MODULE – 5

TRANSACTION PROCESSING

INTRODUCTION TO TRANSACTION PROCESSING

CONCURRENCY CONTROL IN DATABASES

INTRODUCTION TO DATABASE RECOVERY PROTOCOLS

Mr. C. R. Belavi, Dept. of CSE, HIT, NDS

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

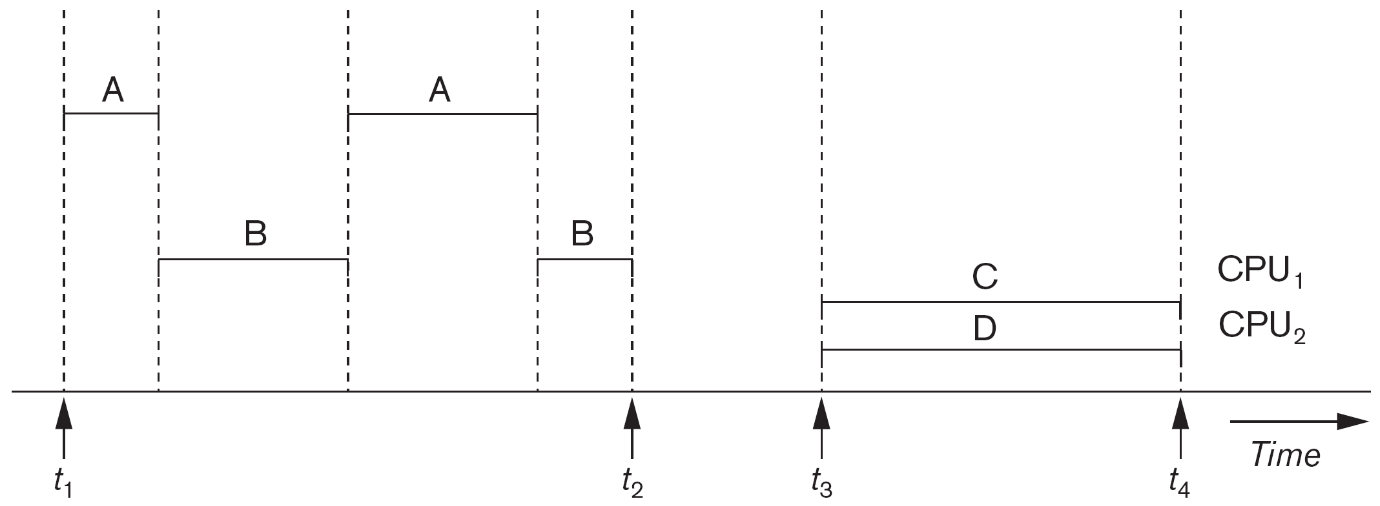


Figure 21.1
Interleaved processing versus parallel processing of concurrent transactions.

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 <u>one disk block (or page)</u>. 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.

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:

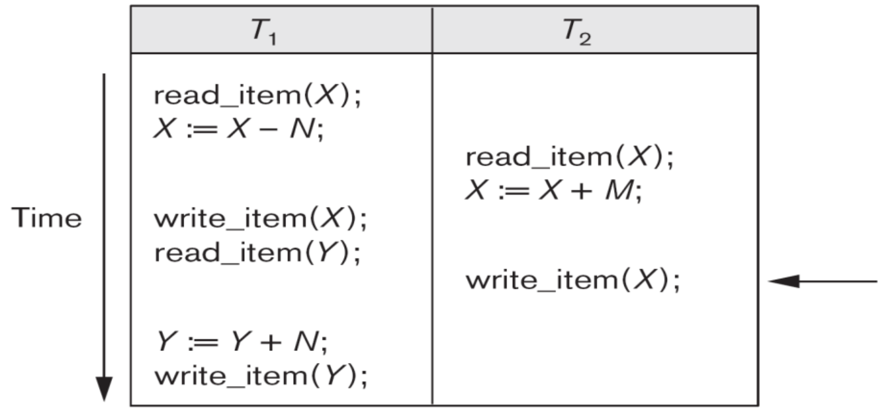
- **Lost Update Problem.**

  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

(a)



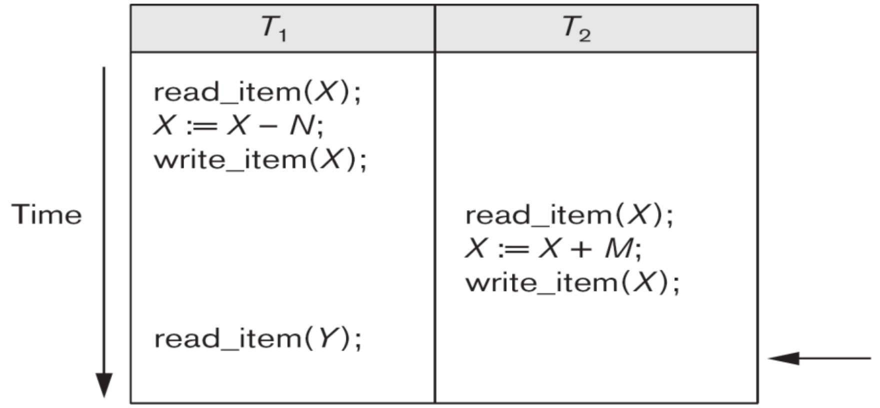
T_1	T_2
read_item(X); $X := X - N$;	
	read_item(X); $X := X + M$;

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)



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.

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.

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 some values before they are updated and others after they are updated.

(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.

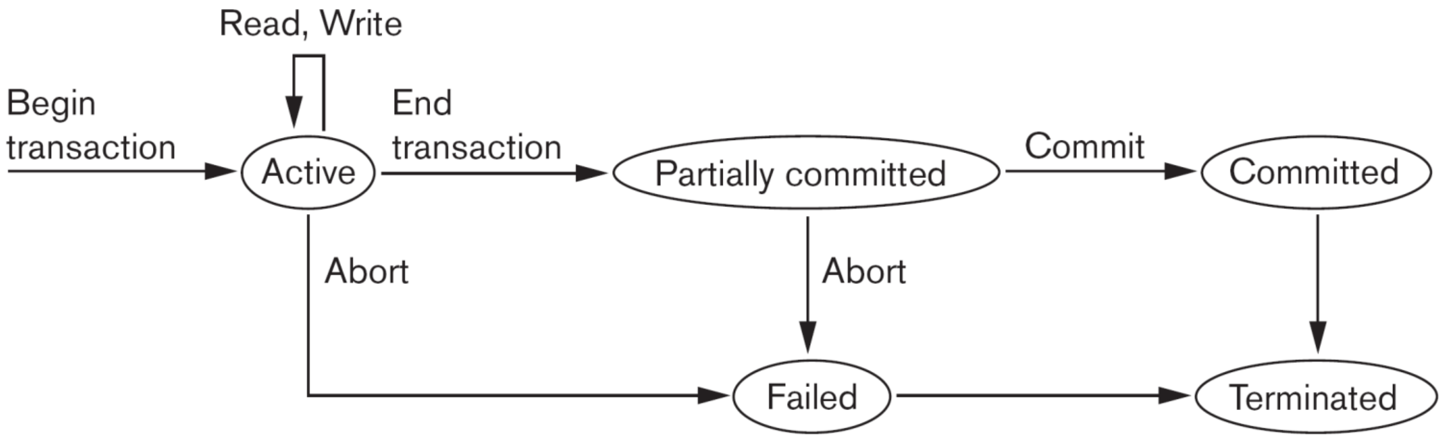
Transaction and System Concepts

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 avoid cascading rollbacks do not require that read operations be written to the system log; 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):

  $$(m1 + m2 + \ldots + mn)! \,/\, (\,(m1)! * (m2)! * \ldots * (mn)!\,)$$

- 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.)

