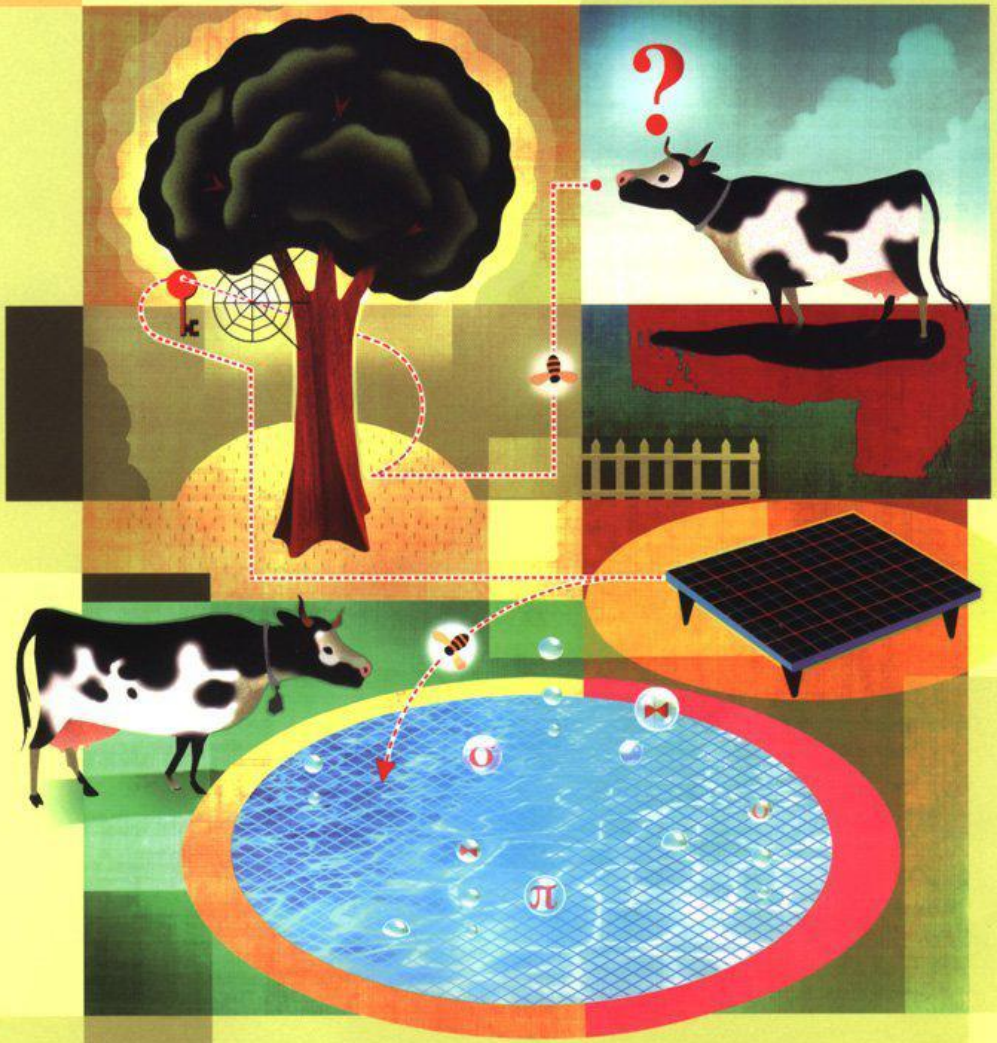
TRANSACTION MANAGEMENT

(introduction)

Database Management Systems

Second Edition



Raghu Ramakrishnan / Johannes Gehrke

Topics

1. ACID Properties
2. Transactions and Schedules
3. Concurrent Execution of Transactions
4. Lock-Based Concurrency Control
5. Introduction to Crash Recovery

What is a transaction ?

A transaction is defined as ***any one execution*** *of a user program in a DBMS*

- It differs from an execution of a program outside the DBMS in important ways.

e.g. Execution of C program on UNIX.

