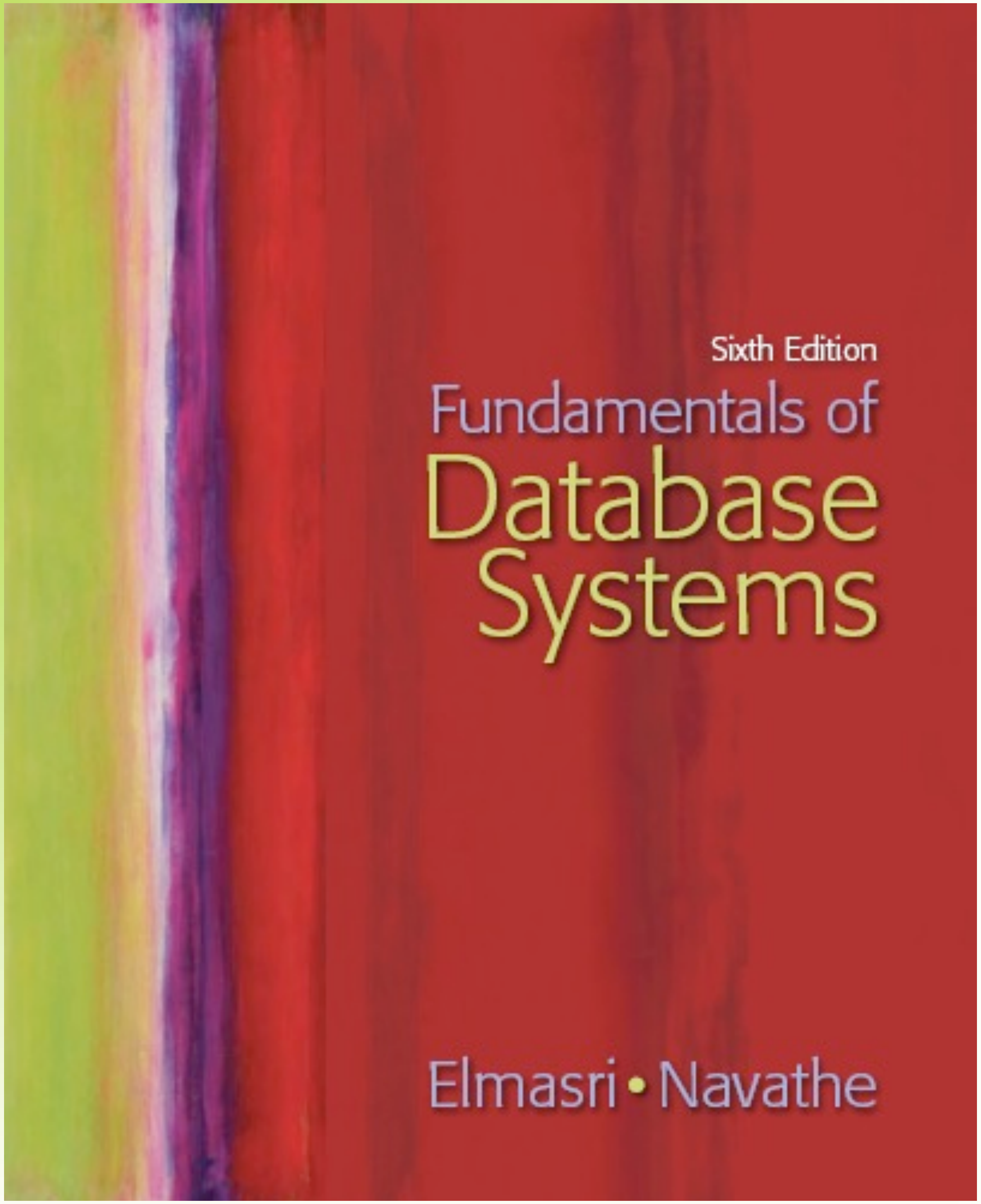
Chapter 21

Introduction to Transaction Processing Concepts and Theory



Chapter 21

Introduction to Transaction Processing Concepts and Theory

Addison-Wesley is an imprint of PEARSON

Chapter 21 Outline

1 Introduction to Transaction Processing

2 Transaction and System Concepts

3 Desirable Properties of Transactions

4 Characterizing Schedules based on Recoverability

5 Characterizing Schedules based on Serializability

6 Transaction Support in SQL

Introduction to Transaction Processing

- **Transaction:** An executing program (process) that includes one or more database access operations
  - Read operations (database retrieval, such as SQL SELECT)
  - Write operations (modify database, such as SQL INSERT, UPDATE, DELETE)
  - Transaction: A logical unit of database processing
  - Example: Bank balance transfer of $100 dollars from a checking account to a saving account in a BANK database
- **Note:** Each execution of a program is a *distinct transaction* with different parameters
  - Bank transfer program parameters: savings account number, checking account number, transfer amount

Introduction to Transaction Processing (cont.)

