

# Transaction Concept

10

- A **Transaction** is a list of actions to perform a single logical unit of work.

or

- A **transaction** is a unit of program execution that accesses and possibly updates various data items.
- Two main issues to deal with:
  - Failures of various kinds, such as hardware failures and system crashes
  - Concurrent execution of multiple transactions
- Access to the DB is done by two operations:
  - Read(x)**
  - Write(x)**

## Transaction Concept (Cont..)

11

- E.g. Transaction to transfer \$50 from account A to account B:

**T1:**

1. **read**(A)
2.  $A := A - 50$
3. **write**(A)
4. **read**(B)
5.  $B := B + 50$
6. **write**(B)

# Transaction State

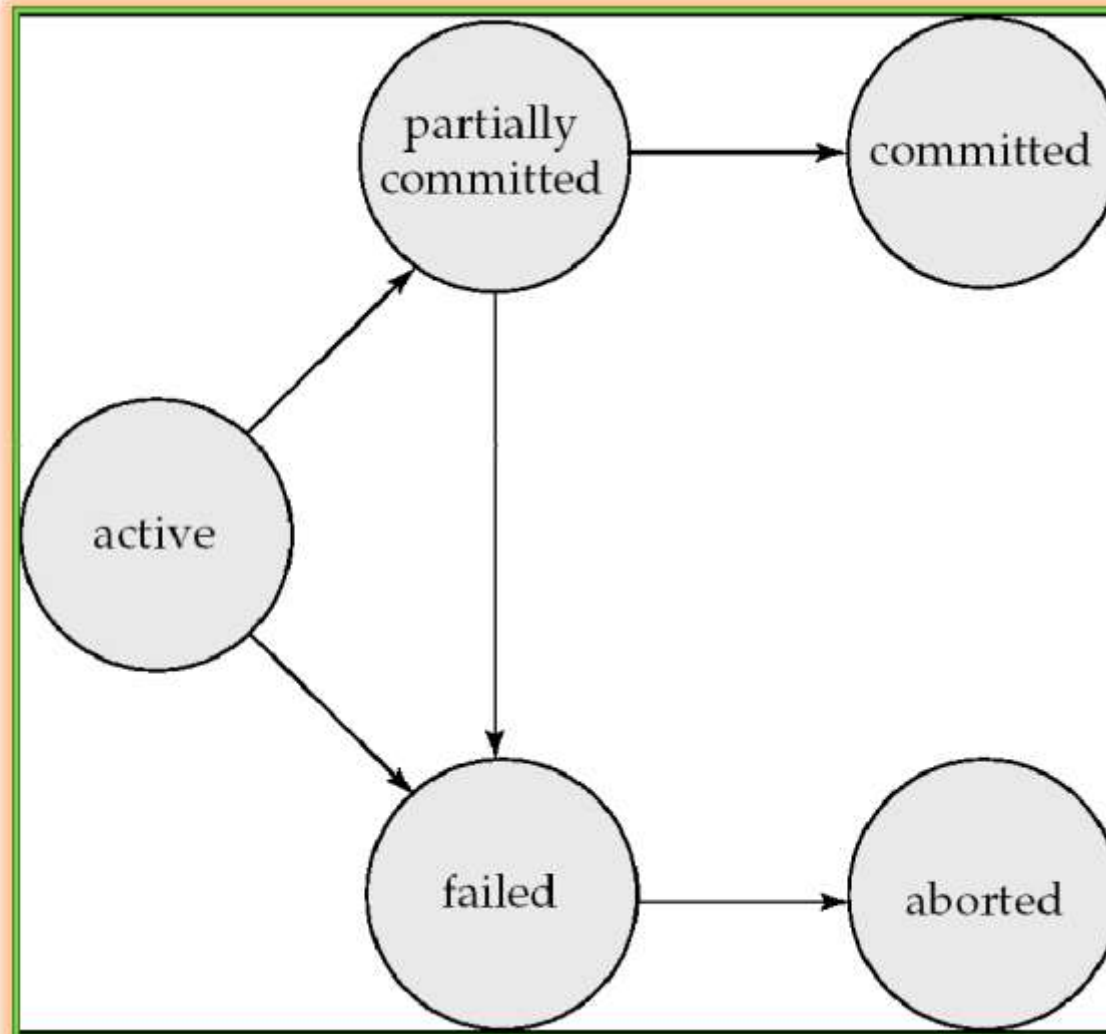
12

- **Active:** The initial state; the transaction stays in this state while it is executing
- **Partially committed:** After the final statement has been executed.
- **Failed:** After the discovery that normal execution can no longer proceed.
- **Aborted:** After the transaction has been rolled back and the database restored to its state prior to the start of the transaction. Two options after it has been aborted:
  - restart the transaction
  - kill the transaction
- **Committed:** After successful completion.

A transaction is said to have *terminated* if it has either *committed* or *aborted*.

## Transaction State (Cont.)

13



# ACID Properties

14

To preserve the integrity of data the database system must ensure:

- **Atomicity:** Either all operations of the transaction are properly reflected in the database or none are.
- **Consistency:** Execution of a transaction in isolation preserves the consistency of the database.
- **Isolation:** Although multiple transactions may execute concurrently, each transaction must be unaware of other concurrently executing transactions. Intermediate transaction results must be hidden from other concurrently executed transactions.
  - That is, for every pair of transactions  $T_i$  and  $T_j$ , it appears to  $T_i$  that either  $T_j$  finished execution before  $T_i$  started, or  $T_j$  started execution after  $T_i$  finished.
- **Durability:** After a transaction completes successfully, the changes it has made to the database persist, even if there are system failures.

# Example of Fund Transfer

15

- Transaction to transfer \$50 from account A to account B:
  1. **read**(A)
  2.  $A := A - 50$
  3. **write**(A)
  4. **read**(B)
  5.  $B := B + 50$
  6. **write**(B)
- **Atomicity requirement**
  - If the transaction fails after step 3 and before step 6, money will be “lost” leading to an inconsistent database state
    - Failure could be due to software or hardware
  - The system should ensure that updates of a partially executed transaction are not reflected in the database
- **Durability requirement** : Once the user has been notified that the transaction has completed (i.e., the transfer of the \$50 has taken place), the updates to the database by the transaction must persist even if there are software or hardware failures.

## Example of Fund Transfer (Cont.)

16

- Transaction to transfer \$50 from account A to account B:
  1. **read**(A)
  2.  $A := A - 50$
  3. **write**(A)
  4. **read**(B)
  5.  $B := B + 50$
  6. **write**(B)
- **Consistency requirement** in above example:
  - The sum of A and B is unchanged by the execution of the transaction
- In general, consistency requirements include
  - A transaction must see a consistent database.
  - During transaction execution the database may be temporarily inconsistent.
  - When the transaction completes successfully the database must be consistent

## Example of Fund Transfer (Cont.)

17

- **Isolation requirement:** If between steps 3 and 6, another transaction T2 is allowed to access the partially updated database, it will see an inconsistent database (the sum  $A + B$  will be less than it should be).

T1	T2
1. <b>read</b> (A)	
2. $A := A - 50$	
3. <b>write</b> (A)	
	read(A), read(B), print(A+B)
4. <b>read</b> (B)	
5. $B := B + 50$	
6. <b>write</b> (B)	

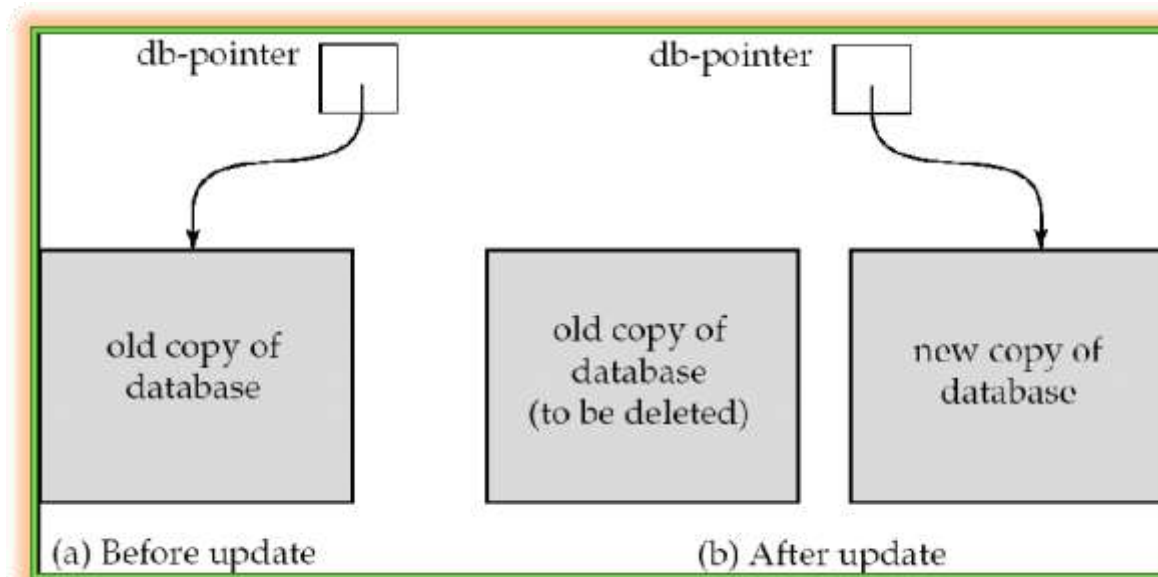
- Isolation can be ensured by running transactions **serially**
  - that is, one after the other.
- However, executing multiple transactions concurrently has significant benefits



# Implementation of Atomicity and Durability

18

- The **recovery-management** component of a database system implements the support for atomicity and durability.
- E.g. the **shadow-database** scheme:
  - all updates are made on a *shadow copy* of the database
    - **db\_pointer** is made to point to the updated shadow copy after
      - the transaction reaches partial commit and
      - all updated pages have been flushed to disk.