What are the issues in Transaction Management?

1. scheduling Several Transactions.

When a DBMS executes several Transitions concurrently , for perform reasons , it has to interleave the actions of several transactions , while giving users the effect of running their programs. An interleaved execution of several transactions, ***called a schedule.***

2. concurrency control.

- **how the DBMS handles concurrent executions.?**

What are the issues in Transaction Management?

3. crash recovery

- It deals with how a DBMS handles partial transaction(Incomplete Transactions).
- The DBMS ensures that the changes made by such partial transactions are not seen by other transactions.

ACID properties of Transactions

1. A→ Atomicity
2. C→ Consistency
3. I→ Isolation
4. D→ Durability

1. Atomicity :

- **Either all actions are carried out or none are.**
- Users should not have to worry about the effect of incomplete transactions (say, when a system crash occurs).

ACID properties of Transactions

2. Consistency

- Each transaction must preserve the consistency of the database.
- This property is called **consistency** and the DBMS assumes that it holds for each transaction.
- Ensuring this property of a transaction is the responsibility of the user.

ACID properties of Transactions

3. Isolation

- Users should be able to understand a transaction without considering the effect of other concurrently executing transactions, even if the DBMS interleaves the actions of several transactions for performance reasons. This property is sometimes referred to as **isolation:**

4. Durability.

- Once the DBMS informs the user that a transaction has been successfully completed, its effects should persist even if the system crashes before all its changes are reflected on disk. This property is called **durability.**

How the ACID properties are ensured in a DBMS?

1. Consistency

- Users are responsible for ensuring transaction consistency
- That is, the user who submits a transaction must ensure that when the transaction is complete , the transaction will leave the database in a **`consistent` state.**

For example, the user may have the consistency criterion that fund transfers between bank accounts should not change the total amount of money in the accounts.

How the ACID properties are ensured in a DBMS?

2. Isolation

- The isolation property is ensured by guaranteeing that even though actions of several transactions might be interleaved, the net effect is identical to executing all transactions one after the other in some serial order.

For example, if two transactions T1 and T2 are executed concurrently, the net effect is guaranteed to be equivalent to executing T1 followed by executing T2 or executing T2 followed by executing T1.

How the ACID properties are ensured in a DBMS?

3. Atomicity

Transactions can be incomplete for three kinds of reasons:

1. A transaction can be aborted
2. Terminated unsuccessfully
3. Some anomaly arises during execution.

In any of the above case , the transaction will the database in an inconsistent state

" A DBMS ensures transaction atomicity by *undoing the actions* of incomplete transactions."

To be able to do this, the DBMS maintains a record, **called the *log***, *of all writes to the database.*

How the ACID properties are ensured in a DBMS?

4. Durability

- The log is also used to ensure durability:
  - If the system crashes before the changes made by a completed transaction are written to disk, the log is used to remember and restore these changes when the system restarts.

Four Actions of a transaction

- A transaction can also be defined as a **set of actions** that are *partially ordered.*

The actions that can be executed by a transaction include :

1. **Reading** (R(A) → Reading Object A from memory)
2. **Writing** (W(A) → writing object A into memory)
3. **Aborting** (terminate and undo all the actions carried out thus far)
4. **Committing** (complete successfully)

What is a Schedule ?

A **schedule is a list of actions** from a set of transactions, and the order in which two actions of a transaction *T appear in* a schedule must be the same as the order in which they appear in *T.*

$T1$	$T2$
$R(A)$	
$W(A)$	
	$R(B)$
	$W(B)$
$R(C)$	
$W(C)$	

A Schedule Involving Two Transactions

Intuitively, a schedule represents an actual or potential execution sequence.

We move forward in time as we go down from one row to the next.

What is a complete schedule?

- A schedule that contains either an abort or a commit for each transaction whose actions are listed in it is called **a complete schedule.**
- A complete schedule must contain all the actions of every transaction that appears in it.

What is a complete schedule?