- A transaction (set of operations) may be:
  - stand-alone, specified in a high level language like SQL submitted interactively, or
  - consist of database operations embedded within a program (most transactions)
- **Transaction boundaries**: Begin and End transaction.
  - Note: An **application program** may contain several transactions separated by Begin and End transaction boundaries

Introduction to Transaction Processing (cont.)

- **Transaction Processing Systems:** Large multi-user database systems supporting thousands of *concurrent transactions* (user processes) per minute
- **Two Modes of Concurrency**
  - **Interleaved processing**: concurrent execution of processes is interleaved in a single CPU
  - **Parallel processing**: processes are concurrently executed in multiple CPUs (Figure 21.1)
  - Basic transaction processing theory assumes interleaved concurrency

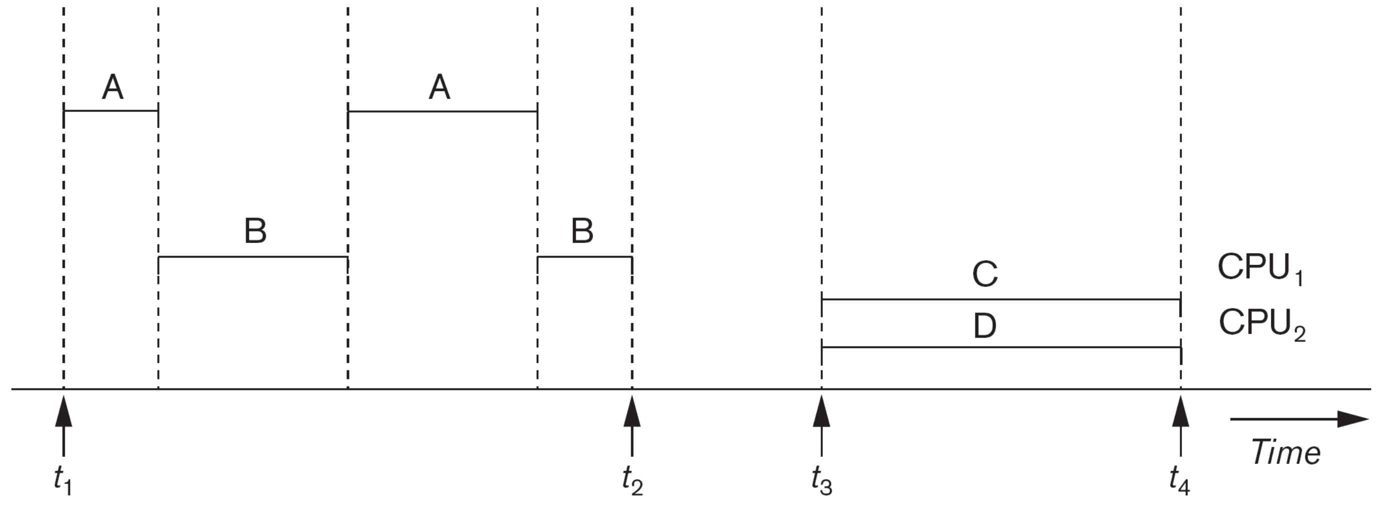


Figure 21.1
Interleaved processing versus parallel processing of concurrent transactions.

Introduction to Transaction Processing (cont.)

For transaction processing purposes, a simple database model is used:

- **A database -** collection of named data items
- **Granularity (size) of a data item** - a field (data item value), a record, or a whole disk block
  - TP concepts are independent of granularity
- Basic operations on an item X:
  - **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 (cont.)

READ AND WRITE OPERATIONS:

- Basic unit of data transfer from the disk to the computer main memory is one disk block (or page). A data item X (what is read or written) will usually be the field of some record in the database, although it may be a larger unit such as a whole record or even a whole block.
- **read_item(X) command includes the following steps:**
  - Find the address of the disk block that contains item X.
  - Copy that disk block into a buffer in main memory (if that disk block is not already in some main memory buffer).
  - Copy item X from the buffer to the program variable named X.

Introduction to Transaction Processing (cont.)

READ AND WRITE OPERATIONS (cont.):

- **write_item(X) command includes the following steps:**
- Find the address of the disk block that contains item X.
- Copy that disk block into a buffer in main memory (if it is not already in some main memory buffer).
- Copy item X from the program variable named X into its correct location in the buffer.
- Store the updated block from the buffer back to disk (either immediately or at some later point in time).