# Implementation of Atomicity and Durability (Cont.)

19

- **db\_pointer** always points to the current consistent copy of the database.
  - In case transaction fails, old consistent copy pointed to by **db\_pointer** can be used, and the shadow copy can be deleted.
- The shadow-database scheme:
  - Assumes that only one transaction is active at a time.
  - Assumes disks do not fail
  - Does not handle concurrent transactions

# Concurrent Executions

20

- Multiple transactions are allowed to run concurrently in the system. Advantages are:
  - **Increased processor and disk utilization**, leading to better transaction *throughput*
    - E.g. one transaction can be using the CPU while another is reading from or writing to the disk
  - **Reduced average response time for transactions**: short transactions need not wait behind long ones.
- **Concurrency control schemes**: Mechanisms to achieve isolation
  - that is, to control the interaction among the concurrent transactions in order to prevent them from destroying the consistency of the database

# Schedules

21

- **Schedule:** A sequences of instructions that specify the *chronological order*(i.e. starting with the earliest date and finishing with most recent) in which instructions of concurrent transactions are executed
  - a schedule for a set of transactions must consist of all instructions of those transactions
  - must preserve the order in which the instructions appear in each individual transaction.
- A transaction that successfully completes its execution will have a commit instructions as the last statement
  - by default transaction assumed to execute commit instruction as its last step
- A transaction that fails to successfully complete its execution will have an abort instruction as the last statement

## Schedules (Cont..)

22

- Transaction T1 to transfer \$50 from account A to account B:

**T1**

1. **read**(A)
2.  $A := A - 50$
3. **write**(A)
4. **read**(B)
5.  $B := B + 50$
6. **write**(B)

- Transaction T2 transfers 10% of the balance from account A to account B:

**T2**

1. **read**(A);
2.  $temp := A * 0.1;$
3.  $A := A - temp;$
4. **write**(A);
5. **read**(B);
6.  $B := B + temp;$
7. **write**(B).

# Schedule 1

23

- Let  $T_1$  transfer \$50 from  $A$  to  $B$ , and  $T_2$  transfer 10% of the balance from  $A$  to  $B$ .
- A **serial** schedule in which  $T_1$  is followed by  $T_2$ :

$A := 1000$   $B := 2000$

$A := 950$

$B := 2050$

$T_1$	$T_2$
read( $A$ ) $A := A - 50$ write( $A$ ) read( $B$ ) $B := B + 50$ write( $B$ )	read( $A$ ) $temp := A * 0.1$ $A := A - temp$ write( $A$ ) read( $B$ ) $B := B + temp$ write( $B$ )

$A := 855$

$B := 2145$

$A + B := 855 + 2145 = 3000$

## Schedule 2

24

- A **serial schedule** where  $T_2$  is followed by  $T_1$

	$T_1$	$T_2$	
		read(A)	A: = 1000
		$temp := A * 0.1$	100
		$A := A - temp$	A: = 900
		write(A)	
		read(B)	B: = 2000
		$B := B + temp$	
		write(B)	B: = 2100
A:=900	read(A)		
	$A := A - 50$		
A:=850	write(A)		
B:=2100	read(B)		
	$B := B + 50$		
B:=2150	write(B)		

$$A+B: = 850+2150 = 3000$$

## Schedule 3

25

- Let  $T_1$  and  $T_2$  be the transactions defined previously. The following schedule is **not a serial schedule**, but it is **equivalent** to Schedule 1.

	$T_1$	$T_2$	
A:=1000	read(A)		
	$A := A - 50$		
A:=950	write(A)	read(A)	A:=950
		$temp := A * 0.1$	
		$A := A - temp$	
		write(A)	A:=855
B:=2000	read(B)		
	$B := B + 50$		
B:=2050	write(B)	read(B)	
		$B := B + temp$	
		write(B)	B:=2145

In Schedules 1, 2 and 3, the sum  $A + B$  is preserved.



## Schedule 4

26

- The following concurrent schedule does not preserve the value of  $(A + B)$ .

	$T_1$	$T_2$	
A:=950	read(A) $A := A - 50$	read(A) $temp := A * 0.1$ $A := A - temp$ write(A) read(B)	A:=1000
A:=900 B:=2000	write(A) read(B) $B := B + 50$ write(B)		A:=900 B:=2000
B:=2050		$B := B + temp$ write(B)	B:=2150

$$A+B: = 900+2150 = 3050$$

# Serializability

27

- **Basic Assumption:** Each transaction preserves database consistency.
- Thus serial execution of a set of transactions preserves database consistency.
- A (possibly concurrent) schedule is *Serializable* if it is equivalent to a serial schedule. Different forms of schedule equivalence give rise to the notions of:
  1. **Conflict Serializability**
  2. **View Serializability**

# Conflicting Instructions

28

- Instructions  $I_i$  and  $I_j$  of transactions  $T_i$  and  $T_j$  respectively, **conflict** if and only if there exists some item  $Q$  accessed by both  $I_i$  and  $I_j$ , and at least one of these instructions wrote  $Q$ .
  1.  $I_i = \text{read}(Q)$ ,  $I_j = \text{read}(Q)$ .  $I_i$  and  $I_j$  don't conflict.
  2.  $I_i = \text{read}(Q)$ ,  $I_j = \text{write}(Q)$ . They conflict.
  3.  $I_i = \text{write}(Q)$ ,  $I_j = \text{read}(Q)$ . They conflict
  4.  $I_i = \text{write}(Q)$ ,  $I_j = \text{write}(Q)$ . They conflict
- Intuitively, a conflict between  $I_i$  and  $I_j$  forces a (logical) temporal order between them.
  - If  $I_i$  and  $I_j$  are consecutive in a schedule and they do not conflict, their results would remain the same even if they had been interchanged in the schedule.

# Anamolies due to interleaved execution

29

- WR CONFLICT
- RW CONFLICT
- WW CONFLICT

**WR(READING UNCOMMITTED DATA):**

**DIRTY READ:** T2 COULD READ A DB OBJECT A THAT HAS BEEN JUST MODIFIED BY ANOTHER TRANSCTION T1, WHICH HAS NOT YET COMMITTED, SUCH A READ IS CALLED A DIRTY READ.

**T1**

**read(A)**  
**A := A - 100**  
**write(A)**

**read(B)**  
**B := B + 100**  
**write(B)**  
**COMMIT**

**T2**

**read(A)**  
**A := A + 0.06A**  
**write(A)**  
**read(B)**  
**B := B + 0.06B**  
**write(B)**  
**COMMIT**

## Anamolies due to interleaved execution (Cont..)

30

- **UNREAPTABLE READS(RW)** While the transaction T1 is still in progress, transaction T2 reads the data item A and commits it.

**T1**

R(A)

A:=A+1

W(A)

**T2**

R(A)

A:=A-1

W(A)

COMMIT

- **OVERWRITING UNCOMMITTED DATA(WW):** THE UPDATED VALUE OF ONE TRANSACTION IS OVERWRITTEN BY ANOTHER TRANSACTION. This is called lost update anamoly.

**T1**

R(A)-1000

W(A)-1000

R(B)-1000

W(B)1000

**T2**

R(B)-2000

W(B)-2000

R(A)-2000

W(A)-2000

# Conflict Serializability

31

- If a schedule  $S$  can be transformed into a schedule  $S'$  by a series of swaps of non-conflicting instructions, we say that  $S$  and  $S'$  are **conflict equivalent**.
- We say that a schedule  $S$  is **conflict serializable** if it is conflict equivalent to a serial schedule

## Conflict Serializability (Cont.)

32

- Schedule 3 can be transformed into Schedule 6, a **serial schedule** where  $T_2$  follows  $T_1$ , by series of swaps of **non-conflicting** instructions.
  - Therefore Schedule 3 is **conflict Serializable**.

$T_1$	$T_2$
read(A) write(A)	read(A) write(A)
read(B) write(B)	
	read(B) write(B)

Schedule 3

$T_1$	$T_2$
read(A) write(A) read(B) write(B)	read(A) write(A) read(B) write(B)

Schedule 6

## Conflict Serializability (Cont.)

33

- Example of a schedule that is **not conflict Serializable**:

$T_3$	$T_4$
read( $Q$ )	write( $Q$ )
write( $Q$ )	

- We are unable to swap instructions in the above schedule to obtain either the serial schedule  $\langle T_3, T_4 \rangle$ , or the serial schedule  $\langle T_4, T_3 \rangle$ .