Examples

Incomplete →

$T1$	$T2$
$R(A)$	
$W(A)$	
	$R(B)$
	$W(B)$
$R(C)$	
$W(C)$	

Complete →

$T1$	$T2$
$R(A)$	
$W(A)$	
	$R(A)$
	$W(A)$
	$R(B)$
	$W(B)$
	Commit
$R(B)$	
$W(B)$	
Commit	

Complete →

$T1$	$T2$
$R(A)$	
$W(A)$	
	$R(A)$
	$W(A)$
	$R(B)$
	$W(B)$
	Commit
Abort	

Incomplete →

$T1$	$T2$
$W(A)$	
	$R(A)$
	$W(B)$
	Commit

What is a serial schedule?

-
- If the actions of different transactions are not interleaved that is, transactions are executed from start to finish, one by one , we call the schedule a **serial schedule**

Not serial →

$T1$	$T2$
$R(A)$	
$W(A)$	
	$R(B)$
	$W(B)$
$R(C)$	
$W(C)$	

serial →

$T1$	$T2$	$T3$
$R(X)$		
$W(X)$		
$Com.$		
	$R(Y)$	
	$W(Y)$	
	$Com.$	
		$R(Z)$
		$W(Z)$
		$Com.$

Why is concurrent execution of several transaction needed?

The DBMS interleaves the actions of different transactions to improve performance, in terms of increased throughput or improved response times for short transactions.

There are 2 motivation for Concurrent Execution

1. while one transaction is waiting for a page to be read in from disk, the CPU can process another transaction. Overlapping I/O and CPU activity reduces the amount of time disks and processors are idle, and increases system throughput .
2. Second, interleaved execution of a short transaction with a long transaction usually allows the short transaction to complete quickly.

What is a serializable schedule ?

When a **complete serial schedule** is executed against a consistent database, the result is also a consistent database.

A serializable schedule over a set S of committed transactions is a schedule whose effect on any consistent database instance is guaranteed to be identical to that of some complete serial schedule over S.

What are the Anomalies Associated with Interleaved Execution?

Let T1 and T2 be two consistency preserving, committed transactions.

Now , if the actions of T1 and T2 are interleaved , then two action on the same data object conflict **, if at least one of them is a write.**

Therefore , three anomalous situations can be described in terms of when the actions of two transactions T1 and T2 conflict with each other:

1. **Write-Read (WR) conflict**
2. **Read-write (RW) conflict**
3. **Write-Write (WW) conflict**

1. WR Conflict (Reading Uncommitted Data)

WR conflict arises when a transaction T2 could read a database object that has been modified by another transaction T1, which has not yet committed. *Such a read is called a* ***dirty read***.

:

1. WR Conflict (Reading Uncommitted Data)

Example :

Consider two transactions *T1 and T2,* each of which, run alone, preserves database consistency :

Consider fund transfer from one account A to another B.

Consider the following serial execution of T1 and T2.

(The task is to transfer $100 from A to B and then increment the funds in A and B by 6%)

1. T1 transfers $100 from A to B
2. T2 increments both A and B by 6 percent.

The database is consistent because the above schedule is serial

1. WR Conflict (Reading Uncommitted Data)

Example :

Now, Suppose that their actions are interleaved as below:

(1) The account transfer program T1 deducts $100 from account A,

(2) The deposit program T2 reads the current values of accounts A and B and adds 6 percent interest

(3) The account transfer program credits $100 to account B

The above schedule will not leave the database in a consistent state.

1. WR Conflict (Reading Uncommitted Data)

The corresponding schedule, which is the view the DBMS has of this series of events, is illustrated in Figure 18.2

$T1$	$T2$
$R(A)$	
$W(A)$	
	$R(A)$
	$W(A)$
	$R(B)$
	$W(B)$
	Commit
$R(B)$	
$W(B)$	
Commit	

Figure 18.2 Reading Uncommitted Data

The problem can be traced to the fact that the value of *A written by T1 is read by T2* before *T1 has completed all its changes.*

2. RW Conflict (Reading Uncommitted Data)

- RW Conflicts arise when a transaction T2 could change the value of an object A that has been read by a transaction T1, while T1 is still in progress.