- **Example:** 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); <u>w3(X);</u> r1(Y); w1(Y); c1; r2(X); w2(X); …
- Schedule E: r1(X); w1(X); r1(Y); w1(Y); c1; <u>w3(X);</u> 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

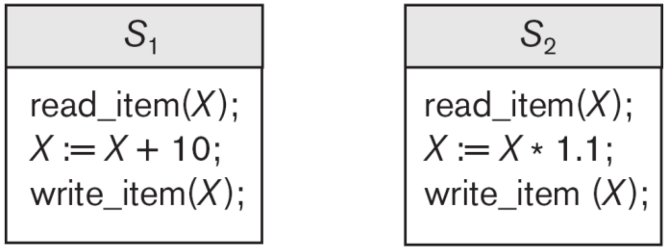
Characterizing Schedules based on Serializability (cont.)

- **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

```
read_item(X);
X := X + 10;
write_item(X);
```

S_2

```
read_item(X);
X := X * 1.1;
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 <u>considered to be correct</u> 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.

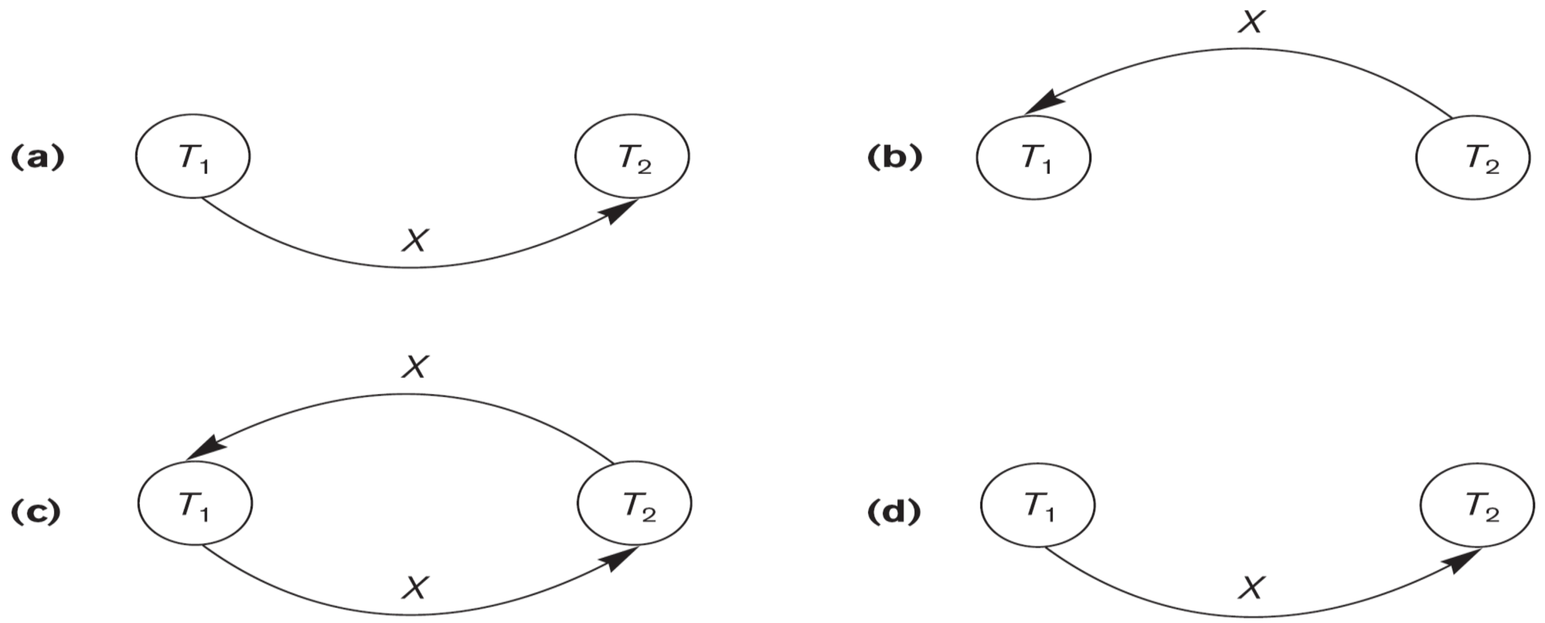


Figure 21.7
Constructing the precedence graphs for schedules A to D from Figure 21.5 to test for conflict serializability. (a) Precedence graph for serial schedule A. (b) Precedence graph for serial schedule B. (c) Precedence graph for schedule C (not serializable). (d) Precedence graph for schedule 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	Transaction T_2	Transaction T_3
read_item(X);	read_item(Z);	read_item(Y);
write_item(X);	read_item(Y);	read_item(Z);
read_item(Y);	write_item(Y);	write_item(Y);
write_item(Y);	read_item(X);	write_item(Z);
	write_item(X);	

(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

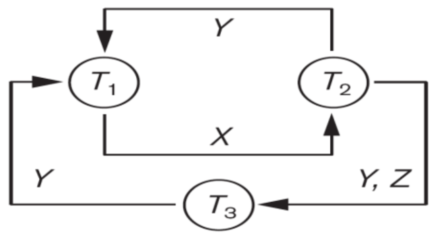
(c)

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)



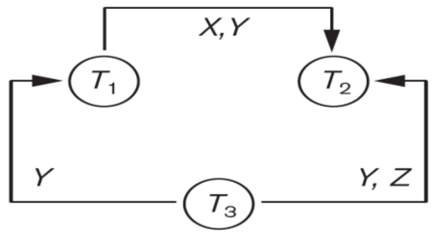
Equivalent serial schedules

None

Reason

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

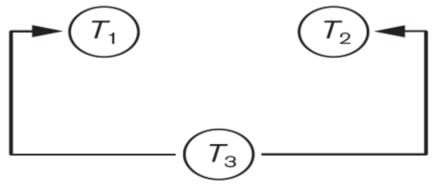
(e)



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

Purpose of Concurrency Control

- To ensure that the Isolation Property is maintained while allowing transactions to execute concurrently (outcome of concurrent transactions *should appear as though they were executed in isolation*).
- To preserve database consistency by ensuring that the schedules of executing transactions are serializable.
- To resolve read-write and write-write conflicts among transactions.

Concurrency Control Protocols (CCPs)

- A CCP is a set of rules enforced by the DBMS to ensure serializable schedules
- Also known as CCMs (Concurrency Control Methods)
- Main protocol known as 2PL (2-phase locking), which is based on *locking the data items*
- Other protocols use different techniques
- We first cover 2PL in some detail, then give an overview of other techniques

2PL Concurrency Control Protocol

Based on each transaction securing a **lock** on a data item before using it:

Locking enforces **mutual exclusion** when accessing a data item – simplest kind of lock is a **binary lock**, which secures permission to Read or Write a data item for a transaction.

Example: If T1 requests Lock(X) operation, the system grants the lock *unless item X is already locked by another transaction*. If request is granted, data item X is locked on behalf of the requesting transaction T1.

Unlocking operation removes the lock.

Example: If T1 issues Unlock (X), data item X is made available to all other transactions.

2PL Concurrency Control Protocol

System maintains **lock table** to keep track of which items are locked by which transactions

Lock(X) and Unlock (X) are hence **system calls**

Transactions that request a lock but do not have it granted can be placed on a **waiting queue** for the item

Transaction T must unlock any items it had locked before T terminates

Next slides gives overview of lock and unlock operations