Transaction Notation

- Figure 21.2 (next slide) shows two examples of transactions
- Notation focuses on the read and write operations
- Can also write in shorthand notation:
  - T1: b1; r1(X); w1(X); r1(Y); w1(Y); e1;
  - T2: b2; r2(Y); w2(Y); e2;
- bi and ei specify transaction boundaries (begin and end)
- i specifies a unique transaction identifier (TId)

(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);

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

Why we need concurrency control

Without Concurrency Control, problems may occur with concurrent transactions:

- **Lost Update Problem.**

  Occurs when two transactions update the same data item, but both read the same original value before update (Figure 21.3(a), next slide)

- **The Temporary Update (or Dirty Read) Problem.**

  This occurs when one transaction T1 updates a database item X, which is accessed (read) by another transaction T2; then T1 fails for some reason (Figure 21.3(b)); X was (read) by T2 before its value is changed back (rolled back or UNDONE) after T1 fails

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.

(a)

Time

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

write_item(X);
read_item(Y);	

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

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

(b)

Time

T_1	T_2

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

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

read_item(Y);	

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

Why we need concurrency control (cont.)

- **The Incorrect Summary Problem .**

  One transaction is calculating an aggregate summary function on a number of records (for example, sum (total) of all bank account balances) while other transactions are updating some of these records (for example, transferring a large amount between two accounts, see Figure 21.3(c)); the aggregate function may read <u>some values before they are updated and others after they are updated</u>.

(c)

T_1	T_3
	sum := 0;
	read_item(A);
	sum := *sum* + A;
	⋮
read_item(X);	
$X := X - N$;	
write_item(X);	
	read_item(X);
	sum := *sum* + X;
	read_item(Y);
	sum := *sum* + Y;
read_item(Y);	
$Y := Y + N$;	
write_item(Y);	

← T_3 reads X after N is subtracted and reads Y before N is added; a wrong summary is the result (off by N).

Why we need concurrency control (cont.)

- **The Unrepeatable Read Problem .**

  A transaction T1 may read an item (say, available seats on a flight); later, T1 may read the same item again and get a different value because another transaction T2 has updated the item (reserved seats on the flight) between the two reads by T1

Why recovery is needed

Causes of transaction failure:

1. **A computer failure (system crash):** A hardware or software error occurs during transaction execution. If the hardware crashes, the contents of the computer's internal main memory may be lost.
2. **A transaction or system error :** Some operation in the transaction may cause it to fail, such as integer overflow or division by zero. Transaction failure may also occur because of erroneous parameter values or because of a logical programming error. In addition, the user may interrupt the transaction during its execution.

Why recovery is needed (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, to be canceled
   - a programmed abort causes the transaction 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 22).

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 kind of failure and item 6 are more severe than items 1 through 4.
6. **Physical problems and catastrophes:** This refers to an endless list of problems that includes power or air-conditioning failure, fire, theft, sabotage, overwriting disks or tapes by mistake, and mounting of a wrong tape by the operator.

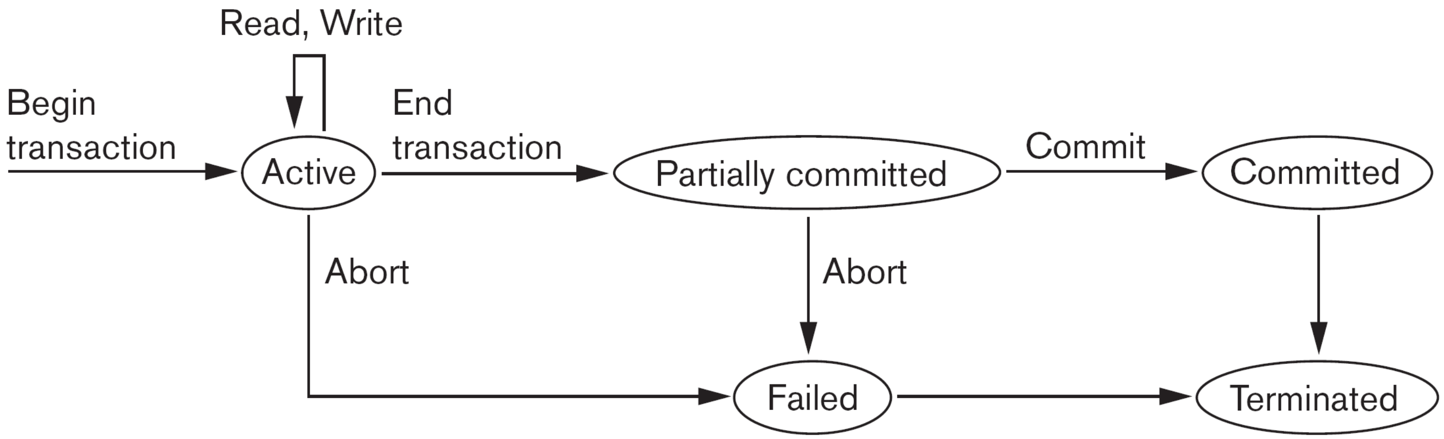
Transaction and System Concepts

A **transaction** is an atomic unit of work that is either completed in its entirety or not done at all. A transaction passes through several states (Figure 21.4, similar to process states in operating systems).