# View Serializability

34

- Let  $S$  and  $S'$  be two schedules with the same set of transactions.  $S$  and  $S'$  are **view equivalent** if the following three conditions are met, for each data item  $Q$ ,
  1. If in schedule  $S$ , transaction  $T_i$  reads the initial value of  $Q$ , then in schedule  $S'$  also transaction  $T_i$  must read the initial value of  $Q$ .
  2. If in schedule  $S$  transaction  $T_i$  executes **read**( $Q$ ), and that value was produced by transaction  $T_j$  (if any), then in schedule  $S'$  also transaction  $T_i$  must read the value of  $Q$  that was produced by the same **write**( $Q$ ) operation of transaction  $T_j$ .
  3. The transaction (if any) that performs the final **write**( $Q$ ) operation in schedule  $S$  must also perform the final **write**( $Q$ ) operation in schedule  $S'$ .

As can be seen, view equivalence is also based purely on **reads** and **writes** alone.

## View Serializability (Cont.)

35

- A schedule  $S$  is **view serializable** if it is view equivalent to a serial schedule.
- Every **conflict serializable** schedule is also **view serializable**.
- Below is a schedule which is **view-serializable** but *not* conflict serializable.

$T_3$	$T_4$	$T_6$
read( $Q$ )	write( $Q$ )	
write( $Q$ )		
		write( $Q$ )

- What serial schedule is above equivalent to?
- Every view serializable schedule that is not conflict serializable has **blind writes**.

## Other Notions of Serializability

36

- The schedule below produces same outcome as the serial schedule  $\langle T_1, T_5 \rangle$ , yet is not conflict equivalent or view equivalent to it.

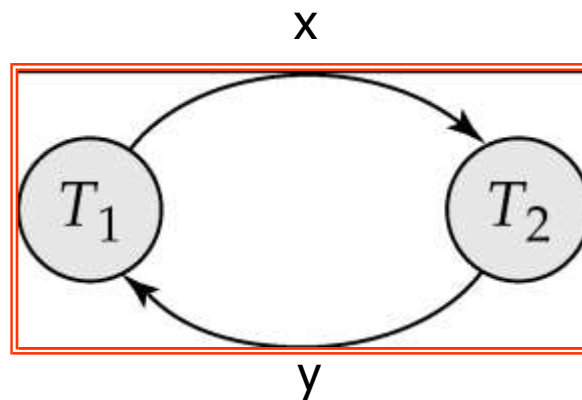
$T_1$	$T_5$
read(A) $A := A - 50$ write(A)	
	read(B) $B := B - 10$ write(B)
read(B) $B := B + 50$ write(B)	
	read(A) $A := A + 10$ write(A)

- Determining such equivalence requires analysis of operations other than read and write.

# Testing for Serializability

37

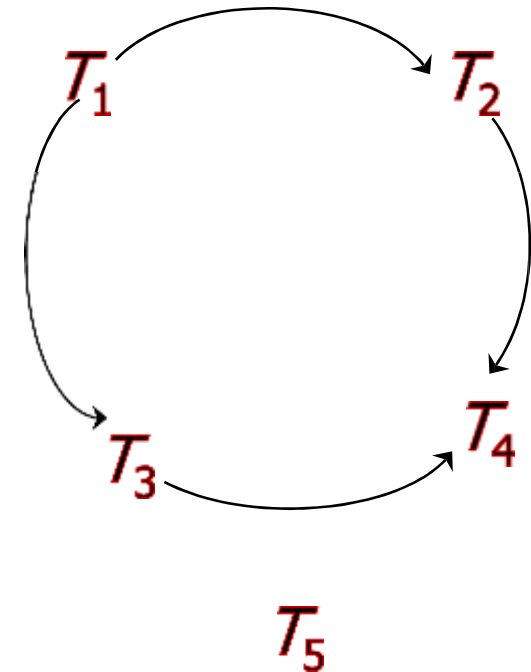
- Consider some schedule of a set of transactions  $T_1, T_2, \dots, T_n$
- **Precedence graph:** A direct graph where the vertices are the transactions (names).
- We draw an arc from  $T_i$  to  $T_j$  if the two transaction conflict, and  $T_i$  accessed the data item on which the conflict arose earlier.
- We may label the arc by the item that was accessed.
- **Example 1**



## Example Schedule (Schedule A) + Precedence Graph

38

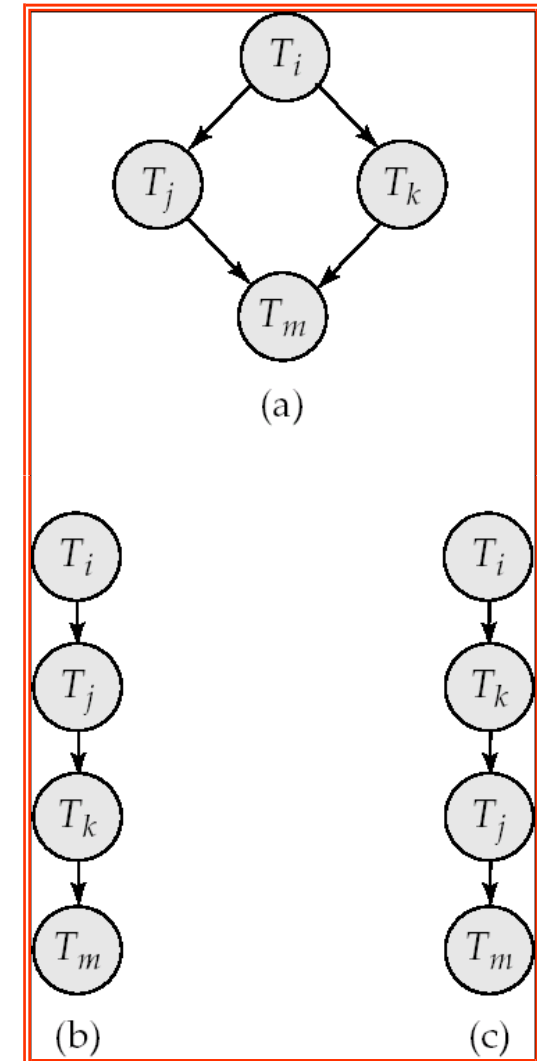
$T_1$	$T_2$	$T_3$	$T_4$	$T_5$
read(Y) read(Z)	read(X)			read(V) read(W) read(W)
	read(Y) write(Y)	write(Z)		
read(U)			read(Y) write(Y) read(Z) write(Z)	
read(U) write(U)				



# Test for Conflict Serializability

39

- A schedule is **conflict Serializable** if and only if its precedence graph is acyclic.
- Cycle-detection algorithms exist which take order  $n^2$  time, where  $n$  is the number of vertices in the graph.
  - (Better algorithms take order  $n + e$  where  $e$  is the number of edges.)
- If precedence graph is acyclic, the Serializability order can be obtained by a **topological sorting** of the graph.
  - This is a linear order consistent with the partial order of the graph.
  - For example, a Serializability order for Schedule A would be  
 $T_5 \rightarrow T_1 \rightarrow T_3 \rightarrow T_2 \rightarrow T_4$ 
    - Are there others?



# Test for View Serializability

40

- The precedence graph test for conflict serializability cannot be used directly to test for **view serializability**.
  - Extension to test for **view serializability** has cost exponential in the size of the precedence graph.
- The problem of checking if a schedule is **view serializable** falls in the class of *NP*-complete problems.
  - Thus existence of an efficient algorithm is *extremely* unlikely.
- However practical algorithms that just check some **sufficient conditions** for **view serializability** can still be used.

# Recoverable Schedules

41

Need to address the effect of transaction failures on concurrently running transactions.

- **Recoverable schedule:** If a transaction  $T_i$  reads a data item previously written by a transaction  $T_j$ , then the commit operation of  $T_i$  appears before the commit operation of  $T_j$ .
- The following schedule (Schedule 11) is not recoverable if  $T_9$  commits immediately after the read

$T_8$	$T_9$
read(A)	
write(A)	
	read(A)
read(B)	

- If  $T_8$  should abort,  $T_9$  would have read (and possibly shown to the user) an inconsistent database state. Hence, database must ensure that schedules are recoverable.



# Cascading Rollbacks

42

- **Cascading rollback:** A single transaction failure leads to a series of transaction rollbacks. Consider the following schedule where none of the transactions has yet committed (so the schedule is recoverable)

$T_{10}$	$T_{11}$	$T_{12}$
read(A) read(B) write(A)	read(A) write(A)	read(A)

If  $T_{10}$  fails,  $T_{11}$  and  $T_{12}$  must also be rolled back.