```
lock_item(X):
B:  if LOCK(X) = 0                    (* item is unlocked *)
        then LOCK(X) ← 1              (* lock the item *)
    else
        begin
        wait (until LOCK(X) = 0
            and the lock manager wakes up the transaction);
        go to B
        end;
unlock_item(X):
    LOCK(X) ← 0;                      (* unlock the item *)
    if any transactions are waiting
        then wakeup one of the waiting transactions;
```

Figure 22.1
Lock and unlock operations for binary locks.

2PL Concurrency Control (cont.)

Locking for database items:

For database purposes, *binary locks are not sufficient*:

- Two locks modes are needed (a) **shared lock** (read lock) and (b) **exclusive lock** (write lock).

Shared mode: Read lock (X). Several transactions can hold shared lock on X (because read operations are not conflicting).

Exclusive mode: Write lock (X). Only one write lock on X can exist at any time on an item X. (No read or write locks on X by other transactions can exist).

Conflict matrix

	Read	Write
Read	Y	N
Write	N	N

2PL Concurrency Control (cont.)

Three operations are now needed:

- read_lock(X): transaction T requests a read (shared) lock on item X
- write_lock(X): transaction T requests a write (exclusive) lock on item X
- unlock(X): transaction T unlocks an item that it holds a lock on (shared or exclusive)

Transaction can be blocked (forced to wait) if the item is held by other transactions **in conflicting lock mode**

Conflicts are write-write or read-write (read-read is not conflicting)

Next slide gives outline of these three operations