- **This situation causes two problems :**

1. First, if T1 tries to read the value of A again, it will get a different result, even though it has not modified A in the meantime. It is called an **unrepeatable read.**

2. RW Conflict (Reading Uncommitted Data)

2. suppose that A=5.

   T1 → 1. Read A 2. Increment A (A =6)

   T2 → 1. Read A 2. Decrement A (A=4)

Any serial order should leave *A with a final value of 5. thus, the interleaved execution leads to an inconsistent state.*

3. WW Conflict (Reading Uncommitted Data)

- WW Conflicts arise when a transaction T2 could overwrite the value of an object A, which has already been modified by a transaction T1, while T1 is still in progress.

- Even if T2 does not read the value of A written by T1, a potential problem exists as the following example illustrates.

.

3. WW Conflict (Reading Uncommitted Data)

Example : Suppose that Harry and Larry are two employees.

Consistent criterion : Their salaries must be kept equal.

Consider the following action of T1 and T2 each of which is consistency preserving:

Actions of T1 :

1. set Harry's salary to $1000.
2. set Larry's salary to $1000

Actions of T2 :

1. set Harry's salary to $2000.
2. set Larry's salary to $2000

Any serial order of T1 and T2 , satisfies the **Consistent criterion .**

Notice that neither transaction reads a salary value before writing it such a write is called a **blind write**

3. WW Conflict (Reading Uncommitted Data)

Now, consider the following interleaving of the actions of T1 and T2:

1. T1 sets Harry's salary to $1,000
2. T2 sets Larry's salary to $2,000
3. T1 sets Larry's salary to $1,000
4. T2 sets Harry's salary to $2,000

It violates the desired consistency criterion that the two salaries must be equal.

The result is not identical to the result of either of the two possible serial executions, and the interleaved schedule is therefore **not serializable.**

Serializable schedules Involving Aborted Transactions

- When there are aborted transactions in a schedule , all actions of aborted transactions are to be undone, and we can therefore imagine that they were never carried out.

- Using this intuition, we extend the definition of a serializable schedule as follows:

A serializable schedule over a set S of transactions is a schedule whose effect on any consistent database instance is guaranteed to be identical to that of some complete serial schedule **over the set of committed transactions in S.**

Serializable schedules Involving Aborted Transactions

Aborting a transactions may be impossible in some situations.

$T1$	$T2$
$R(A)$	
$W(A)$	
	$R(A)$
	$W(A)$
	$R(B)$
	$W(B)$
	Commit
Abort	

Now, *T2 has read a value for A that should never have* been there. If *T2 had not yet committed,* we could deal with the situation by aborting T2 . Since the actions of T2 are committed , we cannot abort T2. Such a schedule becomes ***unrecoverable.***

Serializable schedules Involving Aborted Transactions

- A **recoverable schedule** is one in which transactions commit only after (and if!) all transactions whose changes they read commit.

- If transactions read only the changes of committed transactions, not only is the schedule recoverable ,also aborting a transaction can be accomplished without cascading the abort to other transactions

  Such a schedule is said to **avoid cascading aborts**

In the example , if T1 has committed its transaction before T2 reads A, then this becomes recoverable schedule . Aborting T1 will not affect T2.

Serializable schedules Involving Aborted Transactions

There is another potential problem in undoing the actions of a transaction as explained in the following example:

Let A =5

Consider the following interleaved actions of T1 and T2

1. Transaction T1 reads A and adds 1 to A (A=6)
2. Transaction T2 reads A and adds 1 to A (A=7)
3. T1 aborts
4. T2 commits.