- Can lead to the undoing of a significant amount of work

# Cascadeless Schedules

43

- **Cascadeless schedules:** Cascading rollbacks cannot occur; for each pair of transactions  $T_i$  and  $T_j$  such that  $T_j$  reads a data item previously written by  $T_i$ , the commit operation of  $T_i$  appears before the read operation of  $T_j$ .
- Every cascadeless schedule is also recoverable
- It is desirable to restrict the schedules to those that are cascadeless

# Concurrency Control

44

- A database must provide a mechanism that will ensure that all possible schedules are
  - either conflict or view serializable, and
  - are recoverable and preferably cascadeless
- A policy in which only one transaction can execute at a time generates serial schedules, but provides a poor degree of concurrency
  - Are serial schedules recoverable/cascadeless?
- Testing a schedule for serializability *after* it has executed is a little too late!
- **Goal** – to develop concurrency control protocols that will assure serializability.

# Concurrency Control vs. Serializability Tests

45

- **Concurrency-control protocols** allow concurrent schedules, but ensure that the schedules are conflict/view serializable, and are recoverable and cascadeless .
- **Concurrency control protocols** generally do not examine the precedence graph as it is being created
  - Instead a protocol imposes a discipline that avoids nonserializable schedules.
  - We study such protocols in Chapter 16.
- Different **concurrency control protocols** provide different tradeoffs between the amount of concurrency they allow and the amount of overhead that they incur.
- Tests for serializability help us understand why a **concurrency control protocol** is correct.

# Weak Levels of Consistency

46

- Some applications are willing to live with **weak levels of consistency**, allowing schedules that are not Serializable
  - E.g. a read-only transaction that wants to get an approximate total balance of all accounts
  - E.g. database statistics computed for query optimization can be approximate (why?)
  - Such transactions need not be serializable with respect to other transactions
- Tradeoff accuracy for performance

# Levels of Consistency in SQL-92

47

- **Serializable:** default
  - **Repeatable read:** only committed records to be read, repeated reads of same record must return same value. However, a transaction may not be serializable - it may find some records inserted by a transaction but not find others.
  - **Read committed:** only committed records can be read, but successive reads of record may return different (but committed) values.
  - **Read uncommitted:** even uncommitted records may be read.
- 
- Lower degrees of consistency useful for gathering approximate information about the database
  - **Warning:** some database systems do not ensure serializable schedules by default
    - ❖ **E.g.** Oracle and PostgreSQL by default support a level of consistency called snapshot isolation (not part of the SQL standard)

# Implementation of Isolation

48

- ❑ Schedules must be conflict or view serializable, and recoverable, for the sake of database consistency, and preferably cascadeless.
- ❑ A policy in which only one transaction can execute at a time generates serial schedules, but provides a poor degree of concurrency.
- ❑ Concurrency-control schemes tradeoff between the amount of concurrency they allow and the amount of overhead that they incur.
- ❑ Some schemes allow only conflict-serializable schedules to be generated, while others allow view-serializable schedules that are not conflict-serializable.

# Transaction Definition in SQL

49

- **Data manipulation language** must include a construct for specifying the set of actions that comprise a transaction.
- In SQL, a transaction begins implicitly.
- A transaction in **SQL** ends by:
  - **Commit work** commits current transaction and begins a new one.
  - **Rollback work** causes current transaction to abort.
- In almost all database systems, by default, every **SQL** statement also **commits** implicitly if it executes successfully
  - **Implicit commit** can be turned off by a database directive
    - E.g. in JDBC, `connection.setAutoCommit(false);`



# DATABASE MANAGEMENT SYSTEMS

50

## UNIT – V

❖ **TRANSACTIONS MANAGEMENT**

❖ **CONCURRENCY CONTROL AND RECOVERY SYSTEM**

# UNIT – V (Part - 2)

51

## CONCURRENCY CONTROL

-  Lock-Based Protocols
-  Timestamp-Based Protocols
-  Validation-Based Protocols
-  Multiple Granularity and Deadlock handling
-  Multiversion Schemes
-  Insert and Delete Operations
-  Concurrency in Index Structures

# Lock-Based Protocols

52

- A lock is a mechanism to control concurrent access to a data item
- Data items can be locked in two modes :
  1. *exclusive* (X) mode. Data item can be both read as well as written. X-lock is requested using **lock-X** instruction.
  2. *shared* (S) mode. Data item can only be read. S-lock is requested using **lock-S** instruction.
- Lock requests are made to concurrency-control manager.  
Transaction can proceed only after request is granted.

## Lock-Based Protocols (Cont.)

53

### □ Lock-compatibility matrix

	S	X
S	true	false
X	false	false

- A transaction may be granted a lock on an item if the requested lock is compatible with locks already held on the item by other transactions
- Any number of transactions can hold shared locks on an item,
  - but if any transaction holds an exclusive on the item no other transaction may hold any lock on the item.
- If a lock cannot be granted, the requesting transaction is made to wait till all incompatible locks held by other transactions have been released. The lock is then granted.

## Lock-Based Protocols (Cont.)

54

- Example of a transaction performing locking:

```
 $T_2$ : lock-S(A);  
      read (A);  
      unlock(A);  
      lock-S(B);  
      read (B);  
      unlock(B);  
      display(A+B)
```

- Locking as above is not sufficient to guarantee Serializability - if  $A$  and  $B$  get updated in-between the read of  $A$  and  $B$ , the displayed sum would be wrong.
- A **locking protocol** is a set of rules followed by all transactions while requesting and releasing locks. Locking protocols restrict the set of possible schedules.

# Pitfalls of Lock-Based Protocols

55

- Consider the partial schedule

$T_3$	$T_4$
lock-X( $B$ )	
read( $B$ )	
$B := B - 50$	
write( $B$ )	
	lock-S( $A$ )
	read( $A$ )
	lock-S( $B$ )
lock-X( $A$ )	

- Neither  $T_3$  nor  $T_4$  can make progress - executing **lock-S( $B$ )** causes  $T_4$  to wait for  $T_3$  to release its lock on  $B$ , while executing **lock-X( $A$ )** causes  $T_3$  to wait for  $T_4$  to release its lock on  $A$ .
- Such a situation is called a **deadlock**.
  - To handle a deadlock one of  $T_3$  or  $T_4$  must be rolled back and its locks released.

## Pitfalls of Lock-Based Protocols (Cont.)

56

- The potential for deadlock exists in most locking protocols. Deadlocks are a necessary evil.
- **Starvation** is also possible if concurrency control manager is badly designed. For example:
  - A transaction may be waiting for an X-lock on an item, while a sequence of other transactions request and are granted an S-lock on the same item.
  - The same transaction is repeatedly rolled back due to deadlocks.
- Concurrency control manager can be designed to prevent starvation.

# The Two-Phase Locking Protocol

57

- This is a protocol which ensures conflict-serializable schedules.
- **Phase 1: Growing Phase**
  - transaction may obtain locks
  - transaction may not release locks
- **Phase 2: Shrinking Phase**
  - transaction may release locks
  - transaction may not obtain locks
- The protocol assures serializability. It can be proved that the transactions can be serialized in the order of their **lock points** (i.e. the point where a transaction acquired its final lock).



## The Two-Phase Locking Protocol (Cont.)

58

- Two-phase locking *does not* ensure freedom from deadlocks
- Cascading roll-back is possible under two-phase locking. To avoid this, follow a modified protocol called **strict two-phase locking**. Here a transaction must hold all its exclusive locks till it commits/aborts.
- **Rigorous two-phase locking** is even stricter: here *all* locks are held till commit/abort. In this protocol transactions can be serialized in the order in which they commit.

## The Two-Phase Locking Protocol (Cont.)

59

- There can be conflict serializable schedules that cannot be obtained if two-phase locking is used.
- However, in the absence of extra information (e.g., ordering of access to data), two-phase locking is needed for conflict serializability in the following sense:

Given a transaction  $T_i$  that does not follow two-phase locking, we can find a transaction  $T_j$  that uses two-phase locking, and a schedule for  $T_i$  and  $T_j$  that is not conflict serializable.

# Lock Conversions

60

- Two-phase locking with lock conversions:
  - First Phase:
    - can acquire a lock-S on item
    - can acquire a lock-X on item
    - can convert a lock-S to a lock-X (upgrade)
  - Second Phase:
    - can release a lock-S
    - can release a lock-X
    - can convert a lock-X to a lock-S (downgrade)
- This protocol assures **serializability**. But still relies on the programmer to insert the various locking instructions.