```
read_lock(X):
B:  if LOCK(X) = "unlocked"
        then begin LOCK(X) ← "read-locked";
                no_of_reads(X) ← 1
                end
    else if LOCK(X) = "read-locked"
        then no_of_reads(X) ← no_of_reads(X) + 1
    else begin
            wait (until LOCK(X) = "unlocked"
                and the lock manager wakes up the transaction);
            go to B
            end;
write_lock(X):
B:  if LOCK(X) = "unlocked"
        then LOCK(X) ← "write-locked"
    else begin
            wait (until LOCK(X) = "unlocked"
                and the lock manager wakes up the transaction);
            go to B
            end;
unlock (X):
    if LOCK(X) = "write-locked"
        then begin LOCK(X) ← "unlocked";
                wakeup one of the waiting transactions, if any
                end
    else it LOCK(X) = "read-locked"
        then begin
                no_of_reads(X) ← no_of_reads(X) −1;
                if no_of_reads(X) = 0
                    then begin LOCK(X) = "unlocked";
                            wakeup one of the waiting transactions, if any
                            end
                end;
```

Figure 22.2
Locking and unlocking operations for two-mode (read-write or shared-exclusive) locks.

2PL Concurrency Control (cont.)

Two-Phase Locking Techniques: Essential components

Lock Manager: Subsystem of DBMS that manages locks on data items.

Lock table: Lock manager uses it to store information about *locked data items*, such as: data item id, transaction id, lock mode, list of waiting transaction ids, etc. One simple way to implement a lock table is through linked list (shown). Alternatively, a **hash table** with item id as hash key can be used.

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

2PL Concurrency Control (cont.)

Rules for locking:

- Transaction must request appropriate lock on a data item X before it reads or writes X.
- If T holds a write (exclusive) lock on X, it can both read and write X.
- If T holds a read lock on X, it can only read X.
- T must unlock all items that it holds before terminating
- (also T cannot unlock X unless it holds a lock on X).

2PL Concurrency Control (cont.)

Lock conversion

Lock upgrade: existing read lock to write lock

if Ti holds a read-lock on X, and no other Tj holds a read-lock on X (i ≠ j), then
it is possible to convert (**upgrade**) read-lock(X) to write-lock(X)
else
force Ti to wait until all other transactions Tj that hold read locks on X release their locks

Lock downgrade: existing write lock to read lock

if Ti holds a write-lock on X

(*this implies that no other transaction can have any lock on X*)
then it is possible to convert (**downgrade**) write-lock(X) to read-lock(X)

2PL Concurrency Control (cont.)

Two-Phase Locking Rule:

Each transaction should have **two phases**: (a) Locking (Growing) phase, and (b) Unlocking (Shrinking) Phase.

Locking (Growing) Phase: A transaction applies locks (read or write) on desired data items one at a time. Can also try to upgrade a lock.

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

Requirement: For a transaction these two phases must be mutually exclusively, that is, during locking phase no unlocking or downgrading of locks can occur, and during unlocking phase no new locking or upgrading operations are allowed.

2PL Concurrency Control (cont.)

Basic Two Phase Locking:

When transaction starts executing, it is in the **locking phase,** and it can request locks on new items or upgrade locks. A transaction may be blocked (forced to wait) if a lock request is not granted. (This may lead to several transactions being in a *state of deadlock* – see later)

Once the transaction unlocks an item (or downgrades a lock), it starts its **shrinking phase** and can no longer upgrade locks or request new locks.

The combination of locking rules and 2-phase rule *ensures serializable schedules*

Theorem: If *every transaction* in a schedule follows the 2PL rules, the schedule must be serializable. (Proof is by contradiction.)

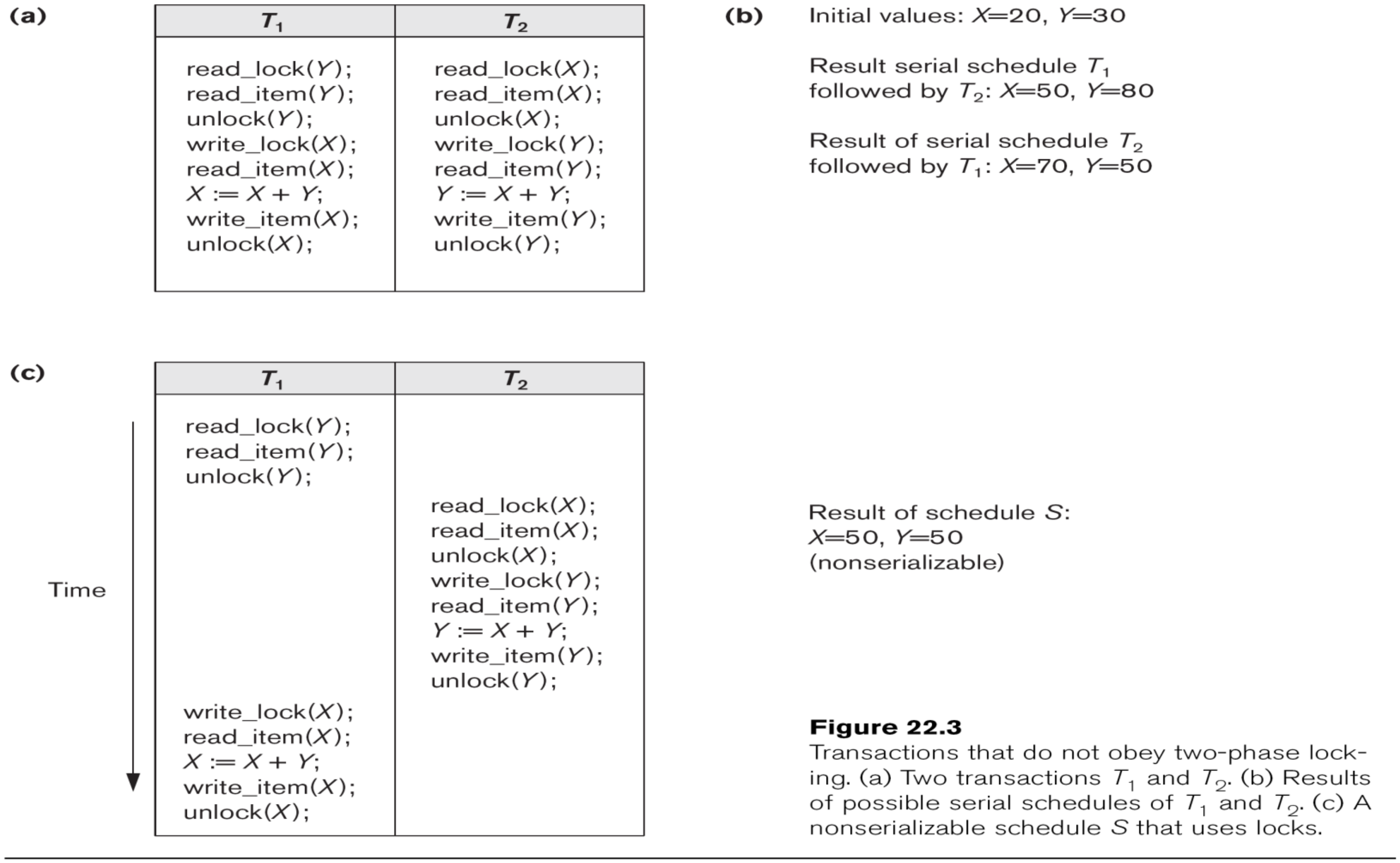
2PL Concurrency Control (cont.)

Some Examples:

Rules of locking alone do not enforce serializability – Figure 22.3(c) shows a schedule that follows locking rules but *is not serializable*

Figure 22.4 shows how the transactions in Figure 22.3(a) can be modified to follow the two-phase rule (by delaying the first unlock operation till all locking is completed).

The schedule in 22.3(c) would not be allowed if the transactions are changed as in 22.4 – a state of deadlock would occur instead because T2 would try to lock Y (which is locked by T1 in conflicting mode) and forced to wait, then T1 would try to lock X (which is locked by T2 in conflicting mode) and forced to wait – neither transaction can continue to unlock the item they hold as they are both blocked (waiting) – see Figure 22.5



(a)

T_1	T_2
read_lock(Y);	read_lock(X);
read_item(Y);	read_item(X);
unlock(Y);	unlock(X);
write_lock(X);	write_lock(Y);
read_item(X);	read_item(Y);
$X := X + Y$;	$Y := X + Y$;
write_item(X);	write_item(Y);