Transaction states:

- Active state (executing read, write operations)
- Partially committed state (ended but waiting for system checks to determine success or failure)
- Committed state (transaction succeeded)
- Failed state (transaction failed, must be rolled back)
- Terminated State (transaction leaves system)

Figure 21.4
State transition diagram illustrating the states for transaction execution.



Transaction and System Concepts (cont.)

DBMS Recovery Manager needs system to keep track of the following operations (in the system **log file**):

- **begin_transaction:** Start of transaction execution.
- **read or write:** Read or write operations on the database items that are executed as part of a transaction.
- **end_transaction:** Specifies end of read and write transaction operations have ended. System may still have to check whether the changes (writes) introduced by transaction can be *permanently applied to the database* (**commit** transaction); or whether the transaction has to be *rolled back* (**abort** transaction) because it violates concurrency control or for some other reason.

Transaction and System Concepts (cont.)

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

- **commit_transaction:** Signals *successful end* of transaction; any changes (writes) executed by transaction can be safely **committed** to the database and will not be undone.
- **abort_transaction (or rollback):** Signals transaction has *ended unsuccessfully*; any changes or effects that the transaction may have applied to the database must be *undone.*

Transaction and System Concepts (cont.)

System operations used during recovery (see Chapter 23):

- **undo(X):** Similar to rollback except that it applies to a single write operation rather than to a whole transaction.
- **redo(X):** This specifies that a *write operation* of a committed transaction must be *redone* to ensure that it has been applied permanently to the database on disk.

Transaction and System Concepts (cont.)

The System Log File

- Is an *append-only file* to keep track of all operations of all transactions *in the order in which they occurred*. This information is needed during recovery from failures
- Log is kept on disk - not affected except for disk or catastrophic failure
- As with other disk files, a *log main memory buffer* is kept for holding the records being appended until the whole buffer is appended to the end of the log file on disk
- Log is periodically backed up to archival storage (tape) to guard against catastrophic failures

Transaction and System Concepts (cont.)

Types of records (entries) in log file:

- [start_transaction,T]: Records that transaction T has started execution.
- [write_item,T,X,old_value,new_value]: T has changed the value of item X from old_value to new_value.
- [read_item,T,X]: T has read the value of item X (not needed in many cases).
- [end_transaction,T]: T has ended execution
- [commit,T]: T has completed successfully, and committed.
- [abort,T]: T has been aborted.

Transaction and System Concepts (cont.)

The System Log (cont.):

- protocols for recovery that <u>avoid cascading rollbacks do not require that read operations be written to the system log;</u> most recovery protocols fall in this category (see Chapter 23)
- strict protocols require simpler write entries that do not include new_value (see Section 21.4).

Transaction and System Concepts (cont.)

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 file (on disk). The transaction is then said to be **committed**.

Transaction and System Concepts (cont.)

Commit Point of a Transaction (cont.):

- **Log file buffers:** Like database files on disk, whole disk blocks must be read or written to main memory buffers.
- For **log file**, the last disk block (or blocks) of the file will be in main memory buffers to easily append log entries at end of file.
- **Force writing the log buffer:** *before* a transaction reaches its commit point, any main memory buffers of the log that have not been written to disk yet must be copied to disk.
- Called **force-writing** the log buffers before committing a transaction.
- Needed to ensure that any write operations by the transaction are recorded in the log file *on disk* before the transaction commits

Desirable Properties of Transactions

Called ACID properties – Atomicity, Consistency, Isolation, Durability:

- **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 (cont.)

ACID properties (cont.):

- **Isolation**: Even though transactions are executing concurrently, they should appear to be executed in isolation – that is, their final effect should be as if each transaction was executed in isolation from start to finish.

- **Durability or permanency**: Once a transaction is committed, its changes (writes) applied to the database must never be lost because of subsequent failure.

Desirable Properties of Transactions (cont.)

- **Atomicity**: Enforced by the recovery protocol.
- **Consistency preservation**: Specifies that each transaction does a correct action on the database *on its own*. Application programmers and DBMS constraint enforcement are responsible for this.
- **Isolation**: Responsibility of the concurrency control protocol.
- **Durability or permanency**: Enforced by the recovery protocol.

Schedules of Transactions

- **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).
- Figure 21.5 (next slide) shows 4 possible schedules (A, B, C, D) of two transactions T1 and T2:
  - Order of operations from top to bottom
  - Each schedule includes *same operations*
  - Different *order of operations* in each schedule

Figure 21.5
Examples of serial and nonserial schedules involving transactions T_1 and T_2. (a) Serial schedule A: T_1 followed by T_2. (b) Serial schedule B: T_2 followed by T_1. (c) Two nonserial schedules C and D with interleaving of operations.

(a)

Time ↓

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

Schedule A

(b)

Time ↓

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

Schedule B

(c)

Time ↓

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

Schedule C

Time ↓

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

Schedule D

Schedules of Transactions (cont.)