# Automatic Acquisition of Locks

61

- A transaction  $T_i$  issues the standard read/write instruction, without explicit locking calls.
- The operation **read**( $D$ ) is processed as:

**if**  $T_i$  has a lock on  $D$

**then**

read( $D$ )

**else begin**

if necessary wait until no other  
transaction has a **lock-X** on  $D$

grant  $T_i$  a **lock-S** on  $D$ ;

read( $D$ )

**end**

## Automatic Acquisition of Locks (Cont.)

62

- **write**( $D$ ) is processed as:
  - if**  $T_i$  has a **lock-X** on  $D$ 
    - then**
      - write( $D$ )
    - else begin**
      - if necessary wait until no other trans. has any lock on  $D$ ,
      - if  $T_i$  has a **lock-S** on  $D$ 
        - then**
          - upgrade** lock on  $D$  to **lock-X**
        - else**
          - grant  $T_i$  a **lock-X** on  $D$
      - write( $D$ )
    - end;**
  - All locks are released after commit or abort

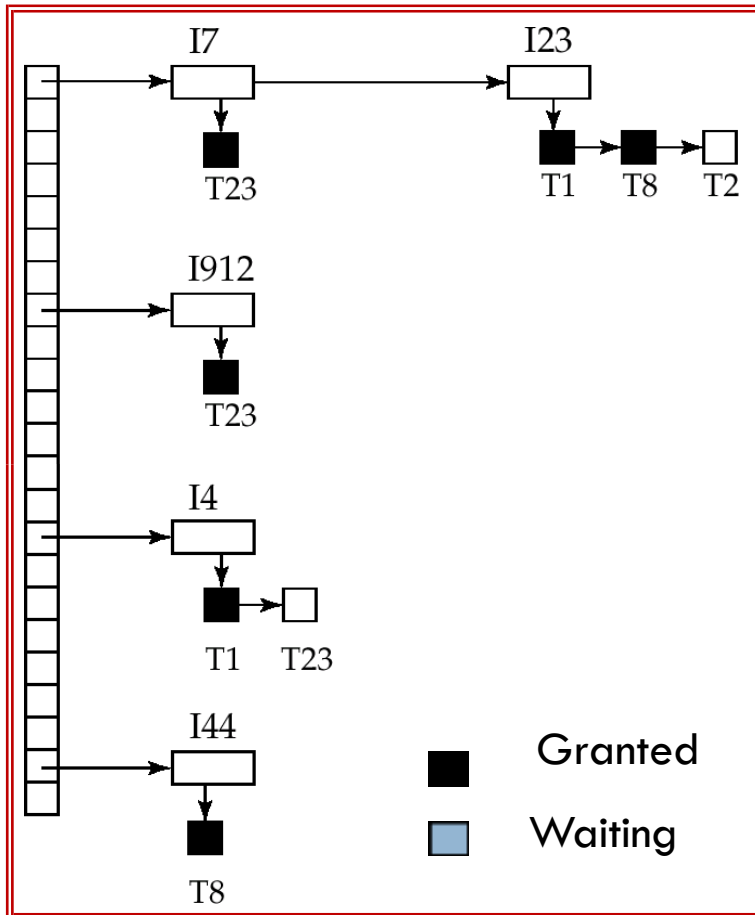
# Implementation of Locking

63

- A **lock manager** can be implemented as a separate process to which transactions send lock and unlock requests
- The **lock manager** replies to a lock request by sending a lock grant messages (or a message asking the transaction to roll back, in case of a deadlock)
- The requesting transaction waits until its request is answered
- The **lock manager** maintains a data-structure called a **lock table** to record granted locks and pending requests
- The **lock table** is usually implemented as an in-memory hash table indexed on the name of the data item being locked

# Lock Table

64



- Black rectangles indicate granted locks, white ones indicate waiting requests
- Lock table also records the type of lock granted or requested
- New request is added to the end of the queue of requests for the data item, and granted if it is compatible with all earlier locks
- Unlock requests result in the request being deleted, and later requests are checked to see if they can now be granted
- If transaction aborts, all waiting or granted requests of the transaction are deleted
  - lock manager may keep a list of locks held by each transaction, to implement this efficiently

# Graph-Based Protocols

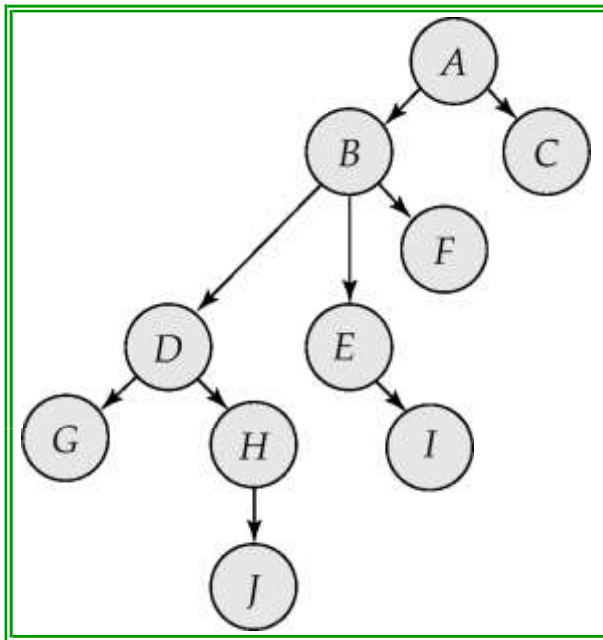
65

- Graph-based protocols are an alternative to two-phase locking
- Impose a partial ordering  $\rightarrow$  on the set  $\mathbf{D} = \{d_1, d_2, \dots, d_h\}$  of all data items.
  - If  $d_i \rightarrow d_j$  then any transaction accessing both  $d_i$  and  $d_j$  must access  $d_i$  before accessing  $d_j$ .
  - Implies that the set  $\mathbf{D}$  may now be viewed as a directed acyclic graph, called a *database graph*.
- The *tree-protocol* is a simple kind of graph protocol.



# Tree Protocol

66



1. Only **exclusive locks** are allowed.
2. The first lock by  $T_i$  may be on any data item. Subsequently, a data  $Q$  can be locked by  $T_i$  only if the parent of  $Q$  is currently locked by  $T_i$ .
3. Data items may be unlocked at any time.
4. A data item that has been locked and unlocked by  $T_i$  cannot subsequently be relocked by  $T_i$ .

# Graph - Based Protocols (Cont.)

67

- The tree protocol ensures **conflict Serializability** as well as freedom from deadlock.
- Unlocking may occur earlier in the tree-locking protocol than in the **two-phase locking protocol**.
  - shorter waiting times, and increase in concurrency
  - protocol is deadlock-free, no rollbacks are required
- **Drawbacks**
  - Protocol does not guarantee recoverability or cascade freedom
    - Need to introduce commit dependencies to ensure recoverability
  - Transactions may have to lock data items that they do not access.
    - increased locking overhead, and additional waiting time
    - potential decrease in concurrency
- Schedules not possible under **two-phase locking** are possible under tree protocol, and vice versa.

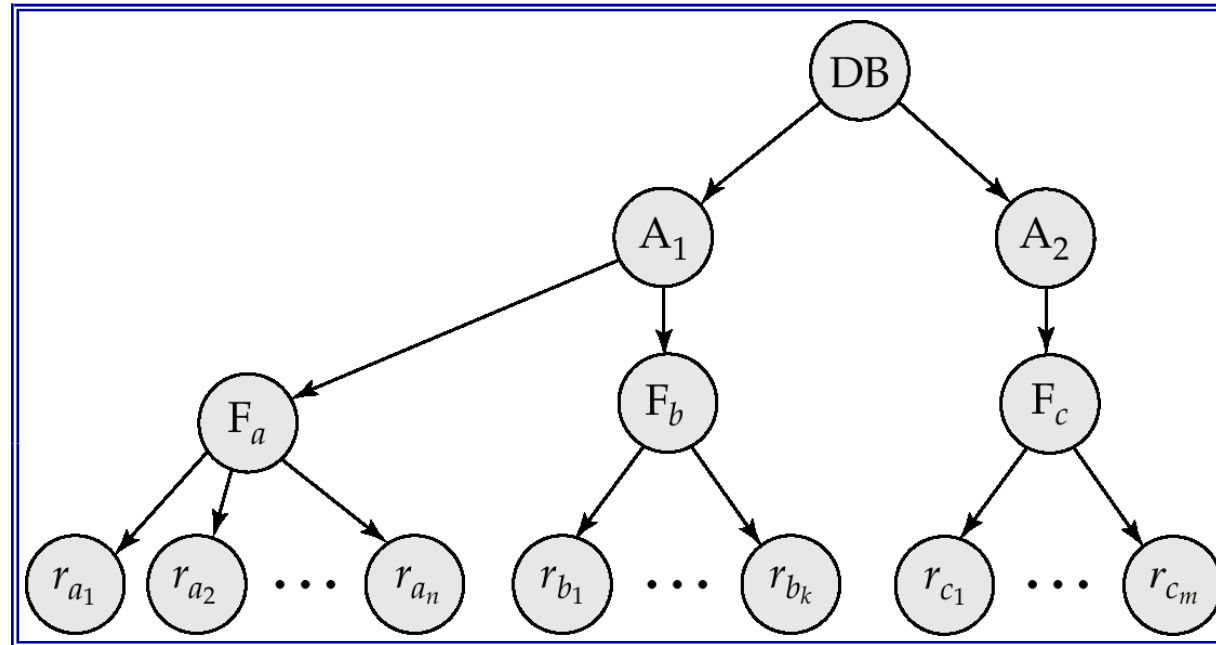
# Multiple Granularity

68

- Allow data items to be of various sizes and define a hierarchy of data granularities, where the small granularities are nested within larger ones
- Can be represented graphically as a tree (but don't confuse with tree-locking protocol)
- When a transaction locks a node in the tree *explicitly*, it *implicitly* locks all the node's descendents in the same mode.
- **Granularity of locking** (level in tree where locking is done):
  - **fine granularity** (lower in tree): high concurrency, high locking overhead
  - **coarse granularity** (higher in tree): low locking overhead, low concurrency

# Example of Granularity Hierarchy

69



The levels, starting from the coarsest (top) level are

- *database*
- *area*
- *file*
- *record*

# Intention Lock Modes

70

- In addition to **S** and **X** lock modes, there are three additional lock modes with multiple granularity:
  - **intention-shared (IS)**: indicates explicit locking at a lower level of the tree but only with shared locks.
  - **intention-exclusive (IX)**: indicates explicit locking at a lower level with exclusive or shared locks
  - **shared and intention-exclusive (SIX)**: the subtree rooted by that node is locked explicitly in shared mode and explicit locking is being done at a lower level with exclusive-mode locks.
- **intention locks** allow a higher level node to be locked in **S** or **X** mode without having to check all descendent nodes.

