. Discuss the ACID properties of a transaction.
- 4.5. What is a system log? What records are stored in the log? Write the content of the log for your transactions in 4.3.
- 4.6. What is a commit point? Give an example.

Check for Understandings

- 4.7. What is a schedule? Give an example.
- 4.8. Given the following transactions:

T_1: $r_1(X)$; $r_1(Z)$; $w_1(X)$; c_1;

T_2: $r_2(Z)$; $r_2(Y)$; $w_2(Z)$; $w_2(Y)$; c_2;

T_3: $r_3(X)$; $r_3(Y)$; $w_3(Y)$; c_3;

What are valid schedules? Write their log contents.

S_1: $r_1(X)$; $r_1(Z)$; $r_3(X)$; $r_3(Y)$; $w_1(X)$; $w_3(Y)$; c_1; c_3;

S_2: $r_2(Z)$; $r_3(X)$; $r_2(Y)$; $r_3(Y)$; $w_2(Z)$; $w_3(Y)$; $w_2(Y)$; a_3; c_2;

S_3: $r_2(Z)$; $r_1(X)$; $r_1(Z)$; $w_2(Z)$; $w_1(X)$; $r_2(Y)$; $w_2(Y)$; c_2; c_1;

S_4: $r_1(X)$; $r_3(Y)$; $r_1(Z)$; $w_3(Y)$; $w_1(X)$; c_1; $r_3(X)$; c_3;

Check for Understandings

- 4.9. What are recoverable schedules? What are cascadeless schedules? What are strict schedules?
- 4.10. What is the recoverability characteristic of each following schedule?

  S_5: $r_1(X)$; $r_2(Z)$; $r_1(Z)$; $r_3(X)$; $r_3(Y)$; $w_1(X)$; c_1; $w_3(Y)$; c_3; $r_2(Y)$; $w_2(Z)$; $w_2(Y)$; c_2;

  S_6: $r_1(X)$; $r_2(Z)$; $r_1(Z)$; $r_3(X)$; $r_3(Y)$; $w_1(X)$; $w_3(Y)$; $r_2(Y)$; $w_2(Z)$; $w_2(Y)$; c_1; c_2; c_3;

  S_7: $r_1(X)$; $r_2(Z)$; $r_3(X)$; $r_1(Z)$; $r_2(Y)$; $r_3(Y)$; $w_1(X)$; c_1; $w_2(Z)$; $w_3(Y)$; $w_2(Y)$; c_3; c_2;

  S_8: $r_1(A)$; $r_2(B)$; $w_1(B)$; c_1; $w_2(C)$; c_2; $r_3(B)$; $r_3(C)$; $w_3(D)$; c_3;

  S_9: $r_1(A)$; $w_1(B)$; c_1; $r_2(B)$; $w_2(C)$; c_2; $r_3(C)$; $w_3(D)$; c_3;

  S_{10}: $r_2(A)$; $r_3(A)$; $r_1(A)$; $w_1(B)$; c_1; $r_2(B)$; $r_3(B)$; $w_2(C)$; c_2; $r_3(C)$; c_3;

  S_{11}: $r_2(A)$; $r_3(A)$; $r_1(A)$; $w_1(B)$; $r_3(B)$; $w_2(C)$; $r_3(C)$; a_1; a_2; a_3;

Check for Understandings

- 4.11. What are conflict-serializable schedules?
- 4.12. What is the conflict-serializable characteristic of each following schedule? Draw their precedence graphs. Determine their equivalent serial schedules.

  S_{12}: $r_1(X)$; $r_3(X)$; $w_1(X)$; $r_2(X)$; $w_3(X)$;

  S_{13}: $r_1(X)$; $r_3(X)$; $w_3(X)$; $w_1(X)$; $r_2(X)$;

  S_{14}: $r_3(X)$; $r_2(X)$; $w_3(X)$; $r_1(X)$; $w_1(X)$;

  S_{15}: $r_3(X)$; $r_2(X)$; $r_1(X)$; $w_3(X)$; $w_1(X)$;

  S_{16}: $r_1(X)$; $r_2(Z)$; $r_1(Z)$; $r_3(X)$; $r_3(Y)$; $w_1(X)$; $w_3(Y)$; $r_2(Y)$; $w_2(Z)$; $w_2(Y)$;

  S_{17}: $r_1(X)$; $r_2(Z)$; $r_3(X)$; $r_1(Z)$; $r_2(Y)$; $r_3(Y)$; $w_1(X)$; $w_2(Z)$; $w_3(Y)$; $w_2(Y)$;

  S_{18}: $r_2(X)$; $r_1(X)$; $r_3(Y)$; $w_3(Y)$; $r_1(Y)$; $w_2(X)$; $w_1(X)$; $w_1(Y)$;

  S_{19}: $r_1(Z)$; $r_1(X)$: $r_2(Y)$; $w_2(Y)$; $r_3(X)$; $r_2(Z)$; $w_1(Z)$; $w_1(X)$; $r_2(X)$; $w_3(X)$; $w_2(X)$;

  S_{20}: $r_1(X)$; $w_1(X)$; $r_2(Y)$; $r_2(X)$; $w_2(Y)$; $r_1(Y)$; $w_2(X)$; $r_3(Y)$; $w_1(Y)$; $r_3(X)$; $w_3(X)$; $w_2(Y)$; $w_3(Y)$;