- Schedules can also be displayed in more compact notation
- Order of operations from left to right
- Include only read (r) and write (w) operations, with transaction id (1, 2, …) and item name (X, Y, …)
- Can also include other operations such as b (begin), e (end), c (commit), a (abort)
- Schedules in Figure 21.5 would be displayed as follows:
  - Schedule A: r1(X); w1(X); r1(Y); w1(Y); r2(X); w2(x);
  - Schedule B: r2(X); w2(X); r1(X); w1(X); r1(Y); w1(Y);
  - Schedule C: r1(X); r2(X); w1(X); r1(Y); w2(X); w1(Y);
  - Schedule D: r1(X); w1(X); r2(X); w2(X); r1(Y); w1(Y);

Schedules of Transactions (cont.)

- Formal definition of a **schedule** (or **history**) S of n transactions T1, T2, ..., Tn :

  An ordering of all the operations of the transactions subject to the constraint that, for each transaction Ti that participates in S, the operations of Ti in S must appear *in the same order* in which they occur in Ti.

Note: Operations from other transactions Tj <u>can be interleaved</u> with the operations of Ti in S.

Schedules of Transactions (cont.)

- For n transactions T1, T2, ..., Tn, where each Ti has mi read and write operations, the number of possible schedules is (! is *factorial* function):

  (m1 + m2 + … + mn)! / ((m1)! * (m2)! * … * (mn)!)

- Generally very large number of possible schedules
- Some schedules are easy to recover from after a failure, while others are not
- Some schedules produce correct results, while others produce incorrect results
- Rest of chapter characterizes schedules by classifying them based on ease of recovery (**recoverability**) and correctness (**serializability**)

Characterizing Schedules based on Recoverability

Schedules classified into two main classes:

- **Recoverable schedule:** One where no *committed* transaction needs to be rolled back (aborted).

  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.

- **Non-recoverable schedule:** A schedule where a committed transaction may have to be rolled back during recovery.

  This violates **Durability** from ACID properties (a committed transaction cannot be rolled back) and so non-recoverable schedules *should not be allowed*.

Characterizing Schedules Based on Recoverability (cont.)

- **<u>Example:</u>** Schedule A below is **non-recoverable** because T2 reads the value of X that was written by T1, but then T2 commits before T1 commits or aborts
- To make it **recoverable**, the commit of T2 (c2) must be delayed until T1 either commits, or aborts (Schedule B)
- If T1 commits, T2 can commit
- If T1 aborts, T2 must also abort because it read a value that was written by T1; this value must be undone (reset to its old value) when T1 is aborted
  - known as *cascading rollback*

- Schedule A: r1(X); <u>w1(X)</u>; <u>r2(X)</u>; w2(X); <u>c2</u>; r1(Y); w1(Y); c1 (or a1)
- Schedule B: r1(X); w1(X); r2(X); w2(X); r1(Y); w1(Y); c1 (or a1); ...

Characterizing Schedules based on Recoverability (cont.)

Recoverable schedules can be further refined:

- **Cascadeless schedule:** A schedule in which a transaction T2 cannot read an item X until the transaction T1 that last wrote X has committed.
- The set of cascadeless schedules is a *subset of* the set of recoverable schedules.

Schedules requiring cascaded rollback: A schedule in which an uncommitted transaction T2 that read an item that was written by a failed transaction T1 must be rolled back.

Characterizing Schedules Based on Recoverability (cont.)

- **Example:** Schedule B below is **not cascadeless** because T2 reads the value of X that was written by T1 before T1 commits
- If T1 aborts (fails), T2 must also be aborted (rolled back) resulting in *cascading rollback*
- To make it **cascadeless**, the r2(X) of T2 must be delayed until T1 commits (or aborts and rolls back the value of X to its previous value) – see Schedule C

- Schedule B: r1(X); w1(X); r2(X); w2(X); r1(Y); w1(Y); c1 (or a1);
- Schedule C: r1(X); w1(X); r1(Y); w1(Y); c1; r2(X); w2(X); ...

Characterizing Schedules based on Recoverability (cont.)

Cascadeless schedules can be further refined:

- **Strict schedule:** A schedule in which a transaction T2 can neither read *nor write* an item X until the transaction T1 that last wrote X has committed.
- The set of strict schedules is a *subset of* the set of cascadeless schedules.
- If *blind writes* are not allowed, all cascadeless schedules are also strict

Blind write: A write operation w2(X) that is not preceded by a read r2(X).

Characterizing Schedules Based on Recoverability (cont.)