## Compatibility Matrix with Intention Lock Modes

71

- The compatibility matrix for all lock modes is:

	IS	IX	S	S IX	X
IS	✓	✓	✓	✓	×
IX	✓	✓	×	×	×
S	✓	×	✓	×	×
S IX	✓	×	×	×	×
X	×	×	×	×	×

# Multiple Granularity Locking Scheme

72

- Transaction  $T_i$  can lock a node  $Q$ , using the following rules:
  1. The lock compatibility matrix must be observed.
  2. The root of the tree must be locked first, and may be locked in any mode.
  3. A node  $Q$  can be locked by  $T_i$  in  $S$  or  $IS$  mode only if the parent of  $Q$  is currently locked by  $T_i$  in either  $IX$  or  $IS$  mode.
  4. A node  $Q$  can be locked by  $T_i$  in  $X$ ,  $SIX$ , or  $IX$  mode only if the parent of  $Q$  is currently locked by  $T_i$  in either  $IX$  or  $SIX$  mode.
  5.  $T_i$  can lock a node only if it has not previously unlocked any node (that is,  $T_i$  is two-phase).
  6.  $T_i$  can unlock a node  $Q$  only if none of the children of  $Q$  are currently locked by  $T_i$ .
- Observe that locks are acquired in root-to-leaf order, whereas they are released in leaf-to-root order.

# Deadlock Handling

73

- Consider the following two transactions:

$T_1$ :    write ( $X$ )  
          write( $Y$ )

$T_2$ :    write( $Y$ )  
          write( $X$ )

- Schedule with deadlock

1	2
<b>lock-X</b> on $X$ write ( $X$ )  wait for <b>lock-X</b> on $Y$	<b>lock-X</b> on $Y$ write ( $X$ ) wait for <b>lock-X</b> on $X$



# Deadlock Handling

74

- System is deadlocked if there is a set of transactions such that every transaction in the set is waiting for another transaction in the set.
- **Deadlock prevention** protocols ensure that the system will *never* enter into a deadlock state. Some prevention strategies :
  - Require that each transaction locks all its data items before it begins execution (**predeclaration**).
  - Impose partial ordering of all data items and require that a transaction can lock data items only in the order specified by the partial order (**graph-based protocol**).

# More Deadlock Prevention Strategies

75

- Following schemes use transaction timestamps for the sake of deadlock prevention alone.
- **wait-die** scheme - non-preemptive
  - older transaction may wait for younger one to release data item. Younger transactions never wait for older ones; they are rolled back instead.
  - a transaction may die several times before acquiring needed data item
- **wound-wait** scheme - preemptive
  - older transaction *wounds* (forces rollback) of younger transaction instead of waiting for it. Younger transactions may wait for older ones.
  - may be fewer rollbacks than *wait-die* scheme.

## Deadlock prevention (Cont.)

76

- Both in *wait-die* and in *wound-wait* schemes, a rolled back transactions is restarted with its original timestamp. Older transactions thus have precedence over newer ones, and starvation is hence avoided.
- **Timeout-Based Schemes :**
  - a transaction waits for a lock only for a specified amount of time. After that, the wait times out and the transaction is rolled back.
  - thus deadlocks are not possible
  - simple to implement; but starvation is possible. Also difficult to determine good value of the timeout interval.

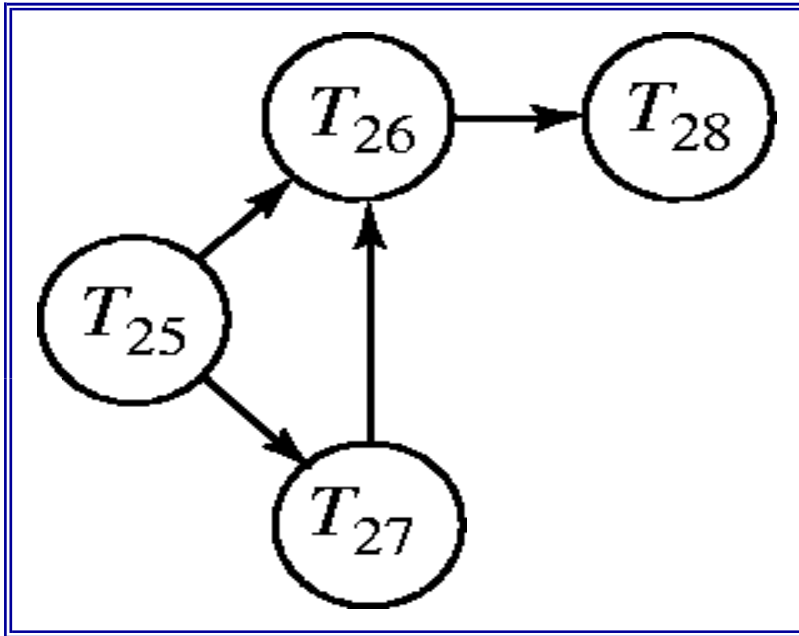
# Deadlock Detection

77

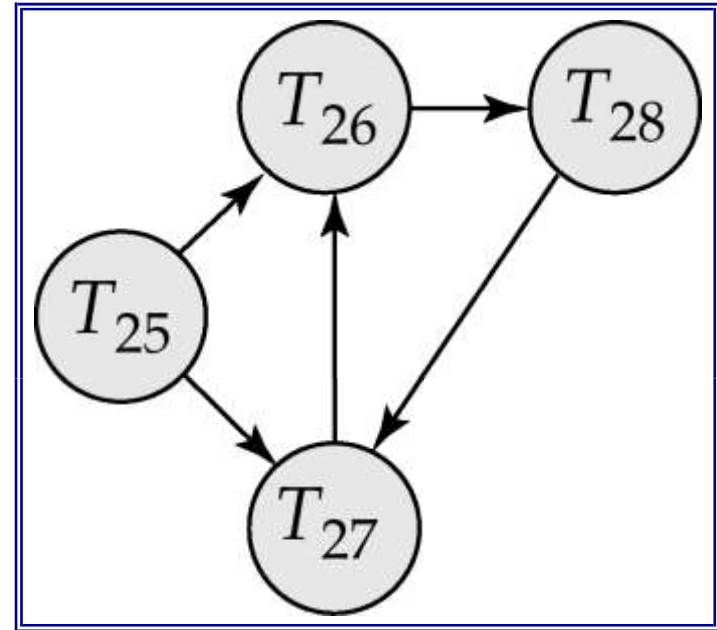
- Deadlocks can be described as a *wait-for graph*, which consists of a pair  $G = (V, E)$ ,
  - $V$  is a set of vertices (all the transactions in the system)
  - $E$  is a set of edges; each element is an ordered pair  $T_i \rightarrow T_j$ .
- If  $T_i \rightarrow T_j$  is in  $E$ , then there is a directed edge from  $T_i$  to  $T_j$ , implying that  $T_i$  is waiting for  $T_j$  to release a data item.
- When  $T_i$  requests a data item currently being held by  $T_j$ , then the edge  $T_i T_j$  is inserted in the wait-for graph. This edge is removed only when  $T_j$  is no longer holding a data item needed by  $T_i$ .
- The system is in a **deadlock state** if and only if the **wait-for graph has a cycle**. Must invoke a **deadlock-detection algorithm** periodically to look for cycles.