unlock(X);	unlock(Y);

(b) Initial values: X=20, Y=30

Result serial schedule T_1 followed by T_2: X=50, Y=80

Result of serial schedule T_2 followed by T_1: X=70, Y=50

(c)

Time

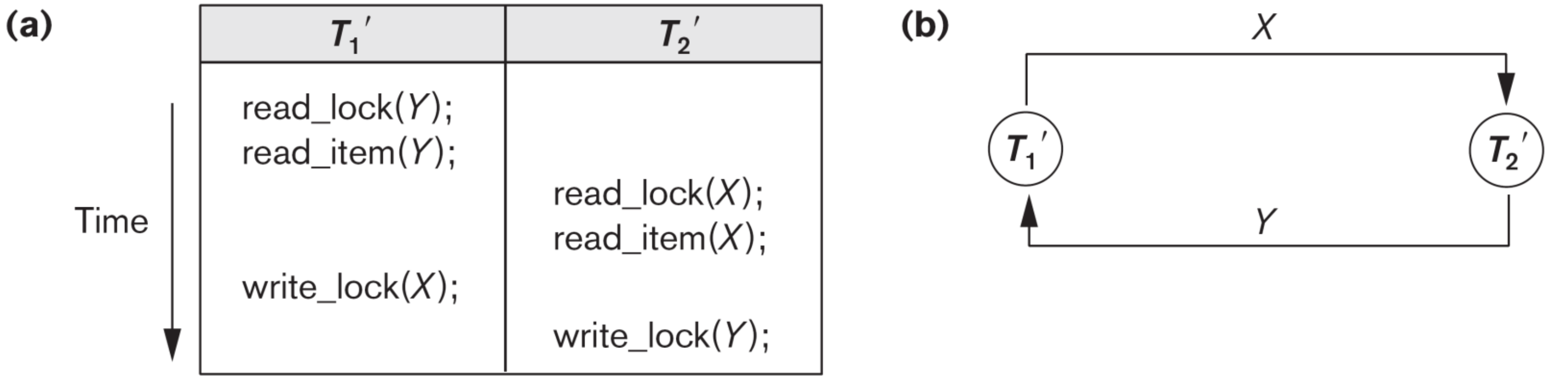
T_1	T_2
read_lock(Y);	
read_item(Y);	
unlock(Y);	
	read_lock(X);
	read_item(X);
	unlock(X);
	write_lock(Y);
	read_item(Y);
	$Y := X + Y$;
	write_item(Y);
	unlock(Y);
write_lock(X);	
read_item(X);	
$X := X + Y$;	
write_item(X);	
unlock(X);	

Result of schedule S:
X=50, Y=50
(nonserializable)

Figure 22.3
Transactions that do not obey two-phase locking. (a) Two transactions T_1 and T_2. (b) Results of possible serial schedules of T_1 and T_2. (c) A nonserializable schedule S that uses locks.

Figure 22.4
Transactions T_1' and T_2', which are the same as T_1 and T_2 in Figure 22.3, but follow the two-phase locking protocol. Note that they can produce a deadlock.

T_1'	T_2'
read_lock(Y);	read_lock(X);
read_item(Y);	read_item(X);
write_lock(X);	write_lock(Y);
unlock(Y)	unlock(X)
read_item(X);	read_item(Y);
$X := X + Y$;	$Y := X + Y$;
write_item(X);	write_item(Y);
unlock(X);	unlock(Y);



T_1'	T_2'
read_lock(Y);	
read_item(Y);	
	read_lock(X);
	read_item(X);
write_lock(X);	
	write_lock(Y);

Figure 22.5
Illustrating the deadlock problem. (a) A partial schedule of T_1' and T_2' that is in a state of deadlock. (b) A wait-for graph for the partial schedule in (a).

2PL Concurrency Control (cont.)

Other Variations of Two-Phase Locking:

Conservative 2PL: A transaction must *lock all its items before starting execution*. If any item it needs is not available, it locks no items and tries again later. Conservative 2PL *has no deadlocks* since no lock requests are issued once transaction execution starts.

Strict 2PL: All items that are *writelocked* by a transaction are not released (unlocked) until *after the transaction commits*. This is the most commonly used two-phase locking algorithm, and ensures strict schedules (for recoverability).

Rigorous 2PL: All items that are *writelocked or readlocked* by a transaction are not released (unlocked) until *after the transaction commits*. Also guarantees strict schedules.

2PL Concurrency Control (cont.)

Deadlock Example (see Figure 22.5)

T1'	**T2'**
read_lock (Y);	
read_item (Y);	
	read_lock (X);
	read_item (Y);
write_lock (X);	
(waits for X)	write_lock (Y);
	(waits for Y)

Deadlock (T1' and T2')

T1' and T2' did follow two-phase policy but they are deadlock

2PL Concurrency Control (cont.)

Dealing with Deadlock

Deadlock prevention

System enforces additional rules (deadlock prevention protocol) to ensure deadlock do not occur – many protocols (see later for some examples)

Deadlock detection and resolution

System checks for a state of deadlock – if a deadlock exists, one of the transactions involved in the deadlock is aborted

2PL Concurrency Control (cont.)

1. Deadlock detection and resolution

In this approach, deadlocks are allowed to happen. The system maintains a *wait-for graph* for detecting cycles. If a cycle exists, then one transaction involved in the cycle is selected (victim) and rolled-back (aborted).

A **wait-for graph** contains a node for each transaction. When a transaction (say Ti) is blocked because it requests an item X held by another transaction Tj in conflicting mode, a directed edge is created from Ti to Tj (Ti is waiting on Tj to unlock the item). The system checks for cycles; if a cycle exists, a state of deadlock is detected (see Figure 22.5).

2PL Concurrency Control (cont.)

2. Deadlock prevention

There are several protocols. Some of them are:

1. Conservative 2PL, as we discussed earlier.

2. No-waiting protocol: A transaction never waits; if Ti requests an item that is held by Tj in conflicting mode, Ti is aborted. Can result in *needless transaction aborts* because deadlock might have never occurred

3. Cautious waiting protocol: If Ti requests an item that is held by Tj in conflicting mode, the system checks the status of Tj; if Tj is *not blocked,* then Ti waits – if Tj *is blocked,* then Ti aborts. Reduces the number of needlessly aborted transactions

2PL Concurrency Control (cont.)

2. Deadlock prevention (cont.)

4. Wound-wait and wait-die: Both use transaction timestamp TS(T), which is a monotonically increasing unique id given to each transaction based on their starting time

TS(T1) < TS(T2) means that T1 started before T2 (T1 *older than* T2)

(Can also say T2 *younger than* T1)

<u>Wait-die:</u>

If Ti requests an item X that is held by Tj in conflicting mode, then

if TS(Ti) < TS(Tj) then Ti waits (on a younger transaction Tj)

else Ti dies (if Ti is younger than Tj, it aborts)

[In wait-die, transactions only wait on *younger transactions* that started later, so no cycle ever occurs in wait-for graph – if transaction requesting lock is younger than that holding the lock, requesting transaction aborts (dies)]

2PL Concurrency Control (cont.)

2. Deadlock prevention (cont.)

4. Wound-wait and wait-die (cont.):

<u>Wound-wait:</u>

If Ti requests an item X that is held by Tj in conflicting mode, then