- **Example:** Schedule C below is **cascadeless** and also **strict** (because it has no blind writes)
- Schedule D is cascadeless, but not strict (because of the blind write w3(X), which writes the value of X before T1 commits)
- To make it strict, w3(X) must be delayed until after T1 commits – see Schedule E

- Schedule C: r1(X); w1(X); r1(Y); w1(Y); c1; r2(X); w2(X); …
- Schedule D: r1(X); w1(X); w3(X); r1(Y); w1(Y); c1; r2(X); w2(X); …
- Schedule E: r1(X); w1(X); r1(Y); w1(Y); c1; w3(X); r2(X); w2(X); …

Characterizing Schedules Based on Recoverability (cont.)

Summary:

- Many schedules can exist for a set of transactions
- The set of all possible schedules can be partitioned into two subsets: **recoverable** and **non-recoverable**
- A subset of the recoverable schedules are **cascadeless**
- If blind writes are allowed, a subset of the cascadeless schedules are **strict**
- If *blind writes are not allowed*, the set of cascadeless schedules is the same as the set of strict schedules

Characterizing Schedules based on Serializability

- Among the large set of possible schedules, we want to characterize which schedules are *guaranteed to give a correct result*
- The **consistency preservation** property of the ACID properties states that: each transaction if executed on its own (from start to finish) will transform a consistent state of the database into another consistent state
- Hence, each transaction is *correct* on its own

Characterizing Schedules based on Serializability (cont.)

- **Serial schedule**: A schedule S is **serial** if, for every transaction T participating in the schedule, all the operations of T are executed consecutively (without interleaving of operations from other transactions) in the schedule. Otherwise, the schedule is called **nonserial.**
- Based on the consistency preservation property, *any serial schedule will produce a correct result* (assuming no inter-dependencies among different transactions)

Characterizing Schedules based on Serializability (cont.)

- Serial schedules are *not feasible* for performance reasons:
  - No interleaving of operations
  - Long transactions force other transactions to wait
  - System cannot switch to other transaction when a transaction is waiting for disk I/O or any other event
  - Need to allow concurrency with interleaving without sacrificing correctness

Characterizing Schedules based on Serializability (cont.)

- **Serializable schedule**: A schedule S is **serializable** if it is **equivalent** to some serial schedule of the same n transactions.
- There are (n)! serial schedules for n transactions – a serializable schedule can be equivalent to *any of the serial schedules*
- **Question:** How do we define equivalence of schedules?

Equivalence of Schedules

- **Result equivalent**: Two schedules are called result equivalent if they produce the same final state of the database.
- Difficult to determine without *analyzing the internal operations of the transactions*, which is not feasible in general.
- May also get result equivalence *by chance* for a particular input parameter even though schedules *are not equivalent in general* (see Figure 21.6, next slide)

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

S_1	S_2
read_item(X);	read_item(X);
$X := X + 10$;	$X := X * 1.1$;
write_item(X);	write_item (X);

Equivalence of Schedules (cont.)

- **Conflict equivalent**: Two schedules are conflict equivalent if the relative order of *any two conflicting operations* is the same in both schedules.
- Commonly used definition of schedule equivalence
- Two operations are **conflicting** if:
  - They access the same data item X
  - They are from two different transactions
  - At least one is a write operation
- Read-Write conflict example: r1(X) and w2(X)
- Write-write conflict example: w1(Y) and w2(Y)

Equivalence of Schedules (cont.)

- Changing the order of conflicting operations generally *causes a different outcome*
- **Example:** changing r1(X); w2(X) to w2(X); r1(X) means that T1 will read *a different value for X*
- **Example:** changing w1(Y); w2(Y) to w2(Y); w1(Y) means that the final value for Y in the database can be different
- Note that read operations are **not conflicting**; changing r1(Z); r2(Z) to r2(Z); r1(Z) does not change the outcome

Characterizing Scedules Based on Serializability (cont.)

- **Conflict equivalence** of schedules is used to determine which schedules are correct in general (serializable)

A schedule S is said to be **serializable** if it is conflict equivalent to some serial schedule S’.

Characterizing Schedules based on Serializability (cont.)

- A serializable schedule is considered to be correct because it is equivalent to a serial schedule, and any serial schedule is considered to be correct
  - 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 and interleaving of operations from different transactions.

Characterizing Schedules based on Serializability (cont.)

- Serializability is generally hard to check at run-time:
  - Interleaving of operations is generally handled by the operating system through the process scheduler
  - Difficult to determine beforehand how the operations in a schedule will be interleaved
  - Transactions are continuously started and terminated

Characterizing Schedules Based on Serializability (cont.)

Practical approach:

- Come up with methods (concurrency control protocols) to ensure serializability (discussed in Chapter 22)
- DBMS concurrency control subsystem will enforce the protocol rules and thus guarantee serializability of schedules
- Current approach used in most DBMSs:
  - Use of locks with two phase locking (see Section 22.1)

Characterizing Schedules based on Serializability (cont.)

Testing for conflict serializability

Algorithm 21.1:

- Looks at only r(X) and w(X) operations in a schedule
- Constructs a precedence graph (serialization graph) – **one node for each transaction**, plus directed edges
- 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
- The schedule is serializable if and only if the precedence graph **has no cycles**.