## Deadlock Detection (Cont.)

78



Wait-for graph without a cycle



Wait-for graph with a cycle

# Deadlock Recovery

79

- When deadlock is detected :
  - Some transaction will have to rolled back (made a victim) to break deadlock. Select that transaction as victim that will incur minimum cost.
  - Rollback -- determine how far to roll back transaction
    - Total rollback: Abort the transaction and then restart it.
    - More effective to roll back transaction only as far as necessary to break deadlock.
  - Starvation happens if same transaction is always chosen as victim. Include the number of rollbacks in the cost factor to avoid starvation

# Timestamp-Based Protocols

80

- Each transaction is issued a timestamp when it enters the system. If an old transaction  $T_i$  has **time-stamp**  $TS(T_i)$ , a new transaction  $T_j$  is assigned time-stamp  $TS(T_j)$  such that  $TS(T_i) < TS(T_j)$ .
- The protocol manages concurrent execution such that the **time-stamps** determine the serializability order.
- In order to assure such behavior, the protocol maintains for each data  $Q$  two timestamp values:
  - **W-timestamp**( $Q$ ) is the largest time-stamp of any transaction that executed **write**( $Q$ ) successfully.
  - **R-timestamp**( $Q$ ) is the largest time-stamp of any transaction that executed **read**( $Q$ ) successfully.

## Timestamp-Based Protocols (Cont.)

81

- The timestamp ordering protocol ensures that any conflicting **read** and **write** operations are executed in timestamp order.
- Suppose a transaction  $T_i$  issues a **read**(Q)
  1. If  $TS(T_i) \leq W\text{-timestamp}(Q)$ , then  $T_i$  needs to read a value of Q that was already overwritten.
    - Hence, the **read** operation is rejected, and  $T_i$  is rolled back.
  2. If  $TS(T_i) \geq W\text{-timestamp}(Q)$ , then the **read** operation is executed, and  $R\text{-timestamp}(Q)$  is set to  $\max(R\text{-timestamp}(Q), TS(T_i))$ .



## Timestamp-Based Protocols (Cont.)

82

- Suppose that transaction  $T_i$  issues **write**(Q).
  1. If  $TS(T_i) < R\text{-timestamp}(Q)$ , then the value of Q that  $T_i$  is producing was needed previously, and the system assumed that that value would never be produced.
    - Hence, the **write** operation is rejected, and  $T_i$  is rolled back.
  2. If  $TS(T_i) < W\text{-timestamp}(Q)$ , then  $T_i$  is attempting to write an obsolete value of Q.
    - Hence, this **write** operation is rejected, and  $T_i$  is rolled back.
  3. Otherwise, the **write** operation is executed, and  $W\text{-timestamp}(Q)$  is set to  $TS(T_i)$ .

## Example Use of the Protocol

83

A partial schedule for several data items for transactions with timestamps 1, 2, 3, 4, 5

$T_1$	$T_2$	$T_3$	$T_4$	$T_5$
read( $Y$ )	read( $Y$ )	write( $Y$ ) write( $Z$ )		read( $X$ )
read( $X$ )	read( $X$ ) abort	write( $Z$ ) abort		read( $Z$ )
				write( $Y$ ) write( $Z$ )

# Correctness of Timestamp-Ordering Protocol

84

- The **timestamp-ordering protocol** guarantees Serializability since all the arcs in the **precedence graph** are of the form:



Thus, there will be no cycles in the precedence graph

- **Timestamp protocol** ensures freedom from deadlock as no transaction ever waits.
- But the schedule may not be **cascade-free**, and may not even be recoverable.

# Recoverability and Cascade Freedom

85

- **Problem with timestamp-ordering protocol:**
  - Suppose  $T_i$  aborts, but  $T_j$  has read a data item written by  $T_i$
  - Then  $T_j$  must abort; if  $T_j$  had been allowed to commit earlier, the schedule is not recoverable.
  - Further, any transaction that has read a data item written by  $T_j$  must abort
  - This can lead to cascading rollback --- that is, a chain of rollbacks
- **Solution 1:**
  - A transaction is structured such that its writes are all performed at the end of its processing
  - All writes of a transaction form an atomic action; no transaction may execute while a transaction is being written
  - A transaction that aborts is restarted with a new timestamp
- **Solution 2:** Limited form of locking: wait for data to be committed before reading it
- **Solution 3:** Use commit dependencies to ensure recoverability

# Thomas' Write Rule

86

- Modified version of the **timestamp-ordering protocol** in which obsolete **write** operations may be ignored under certain circumstances.
- When  $T_i$  attempts to write data item  $Q$ , if  $TS(T_i) < W\text{-timestamp}(Q)$ , then  $T_i$  is attempting to write an obsolete value of  $\{Q\}$ .
  - Rather than rolling back  $T_i$  as the timestamp ordering protocol would have done, this **{write}** operation can be ignored.
- Otherwise this protocol is the same as the **timestamp ordering protocol**.
- **Thomas' Write Rule** allows greater potential concurrency.
  - Allows some **view-serializable** schedules that are not conflict-serializable.

# Validation-Based Protocol

87

- Execution of transaction  $T_i$  is done in three phases.
  1. **Read and execution phase:** Transaction  $T_i$  writes only to temporary local variables
  2. **Validation phase:** Transaction  $T_i$  performs a ``validation test" to determine if local variables can be written without violating serializability.
  3. **Write phase:** If  $T_i$  is validated, the updates are applied to the database; otherwise,  $T_i$  is rolled back.
- The three phases of concurrently executing transactions can be interleaved, but each transaction must go through the three phases in that order.
  - Assume for simplicity that the validation and write phase occur together, atomically and serially
    - I.e., only one transaction executes validation/write at a time.
- Also called as **optimistic concurrency control** since transaction executes fully in the hope that all will go well during validation

## Validation-Based Protocol (Cont.)

88

- Each transaction  $T_i$  has 3 timestamps
  - $\text{Start}(T_i)$  : the time when  $T_i$  started its execution
  - $\text{Validation}(T_i)$ : the time when  $T_i$  entered its validation phase
  - $\text{Finish}(T_i)$  : the time when  $T_i$  finished its write phase
- Serializability order is determined by timestamp given at validation time, to increase concurrency.
  - Thus  $\text{TS}(T_i)$  is given the value of  $\text{Validation}(T_i)$ .
- This protocol is useful and gives greater degree of concurrency if probability of conflicts is low.
  - because the Serializability order is not pre-decided, and
  - relatively few transactions will have to be rolled back.

# Validation Test for Transaction $T_j$

89

- If for all  $T_i$  with  $TS(T_i) < TS(T_j)$  either one of the following condition holds:
  - **finish**( $T_i$ ) < **start**( $T_j$ )
  - **start**( $T_j$ ) < **finish**( $T_i$ ) < **validation**( $T_j$ ) and the set of data items written by  $T_j$  does not intersect with the set of data items read by  $T_i$ .

then validation succeeds and  $T_j$  can be committed. Otherwise, validation fails and  $T_j$  is aborted.

- **Justification:** Either the first condition is satisfied, and there is no overlapped execution, or the second condition is satisfied and
  - the writes of  $T_j$  do not affect reads of  $T_i$  since they occur after  $T_i$  has finished its reads.
  - the writes of  $T_j$  do not affect reads of  $T_i$  since  $T_j$  does not read any item written by  $T_i$ .



# Schedule Produced by Validation

90

- Example of schedule produced using validation

$T_{14}$	$T_{15}$
<b>read</b> ( $B$ )	<b>read</b> ( $B$ )
	$B := B - 50$
	<b>read</b> ( $A$ )
	$A := A + 50$
<b>read</b> ( $A$ )	
( <i>validate</i> )	
<b>display</b> ( $A + B$ )	
	( <i>validate</i> )
	<b>write</b> ( $B$ )
	<b>write</b> ( $A$ )

# Multiversion Schemes

91

- Multiversion schemes keep old versions of data item to increase concurrency.
  - Multiversion Timestamp Ordering
  - Multiversion Two-Phase Locking
- Each successful **write** results in the creation of a new version of the data item written.
- Use timestamps to label versions.
- When a **read**(Q) operation is issued, select an appropriate version of Q based on the timestamp of the transaction, and return the value of the selected version.
- **reads** never have to wait as an appropriate version is returned immediately.

# Multiversion Timestamp Ordering

92

- Each data item  $Q$  has a sequence of versions  $\langle Q_1, Q_2, \dots, Q_m \rangle$ . Each version  $Q_k$  contains three data fields:
  - **Content** -- the value of version  $Q_k$ .
  - **W-timestamp**( $Q_k$ ) -- timestamp of the transaction that created (wrote) version  $Q_k$
  - **R-timestamp**( $Q_k$ ) -- largest timestamp of a transaction that successfully read version  $Q_k$
- when a transaction  $T_i$  creates a new version  $Q_k$  of  $Q$ ,  $Q_k$ 's W-timestamp and R-timestamp are initialized to  $TS(T_i)$ .
- R-timestamp of  $Q_k$  is updated whenever a transaction  $T_j$  reads  $Q_k$ , and  $TS(T_j) > \text{R-timestamp}(Q_k)$ .