```
    if TS(Ti) < TS(Tj) then Tj is aborted (Ti wounds younger Tj)
        else Ti waits (on an older transaction Tj)
```

[In wound-wait, transactions only wait on older *transactions* that started earlier, so no cycle ever occurs in wait-for graph – if transaction requesting lock is older than that holding the lock, transaction holding the lock is preemptively aborted]

Database Concurrency Control

Dealing With Starvation

Starvation occurs when a particular transaction consistently waits or gets restarted and never gets a chance to proceed further. Solution is to use a *fair priority-based scheme* that increases the priority for transaction the longer they wait.

Examples of Starvation:

1. In deadlock detection/resolution it is possible that the same transaction may consistently be selected as victim and rolled-back.
2. In conservative 2PL, a transaction may never get started because all the items needed are never available at the same time.
3. In Wound-Wait scheme a younger transaction may always be wounded (aborted) by a long running older transaction which may create starvation.

Other Concurrency Control Methods

Some CCMs other than 2PL:

- Timestamp Ordering (TO) CCM
- Optimistic (Validation-Based) CCMs
- Multi-version CCMs:
  - Multiversion 2PL with Certify Locks
  - Multiversion TO

Timestamp Ordering (TO) CCM

Timestamp

A monotonically increasing identifier (e.g., an integer, or the system clock time when a transaction starts) indicating the start order of a transaction. A larger timestamp value indicates a more recently started transaction.

Timestamp of transaction T is denoted by **TS(T)**

Timestamp ordering CCM uses the transaction timestamps to serialize the execution of concurrent transactions – allows only *one equivalent serial order* based on the transaction timestamps

Timestamp Ordering (TO) CCM (cont.)

Instead of locks, systems keeps track of two values for each data item X:

- **Read_TS(X):** The largest timestamp among all the timestamps of transactions that have successfully read item X
- **Write_TS(X):** The largest timestamp among all the timestamps of transactions that have successfully written X

- When a transaction T requests to read or write an item X, TS(T) is compared with read_TS(X) and write_TS(X) to determine if request is *out-of-order*

Timestamp Ordering (TO) CCM (cont.)

Basic Timestamp Ordering

1. Transaction T requests a write_item(X) operation:
   a. If read_TS(X) > TS(T) or if write_TS(X) > TS(T), then a younger transaction has already read or written the data item so abort and roll-back T and reject the operation.
   b. Otherwise, execute write_item(X) of T and set write_TS(X) to TS(T).
2. Transaction T requests a read_item(X) operation:
   a. If write_TS(X) > TS(T), then a younger transaction has already written the data item X so abort and roll-back T and reject the operation.
   b. Otherwise, execute read_item(X) of T and set read_TS(X) to the larger of TS(T) and the current read_TS(X).

Multiversion CCMs

Assumes that multiple versions of an item can exists *at the same time*

An implicit assumption is that when a data item is updated, the new value *replaces the old value*

Only **current version** of an item exists

Multiversions techniques assume that *multiple versions of the same item* coexist and can be utilized by the CCM

We discuss two variations of multiversion CCMs:

- Multiversion Timestamp Ordering (TO)
- Multiversion Two-phase Locking (2PL)

Multiversion CCMs (cont.)

Multiversion Timestamp Ordering

Concept

Assumes that a number of versions of a data item X exist (X0, X1, X2, ...); each version has its own read_TS and write_TS

Can allocate the right version to a read operation - thus read operation is *always successful*

Significantly more storage (RAM and disk) is required to maintain multiple versions; to check unlimited growth of versions, a *window* is kept – say last 10 versions.

Multiversion CCMs (cont.)

Multiversion Timestamp Ordering (cont.)

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

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

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

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

Multiversion CCMs (cont.0

Multiversion Timestamp Ordering (cont.)

To ensure serializability, the following two rules are used.

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

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

Multiversion CCMs (cont.)

Multiversion 2PL Using Certify Locks

Concepts:

Allow one or more transactions T’ to concurrently read a data item X while X is write locked by a different transaction T.

Accomplished by allowing **two versions** of a data item X: one **committed version** (this can be read by other transactions) and one **local version** being written by T.

Multiversion CCMs (cont.)

Multiversion 2PL Using Certify Locks (cont.)

Concepts (cont.):

Write lock on X by T is no longer exclusive – it can be held with *read locks* from other transactions T'; however, only one transaction can hold *write lock* on X.

Before **local value** of X (written by T) can become the committed version, the write lock must be **upgraded to a certify lock** by T (*certify lock* is exclusive in this scheme) – T must wait until any read locks on X by other transactions T' are released before upgrading to certify lock.

(a)

	Read	Write
Read	Yes	No
Write	No	No

(b)

	Read	Write	Certify
Read	Yes	Yes	No
Write	Yes	No	No
Certify	No	No	No

Figure 22.6
Lock compatibility tables.
(a) A compatibility table for read/write locking scheme.
(b) A compatibility table for read/write/certify locking scheme.

Multiversion CCMs (cont.)

Multiversion 2PL Using Certify Locks (cont.)

Steps

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

Multiversion CCMs (cont.)

Multiversion 2PL Using Certify Locks (cont.)

Note:

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

Validation-based (optimistic) CCMs

Known as **optimistic** concurrency control because it assumes that *few conflicts will occur*.

Unlike other techniques, system does not have to perform checks before each read and write operation – system performs checks only at end of transaction (during validation phase)

Three Phases: **Read** phase, **Validation** Phase, **Write** Phase – we discuss each phase next

Validation-based (optimistic) CCMs

While transaction T is executing, system collects information about **read_set(T)** and **write_set(T)** (the set of item read and written by T) as well as start/end time of each phase

Read phase is the time when the transaction is actually running and executing read and write operations

Read phase: A transaction can read values of committed data items. However, writes are applied only to **local copies** (versions) of the data items (hence, it can be considered as a multiversion CCM).

Validation-based (optimistic) CCMs (cont.)

Validation phase: Serializability is checked by determining any conflicts with other concurrent transactions.

This phase for Ti checks that, for each transaction Tj that is either committed or is in its validation phase, one of the following conditions holds:

1. Tj completes its *write phase* before Ti starts its *read phase*.
2. Ti starts its *write phase* after Tj completes its *write phase*, and the **read_set(Ti)** has no items in common with the **write_set(Tj)**
3. Both **read_set(Ti)** and **write_set(Ti)** have no items in common with the **write_set(Tj)**, and Tj completes its read phase before Ti completes its write phase.

Validation-based (optimistic) CCMs (cont.)

When validating Ti, the first condition is checked first for each transaction Tj, since (1) is the simplest condition to check. If (1) is false for a particular Tj, then (2) is checked and only if (2) is false is (3) is checked. If none of these conditions holds for any Tj, the validation fails and Ti is aborted.

Write phase: On a successful validation, transaction updates are applied to the database on disk and become the committed versions of the data items; otherwise, transactions that fail the validation phase are restarted.

Multiple-Granularity 2PL

The size of a data item is called its **granularity**. Granularity can be coarse (entire database) or it can be fine (a tuple (record) or an attribute value of a record). Data item granularity significantly affects concurrency control performance. Degree of concurrency is low for coarse granularity and high for fine granularity. Example of data item granularity:

1. A field of a database record (an attribute of a tuple).
2. A database record (a tuple).
3. A disk block.
4. An entire file (relation).
5. The entire database.

Multiple-Granularity 2PL (cont.)

The following slide illustrates a hierarchy of granularity of items from coarse (database) to fine (record). The root represents an item that includes the whole database, followed by file items (tables/relations), disk page items within each file, and record items within each disk page.

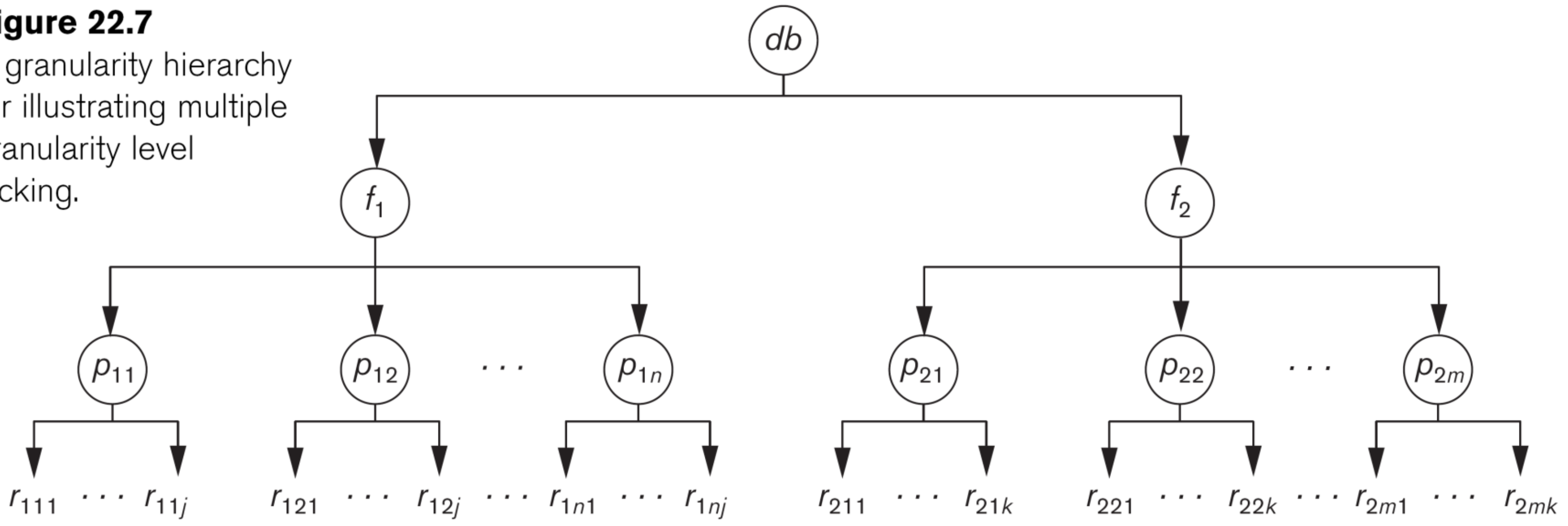

db
f_1
f_2
p_{11}
p_{12}
. . .
p_{1n}
p_{21}
p_{22}
. . .
p_{2m}
r_{111} . . . r_{11j}
r_{121} . . . r_{12j} . . . r_{1n1} . . . r_{1nj}
r_{211} . . . r_{21k}
r_{221} . . . r_{22k} . . . r_{2m1} . . . r_{2mk}

Figure 22.7
A granularity hierarchy for illustrating multiple granularity level locking.

Multiple-Granularity 2PL (cont.)

To manage such hierarchy, in addition to read or **shared** (S) and write or **exclusive** (X) locking modes, three additional locking modes, called **intention lock** modes are defined:

Intention-shared (IS): indicates that a shared lock(s) will be requested on some descendent nodes(s).

Intention-exclusive (IX): indicates that an exclusive lock(s) will be requested on some descendent nodes(s).

Shared-intention-exclusive (SIX): indicates that the current node is requested to be locked in shared mode but an exclusive lock(s) will be requested on some descendent nodes(s).

Multiple-Granularity 2PL (cont.)

These locks are applied using the lock compatibility table in the next slide. Locking always begins at the root node and proceeds down the tree, while unlocking proceeds in the opposite direction:

	IS	IX	S	SIX	X
IS	Yes	Yes	Yes	Yes	No
IX	Yes	Yes	No	No	No
S	Yes	No	Yes	No	No
SIX	Yes	No	No	No	No
X	No	No	No	No	No

Figure 22.8
Lock compatibility matrix for multiple granularity locking.

Multiple-Granularity 2PL (cont.)

The set of rules which must be followed for producing serializable schedule are

1. The lock compatibility table must be adhered to.
2. The root of the tree must be locked first, in any mode.
3. A node N can be locked by a transaction T in S (or X) mode only if the parent node is already locked by T in either IS (or IX) mode.
4. A node N can be locked by T in X, IX, or SIX mode only if the parent of N is already locked by T in either IX or SIX mode.
5. T can lock a node only if it has not unlocked any node (to enforce 2PL policy).
6. T can unlock a node, N, only if none of the children of N are currently locked by T.

Multiple-Granularity 2PL (cont.)

Next slide shows an example of how the locking and unlocking may proceed in a schedule:

T_1	T_2	T_3
IX(db)		
IX(f_1)		
	IX(db)	
		IS(db)
		IS(f_1)
		IS(p_{11})
IX(p_{11})		
X(r_{111})		
	IX(f_1)	
	X(p_{12})	
		S(r_{11j})
IX(f_2)		
IX(p_{21})		
X(p_{211})		
unlock(r_{211})		
unlock(p_{21})		
unlock(f_2)		
		S(f_2)
	unlock(p_{12})	
	unlock(f_1)	
	unlock(db)	
unlock(r_{111})		
unlock(p_{11})		
unlock(f_1)		
unlock(db)		
		unlock(r_{11j})
		unlock(p_{11})
		unlock(f_1)
		unlock(f_2)
		unlock(db)

Figure 22.9
Lock operations to illustrate a serializable schedule.

Purpose of Database Recovery

- To bring the database into a consistent state after a failure occurs.
- To ensure the transaction properties of **Atomicity** (a transaction must be done in its entirety; otherwise, it has to be rolled back) and **Durability** (a committed transaction cannot be canceled and all its updates must be applied permanently to the database).
- After a failure, the **DBMS recovery manager** is responsible for bringing the system into a consistent state before transactions can resume.

Types of Failure

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

The Log File

- Holds the information that is necessary for the recovery process
- Records all relevant operations in the order in which they occur (looks like a *schedule of transactions*, see Chapter 21)
- Is an append-only file.
- Holds various types of log records (or log entries).

Log File Entries (from Chapter 21)

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.

The Log File (cont.)

For **write_item** log entry, *old value* of item before modification (**BFIM** - BeFore Image) and the *new value* after modification (**AFIM** – AFter Image) are stored. BFIM needed for UNDO, AFIM needed for REDO. A sample log is given below. **Back P** and **Next P** point to the previous and next log records of the same transaction.

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

Database Cache

Database Cache: A set of *main memory* buffers; each buffer typically holds contents of one disk block. Stores the disk blocks that contain the data items being read and written by the database transactions.

Data Item Address: (disk block address, offset, size in bytes).

Cache Table: Table of entries of the form (buffer addr, disk block addr, modified bit, pin/unpin bit, ...) to indicate which disk blocks are currently in the cache buffers.

Database Cache (cont.)

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

Pin-Unpin: If a buffer is pinned, it cannot be written back to disk until it is unpinned.

Modified: Indicates that one or more data items in the buffer have been changed.

Data Update

- **Immediate Update:** A data item modified in cache can be written back to disk *before the transaction commits.*
- **Deferred Update:** A modified data item in the cache cannot be written back to disk till *after the transaction commits* (buffer is **pinned**).

- **Shadow update:** The modified version of a data item does not overwrite its disk copy but is written at a separate disk location (new version).
- **In-place update:** The disk version of the data item is overwritten by the cache version.

UNDO and REDO Recovery Actions

To maintain atomicity and durability, some transaction's may have their operations **redone** or **undone** during recovery. UNDO (roll-back) is needed for transactions that are not committed yet. REDO (roll-forward) is needed for committed transactions whose writes may have not yet been flushed from cache to disk.

Undo: Restore all BFIMs from log to database on disk. UNDO proceeds backward in log (from most recent to oldest UNDO).

Redo: Restore all AFIMs from log to database on disk. REDO proceeds forward in log (from oldest to most recent REDO).

Write-ahead Logging Protocol

The information needed for recovery must be written to the log file on disk before changes are made to the database on disk. **Write-Ahead Logging** (WAL) protocol consists of two rules:

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

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

Checkpointing

To minimize the **REDO** operations during recovery. The following steps define a checkpoint operation:

- Suspend execution of transactions temporarily.
- Force write modified buffers from cache to disk.
- Write a [checkpoint] record to the log, save the log to disk. This record also includes other info., such as the *list of active transactions* at the time of checkpoint.
- Resume normal transaction execution.

During recovery **redo** is required only for transactions that have committed *after the last [checkpoint] record* in the log.

Checkpointing (cont.)

Steps 1 and 4 in previous slide are not realistic.

A variation of checkpointing called **fuzzy checkpointing** allows transactions to continue execution during the checkpointing process.

We discuss fuzzy checkpointing in the ARIES protocol later.

Other Database Recovery Concepts

Steal/No-Steal and Force/No-Force

Specify how to flush database cache buffers to database on disk:

Steal: Cache buffers updated by a transaction may be flushed to disk before the transaction commits (recovery may require UNDO).

No-Steal: Cache buffers cannot be flushed until after transaction commit (NO-UNDO). (Buffers are *pinned* till transactions commit).

Force: Cache is flushed (forced) to disk before transaction commits (NO-REDO).

No-Force: Some cache flushing may be deferred till after transaction commits (recovery may require REDO).

These give rise to four different ways for handling recovery:

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

Deferred Update (NO-UNDO/REDO) Recovery Protocol

System must impose **NO-STEAL** rule. Recovery subsystem analyzes the log, and creates two lists:

Active Transaction list: All active (uncommitted) transaction ids are entered in this list.
Committed Transaction list: Transactions committed after the last checkpoint are entered in this table.