Algorithm 21.1. 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.

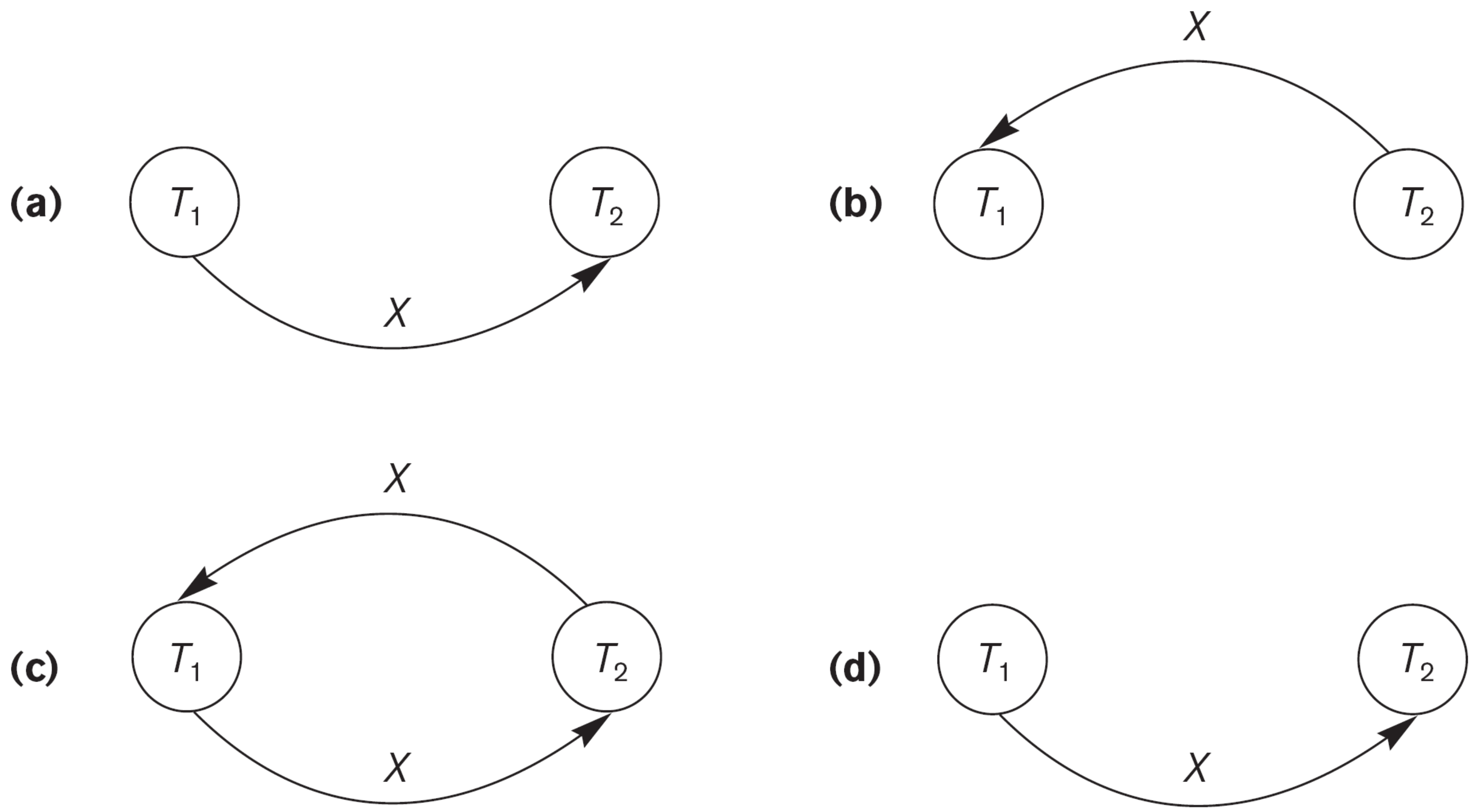


Figure 21.7
Constructing the precedence graphs for schedules A to D from Figure 21.5 to test for conflict serializability. (a) Precedence graph for serial schedule A. (b) Precedence graph for serial schedule B. (c) Precedence graph for schedule C (not serializable). (d) Precedence graph for schedule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

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.