Since T1 aborts (in step 3) its changes must be undone. That is, that value of A must be restored to 5. Now when T2 decides to commit , its change to *A is inadvertently lost. (because A is now 5, not 7).*

A concurrency control technique called Strict 2PL can prevent this problem

Lock Based Concurrency Control

What is a locking protocol?

So, now it is clear that a DBMS must ensure that

1. Only serializable and recoverable schedules are allowed
2. No actions of committed transactions are lost while undoing aborted transactions.

DBMS typically uses a ***locking protocol*** *to achieve this.*

A locking protocol is a set of rules to be followed by each transaction in order to ensure that even though actions of several transactions might be interleaved, the net effect is identical to executing all transactions in some serial order

Strict Two-Phase Locking (Strict 2PL).

The most widely used locking protocol, called Strict Two-Phase Locking, or Strict 2PL, has two rules. The first rule is:

1st Rule : If a transaction T wants to read (respectively, modify) an object, it first requests a shared (respectively, exclusive) lock on the object.

- If T holds an exclusive lock on an object , then DBMS ensures that no other transaction holds shared or exclusive lock on the same object.
- A transaction that requests a lock is suspended until the DBMS is able to grant it the requested lock.

Strict Two-Phase Locking (Strict 2PL).

2 Rule : All locks held by a transaction are released when the transaction is completed.

In general , Strict 2PL protocol ensures the following :

1. Only `safe' interleavings of transactions are allowed.
2. If two transactions access the same object, and one of them wants to modify it, their actions are effectively ordered serially

Strict Two-Phase Locking (Strict 2PL).

Notations for shared and exclusive locks:

S(O) → A transaction T requesting a shared lock on object O.

X(O) → A transaction T requesting an exclusive lock on object O.

Example : consider the schedule shown below.

$T1$	$T2$
$R(A)$	
$W(A)$	
	$R(A)$
	$W(A)$
	$R(B)$
	$W(B)$
	Commit
$R(B)$	
$W(B)$	
Commit	

this schedule is not serializable. Because this interleaved execution is not equivalent to any serial execution of T1 and T2

Figure 18.2 Reading Uncommitted Data

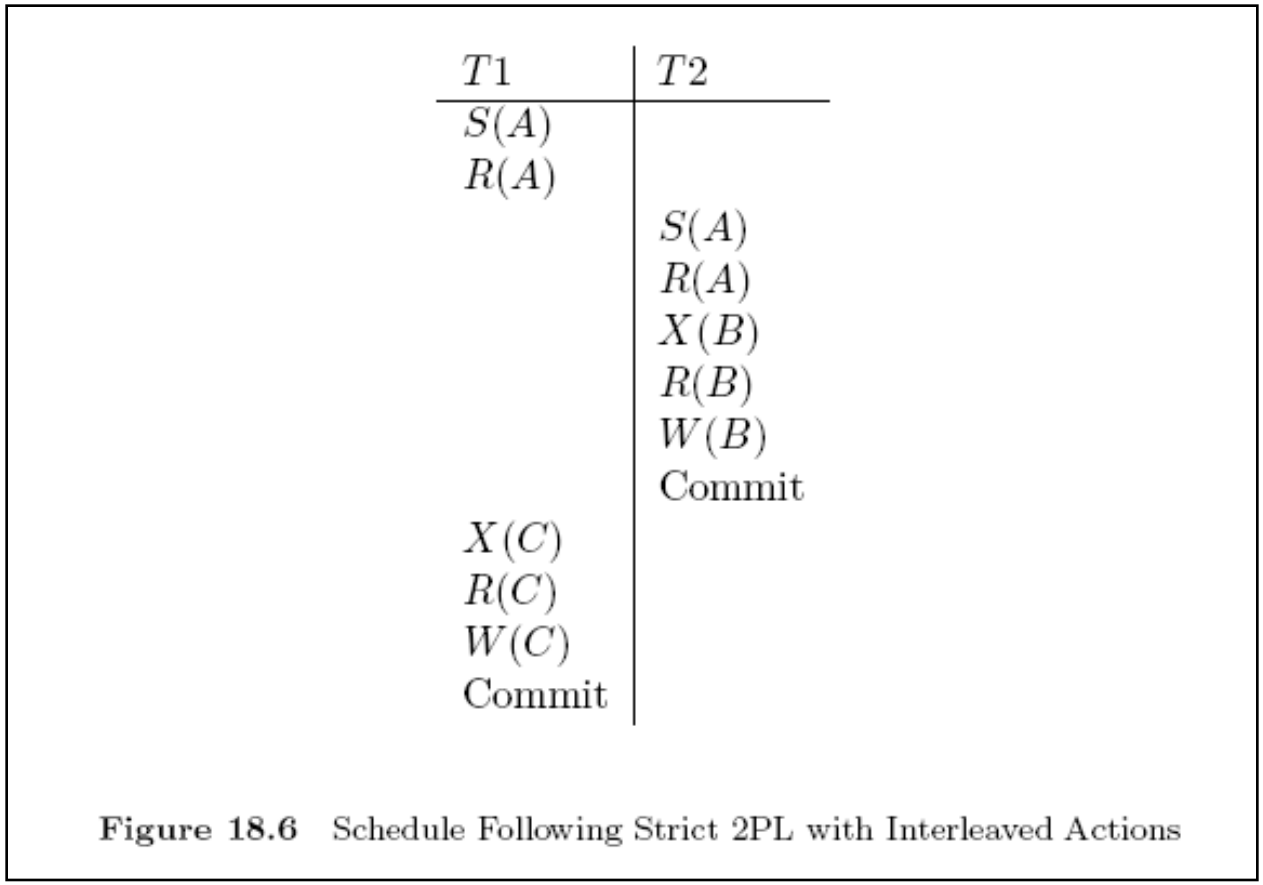
Strict Two-Phase Locking (Strict 2PL).