During recovery, transactions in **commit** list are **redone;** transactions in **active** list are *ignored* (because of NO-STEAL rule, none of their writes have been applied to the database on disk). Some transactions may be **redone** twice; this does not create inconsistency because **REDO** is "**idempotent**", that is, one REDO for an AFIM is equivalent to multiple REDO for the same AFIM.

Deferred Update (NO-UNDO/REDO) Recovery Protocol (cont.)

Advantage: Only **REDO** is needed during recovery.

Disadvantage: Many buffers may be pinned while waiting for transactions that updated them to commit, so system may run out of cache buffers when requests are made by new transactions.

UNDO/NO-REDO Recovery Protocol

In this method, **FORCE** rule is imposed by system (AFIMs of a transaction are flushed to the database on disk under Write Ahead Logging *before the transaction commits*).

Transactions in active list are **undone**; transactions in committed list are ignored (because based on FORCE rule, all their changes are already written to the database on disk).

<u>Advantage:</u> During recovery, only **UNDO** is needed.

<u>Disadvantages:</u> 1. Commit of a transaction is delayed until all its changes are force-written to disk. 2. Some buffers may be written to disk multiple times if they are updated by several transactions.

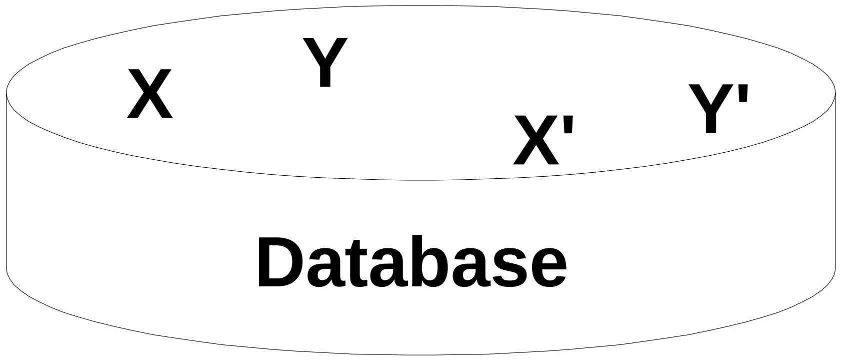
UNDO/REDO Recovery Protocol

Recovery can require both UNDO of some transactions and REDO of other transactions (Corresponds to STEAL/NO-FORCE). Used most often in practice because of disadvantages of the other two methods. To minimize REDO, checkpointing is used. The recovery performs:

1. **Undo** of a transaction if it is in the active transaction list.
2. **Redo** of a transaction if it is in the list of transactions that committed since the last checkpoint.

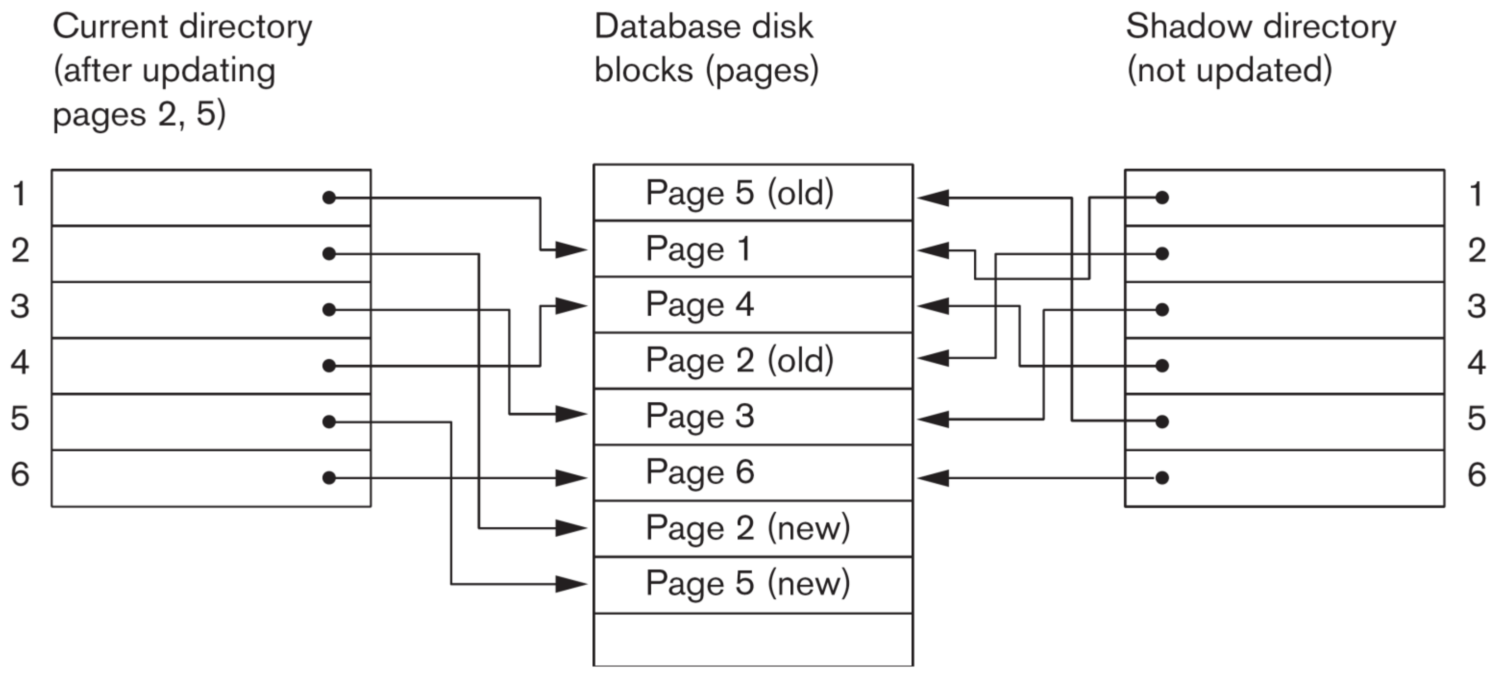
Shadow Paging (NO-UNDO/NO-REDO)

The AFIM does not overwrite its BFIM but is recorded at another place (new version) on the disk. Thus, a data item can have AFIM and BFIM (Shadow copy of the data item) at two different places on the disk.



X and Y: Shadow (old) copies of data items
X` and Y`: Current (new) copies of data items

Figure 23.4
An example of shadow paging.


Current directory (after updating pages 2, 5)
Database disk blocks (pages)
Shadow directory (not updated)
1
2
3
4
5
6
Page 5 (old)
Page 1
Page 4
Page 2 (old)
Page 3
Page 6
Page 2 (new)
Page 5 (new)
1
2
3
4
5
6

[5]The directory is similar to the page table maintained by the operating system for each process.

ARIES Database Recovery

Used in practice, it is based on:

1. WAL (Write Ahead Logging)
2. Repeating history during redo: ARIES will retrace all actions of the database system prior to the crash to reconstruct the correct database state.
3. Logging changes during undo: It will prevent ARIES from repeating the completed undo operations if a failure occurs during recovery, which causes a restart of the recovery process.

ARIES Database Recovery (cont.)

The ARIES recovery algorithm consists of three steps:

1. **Analysis:** step identifies the dirty (updated) page buffers in the cache and the set of transactions active at the time of crash. The set of transactions that committed after the last checkpoint is determined, and the appropriate point in the log where redo is to start is also determined.

2. **Redo:** necessary redo operations are applied.

3. **Undo:** log is scanned backwards and the operations of transactions active at the time of crash are undone in reverse order.

ARIES Database Recovery (cont.)

The Log and Log Sequence Number (LSN)

A log record is written for (a) data update (Write), (b) transaction commit, (c) transaction abort, (d) undo, and (e) transaction end. In the case of undo a *compensating* log record is written.

A **unique LSN** is associated with every log record. LSN increases monotonically and indicates the disk address of the log record it is associated with. In addition, each data page stores the LSN of the latest log record corresponding to a change for that page.

ARIES Database Recovery (cont.)

The Log and Log Sequence Number (LSN) (cont.)

A log record stores:

1. LSN of previous log record for same transaction: It links the log records of each transaction.
2. Transaction ID.
3. Type of log record.

For a write operation, additional information is logged:

1. Page ID for the page that includes the item
2. Length of the updated item
3. Its offset from the beginning of the page
4. BFIM of the item
5. AFIM of the item

ARIES Database Recovery (cont.)

The Transaction table and the Dirty Page table

For efficient recovery, the following tables are also stored in the log during checkpointing:

Transaction table: Contains an entry for each active transaction, with information such as transaction ID, transaction status, and the LSN of the most recent log record for the transaction.

Dirty Page table: Contains an entry for each dirty page (buffer) in the cache, which includes the page ID and the LSN corresponding to the earliest update to that page.

ARIES Database Recovery (cont.)

"Fuzzy" Checkpointing

A checkpointing process does the following:

1. Writes a *begin_checkpoint* record in the log, then forces updated (dirty) buffers to disk.
2. Writes an *end_checkpoint* record in the log, along with the contents of transaction table and dirty page table.
3. Writes the LSN of the *begin_checkpoint* record to a special file. This special file is accessed during recovery to locate the last checkpoint information.

To allow the system to continue to execute transactions, ARIES uses "fuzzy checkpointing".

ARIES Database Recovery (cont.)

The following steps are performed for recovery

1. **Analysis phase:** Start at the begin_checkpoint record and proceed to the end_checkpoint record. Access transaction table and dirty page table that were appended to the log. During this phase some other records may be written to the log and the transaction table may be modified. The analysis phase compiles the set of redo and undo operations to be performed and ends.
2. **Redo phase:** Starts from the point in the log where all dirty pages have been flushed, and move forward. Operations of committed transactions are redone.
3. **Undo phase:** Starts from the end of the log and proceeds backward while performing appropriate undo. For each undo it writes a compensating record in the log.

The recovery completes at the end of undo phase.

(a)

Lsn	Last_lsn	Tran_id	Type	Page_id	Other_information
1	0	T_1	update	C	. . .
2	0	T_2	update	B	. . .
3	1	T_1	commit		. . .
4	begin checkpoint				
5	end checkpoint				
6	0	T_3	update	A	. . .
7	2	T_2	update	C	. . .
8	7	T_2	commit		. . .

(b)

TRANSACTION TABLE

Transaction_id	Last_lsn	Status
T_1	3	commit
T_2	2	in progress

DIRTY PAGE TABLE

Page_id	Lsn
C	1
B	2

(c)

TRANSACTION TABLE

Transaction_id	Last_lsn	Status
T_1	3	commit
T_2	8	commit
T_3	6	in progress

DIRTY PAGE TABLE

Page_id	Lsn
C	1
B	2
A	6

Figure 23.5
An example of recovery in ARIES. (a) The log at point of crash. (b) The Transaction and Dirty Page Tables at time of checkpoint. (c) The Transaction and Dirty Page Tables after the analysis phase.

Recovery in Multi-database Transactions (Two-phase commit)

A multidatabase transaction can access several databases: e.g. airline database, car rental database, credit card database. The transaction commits only when all these multiple databases agree to commit individually the part of the transaction they were executing. This commit scheme is referred to as "*two-phase commit*" (2PC). If any one of these nodes fails or cannot commit its part of the transaction, then the whole transaction is aborted. Each node recovers the transaction under its own recovery protocol.

Two-phase commit (cont.)

Phase 1: Coordinator (usually application program running in middle-tier of 3-tier architecture) sends "Ready-to-commit?" query to each participating database, then waits for replies. A participating database replies Ready-to-commit only after saving all actions in its local log on disk.

Phase 2: If coordinator receives Ready-to-commit signals from all participating databases, it sends Commit to all; otherwise, it send Abort to all.

This protocol can survive most types of crashes.

Questions

1. Explain the properties of transactions.
2. What is serializability? Explain.
3. What is 2PL? Explain.
4. Explain shadow paging.
5. Explain ARIES recovery algorithm.