(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);

(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

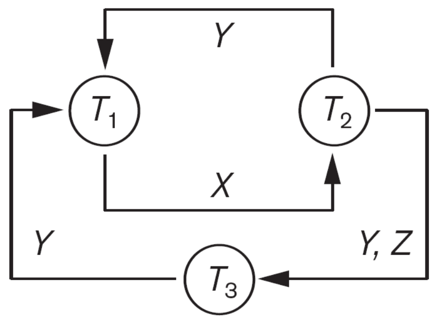
(c)

Time ↓

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

Schedule F

(d)



Equivalent serial schedules

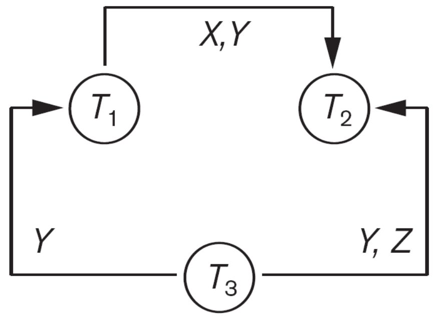
None

Reason

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

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

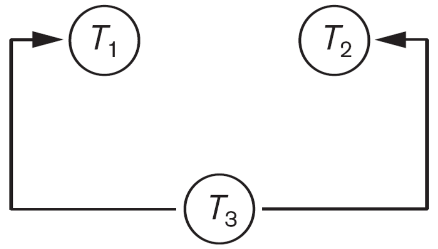
(e)



Equivalent serial schedules

$T_3 \rightarrow T_1 \rightarrow T_2$

(f)



Equivalent serial schedules

$T_3 \rightarrow T_1 \rightarrow T_2$

$T_3 \rightarrow T_2 \rightarrow T_1$

Figure 21.8 (continued)
Another example of serializability testing.
(d) Precedence graph for schedule E.
(e) Precedence graph for schedule F.
(f) Precedence graph with two equivalent serial schedules.

Characterizing Schedules based on Serializability (cont.)

- **View equivalence**: A less restrictive definition of equivalence of schedules than conflict serializability *when blind writes are allowed*
- **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 (cont.)

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

- The same set of transactions participates in S and S', and S and S' include the same operations of those transactions.
- For any operation Ri(X) of Ti in S, if the value of X read was 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'.
- 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 (cont.)

The premise behind view equivalence:

- Each read operation of a transaction reads the result of *the same write operation* in both schedules.
- **"The view"**: the read operations are said to see the *the same view* in both schedules.
- The final write operation on each item is the same on both schedules resulting in the same final database state in case of blind writes

Characterizing Schedules based on Serializability (cont.)

Relationship between view and conflict equivalence:

- The two are same under **constrained write assumption** (no blind writes allowed)
- Conflict serializability is **stricter** than view serializability when **blind writes occur** (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 (cont.)

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 T2 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 (cont.)

Other Types of Equivalence of Schedules

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

Characterizing Schedules based on Serializability (cont.)

Other Types of Equivalence of Schedules (cont.)

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

Consider the following schedule S for the two transactions:

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

Using conflict serializability, it is not **serializable.**

However, if it came from a (read,update, write) sequence as follows:

r1(X); X := X – 10; w1(X); r2(Y); Y := Y – 20; w2(Y); r1(Y);

Y := Y + 10; w1(Y); r2(X); X := X + 20; w2(X);

Sequence explanation: debit, debit, credit, credit.

It is a **correct schedule <u>for the given semantics</u>**

Introduction to 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.

Introduction to Transaction Support in SQL (cont.)

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 UNCOMITTED is specified, in which case READ ONLY is assumed.
- **Diagnostic size** n, specifies an integer value n, indicating the number of conditions that can be held simultaneously in the diagnostic area. (To supply run-time feedback information to calling program for SQL statements executed in program)

Transaction Support in SQL (cont.)

Characteristics specified by a SET TRANSACTION statement in SQL (cont.):

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

  If all transactions is a schedule specify isolation level SERIALIZABLE, the interleaved execution of transactions will adhere to serializability. However, if any transaction in the schedule executes at a lower level, serializability may be violated.

Transaction Support in SQL (cont.)

Potential problem with lower isolation levels:

- **Dirty Read**: Reading a value that was written by a transaction that 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 then T1 reads that value again, T1 will see a different value. Example: T1 reads the No. of seats on a flight. Next, T2 updates that number (by reserving some seats). If T1 reads the No. of seats again, it will see a different value.

Transaction Support in SQL (cont.)

Potential problem with lower isolation levels (cont.):

- **Phantoms**: New row inserted after another transaction accessing that row was started.

  A transaction T1 may read a set of rows from a table (say EMP), based on some condition specified in the SQL WHERE clause (say DNO=5). Suppose a transaction T2 inserts a new EMP row whose DNO value is 5. T1 should see the new row (if equivalent serial order is T2; T1) or not see it (if T1; T2). The record that did not exist when T1 started is called a **phantom record**.

Transaction Support in SQL2 (cont.)

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;
GO TO  THE_END;
 UNDO: EXEC SQL ROLLBACK;
THE_END:  ...
```

Table 21.1 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

Chapter 21 Summary

- Introduction to Transaction Processing
- Transaction and System Concepts
- Desirable Properties of Transactions (ACID properties)
- Characterizing Schedules based on Recoverability
- Characterizing Schedules based on Serializability
- Transaction Support in SQL