If the Strict 2PL protocol is used , the previous schedule can be serialized as explain below :

$T1$	$T2$
$X(A)$	
$R(A)$	
$W(A)$	
$X(B)$	
$R(B)$	
$W(B)$	
Commit	
	$X(A)$
	$R(A)$
	$W(A)$
	$X(B)$
	$R(B)$
	$W(B)$
	Commit

Strict Two-Phase Locking (Strict 2PL).

In general, using 2PL, the actions of different transactions could be interleaved.

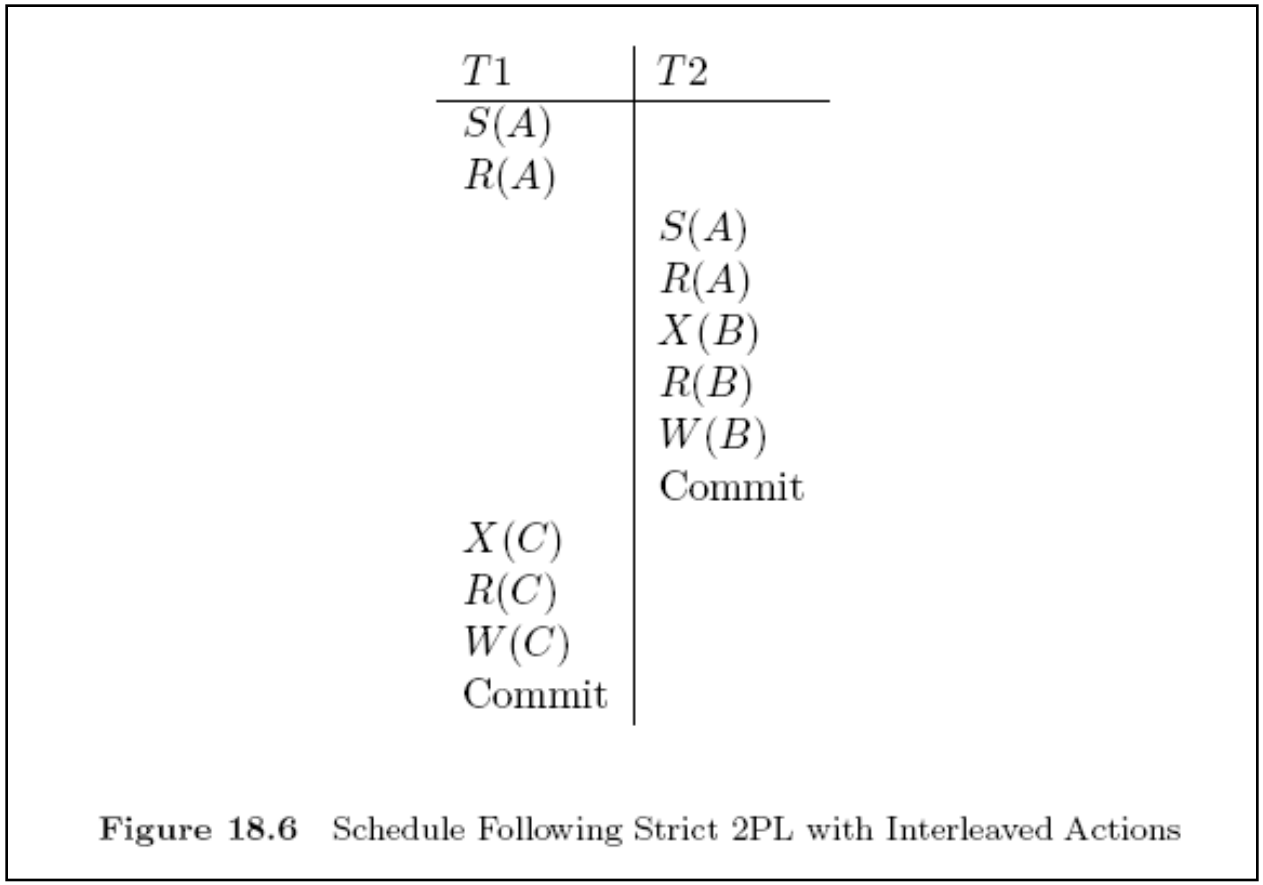


$T1$	$T2$
$S(A)$	
$R(A)$	
	$S(A)$
	$R(A)$
	$X(B)$
	$R(B)$
	$W(B)$
	Commit
$X(C)$	
$R(C)$	
$W(C)$	
Commit	

Figure 18.6 Schedule Following Strict 2PL with Interleaved Actions

Strict Two-Phase Locking (Strict 2PL).

In general, using 2PL, the actions of different transactions could be interleaved.



$T1$	$T2$
$S(A)$	
$R(A)$	
	$S(A)$
	$R(A)$
	$X(B)$
	$R(B)$
	$W(B)$
	Commit
$X(C)$	
$R(C)$	
$W(C)$	
Commit	

Figure 18.6 Schedule Following Strict 2PL with Interleaved Actions

TRANSACTION MANAGEMENT

What are the responsibility of crash recovery manager?

The **recovery manager** of a DBMS is responsible for ensuring transaction atomicity and durability.

1. The recovery manager ensures atomicity by undoing the actions of transactions that do not commit.
2. It ensures durability by making sure that all actions of committed transactions survive system crashes and **media failures**
3. The recovery manager is also responsible for undoing the actions of an aborted transaction.

When a DBMS is restarted after crashes, the recovery manager is given control and must bring the database to a consistent state.

TRANSACTION MANAGEMENT

How is log used by crash recovery manager?

The log enables the recovery manager to undo the actions of aborted and incomplete transactions and to redo the actions of committed transactions.

- A **log** of all modifications to the database is saved on stable storage, which is guaranteed (with very high probability) to survive crashes and media failures.
- Stable storage is implemented by maintaining multiple copies of information (perhaps in different locations) on nonvolatile storage devices such as disks or tapes.

It is important to ensure that the log entries describing a change to the database are written to stable storage before the change is made; otherwise, the system might crash just after the change, leaving us without a record of the change.