# Multiversion Timestamp Ordering (Cont)

93

- Suppose that transaction  $T_i$  issues a **read**(Q) or **write**(Q) operation. Let  $Q_k$  denote the version of Q whose write timestamp is the largest write timestamp less than or equal to  $TS(T_i)$ .
  1. If transaction  $T_i$  issues a **read**(Q), then the value returned is the content of version  $Q_k$ .
  2. If transaction  $T_i$  issues a **write**(Q)
    1. if  $TS(T_i) < R\text{-timestamp}(Q_k)$ , then transaction  $T_i$  is rolled back.
    1. if  $TS(T_i) = W\text{-timestamp}(Q_k)$ , the contents of  $Q_k$  are overwritten
    2. else a new version of Q is created.
- Observe that
  - Reads always succeed
  - A write by  $T_i$  is rejected if some other transaction  $T_j$  that (in the serialization order defined by the timestamp values) should read  $T_i$ 's write, has already read a version created by a transaction older than  $T_i$ .
- Protocol guarantees serializability

# Multiversion Two-Phase Locking

94

- Differentiates between read-only transactions and update transactions
- *Update transactions* acquire read and write locks, and hold all locks up to the end of the transaction. That is, update transactions follow rigorous two-phase locking.
  - Each successful **write** results in the creation of a new version of the data item written.
  - each version of a data item has a single timestamp whose value is obtained from a counter **ts-counter** that is incremented during commit processing.
- *Read-only transactions* are assigned a timestamp by reading the current value of **ts-counter** before they start execution; they follow the multiversion timestamp-ordering protocol for performing reads.

## Multiversion Two-Phase Locking (Cont.)

95

- When an update transaction wants to read a data item:
  - it obtains a shared lock on it, and reads the latest version.
- When it wants to write an item
  - it obtains X lock on; it then creates a new version of the item and sets this version's timestamp to  $\infty$ .
- When update transaction  $T_i$  completes, commit processing occurs:
  - $T_i$  sets timestamp on the versions it has created to **ts-counter** + 1
  - $T_i$  increments **ts-counter** by 1
- Read-only transactions that start after  $T_i$  increments **ts-counter** will see the values updated by  $T_i$ .
- Read-only transactions that start before  $T_i$  increments the **ts-counter** will see the value before the updates by  $T_i$ .
- Only serializable schedules are produced.

# MVCC (Multiversion Concurrency Control): Implementation Issues

96

- Creation of multiple versions increases storage overhead
  - Extra tuples
  - Extra space in each tuple for storing version information
- Versions can, however, be garbage collected
  - E.g. if Q has two versions Q5 and Q9, and the oldest active transaction has timestamp  $> 9$ , then Q5 will never be required again

# Insert and Delete Operations

97

- If two-phase locking is used :
  - A **delete** operation may be performed only if the transaction deleting the tuple has an exclusive lock on the tuple to be deleted.
  - A transaction that inserts a new tuple into the database is given an X-mode lock on the tuple
- Insertions and deletions can lead to the **phantom phenomenon**.
  - A transaction that scans a relation
    - (e.g., find sum of balances of all accounts in Perryridge)and a transaction that inserts a tuple in the relation
    - (e.g., insert a new account at Perryridge)(conceptually) conflict in spite of not accessing any tuple in common.
  - If only tuple locks are used, non-serializable schedules can result
    - E.g. the scan transaction does not see the new account, but reads some other tuple written by the update transaction



# Insert and Delete Operations (Cont.)

98

- The transaction scanning the relation is reading information that indicates what tuples the relation contains, while a transaction inserting a tuple updates the same information.
  - The information should be locked.
- **One solution:**
  - Associate a data item with the relation, to represent the information about what tuples the relation contains.
  - Transactions scanning the relation acquire a shared lock in the data item,
  - Transactions inserting or deleting a tuple acquire an exclusive lock on the data item. (Note: locks on the data item do not conflict with locks on individual tuples.)
- Above protocol provides very low concurrency for insertions/deletions.
- Index locking protocols provide higher concurrency while preventing the phantom phenomenon, by requiring locks on certain index buckets.

# Index Locking Protocol

99

- Index locking protocol:
  - Every relation must have at least one index.
  - A transaction can access tuples only after finding them through one or more indices on the relation
  - A transaction  $T_i$  that performs a lookup must lock all the index leaf nodes that it accesses, in S-mode
    - Even if the leaf node does not contain any tuple satisfying the index lookup (e.g. for a range query, no tuple in a leaf is in the range)
  - A transaction  $T_i$  that inserts, updates or deletes a tuple  $t_i$  in a relation  $r$ 
    - must update all indices to  $r$
    - must obtain exclusive locks on all index leaf nodes affected by the insert/update/delete
  - The rules of the two-phase locking protocol must be observed
- Guarantees that phantom phenomenon won't occur

# Weak Levels of Consistency

100

- **Degree-two consistency:** differs from two-phase locking in that S-locks may be released at any time, and locks may be acquired at any time
  - X-locks must be held till end of transaction
  - Serializability is not guaranteed, programmer must ensure that no erroneous database state will occur]
- **Cursor stability:**
  - For reads, each tuple is locked, read, and lock is immediately released
  - X-locks are held till end of transaction
  - Special case of degree-two consistency

# Weak Levels of Consistency in SQL

101

- SQL allows non-serializable executions
  - **Serializable:** is the default
  - **Repeatable read:** allows only committed records to be read, and repeating a read should return the same value (so read locks should be retained)
    - However, the phantom phenomenon need not be prevented
      - T1 may see some records inserted by T2, but may not see others inserted by T2
  - **Read committed:** same as degree two consistency, but most systems implement it as cursor-stability
  - **Read uncommitted:** allows even uncommitted data to be read
- In many database systems, read committed is the default consistency level
  - has to be explicitly changed to serializable when required
    - **set isolation level serializable**

# Concurrency in Index Structures

102

- **Indices** are unlike other database items in that their only job is to help in accessing data.
- **Index-structures** are typically accessed very often, much more than other database items.
  - Treating **index-structures** like other database items, e.g. by **2-phase locking** of index nodes can lead to low concurrency.
- There are several **index concurrency protocols** where locks on internal nodes are released early, and not in a **two-phase fashion**.
  - It is acceptable to have **non serializable concurrent access** to an **index** as long as the accuracy of the **index** is maintained.
    - In particular, the exact values read in an internal node of a **B<sup>+</sup>-tree** are irrelevant so long as we land up in the correct leaf node.

# Concurrency in Index Structures (Cont.)

103

- **Example of index concurrency protocol:**
- Use **crabbing** instead of **two-phase locking** on the nodes of the **B<sup>+</sup>-tree**, as follows. During **search/insertion/deletion**:
  - **First lock** the root node in **shared mode**.
  - After locking all required children of a node in **shared mode**, release the lock on the node.
  - During insertion/deletion, upgrade leaf node locks to **exclusive mode**.
  - When splitting or coalescing requires changes to a parent, lock the parent in **exclusive mode**.
- Above protocol can cause **excessive deadlocks**
  - Searches coming down the tree deadlock with updates going up the tree
  - Can abort and restart search, without affecting transaction
- Better protocols are available; one such protocol, the **B-link tree protocol**
  - **Intuition:** release lock on parent before acquiring lock on child
    - And deal with changes that may have happened between lock release and acquire

# Next-Key Locking











104

- Index-locking protocol to prevent phantoms required locking entire leaf
  - Can result in poor concurrency if there are many inserts
- Alternative: for an index lookup
  - Lock all values that satisfy index lookup (match lookup value, or fall in lookup range)
  - Also lock next key value in index
  - Lock mode: S for lookups, X for insert/delete/update
- Ensures that range queries will conflict with inserts/deletes/updates
  - Regardless of which happens first, as long as both are concurrent

# UNIT – V (Part - 2)

105

## RECOVERY SYSTEM

-  Failure classification
-  Storage structure
-  Recovery and atomicity
-  Log-based recovery
-  Shadow paging
-  Recovery with concurrent transactions
-  Buffer management
-  Failure with loss of non-volatile storage
-  Advanced recovery techniques
-  Remote backup systems.



# Failure Classification

106

- **Transaction failure :**
  - **Logical errors:** transaction cannot complete due to some internal error condition
  - **System errors:** the database system must terminate an active transaction due to an error condition (e.g., deadlock)
- **System crash:** a power failure or other hardware or software failure causes the system to crash.
  - **Fail-stop assumption:** non-volatile storage contents are assumed to not be corrupted by system crash
    - Database systems have numerous integrity checks to prevent corruption of disk data
- **Disk failure:** a head crash or similar disk failure destroys all or part of disk storage
  - **Destruction is assumed to be detectable:** disk drives use checksums to detect failures

# Recovery Algorithms

107

- Recovery algorithms are techniques to ensure database consistency and transaction atomicity and durability despite failures
  - Focus of this chapter
- Recovery algorithms have two parts
  1. Actions taken during normal transaction processing to ensure enough information exists to recover from failures
  2. Actions taken after a failure to recover the database contents to a state that ensures atomicity, consistency and durability

# Storage Structure

108

- **Volatile storage:**

- does not survive system crashes
- examples: main memory, cache memory

- **Nonvolatile storage:**

- survives system crashes
- examples: disk, tape, flash memory, non-volatile (battery backed up) RAM

- **Stable storage:**

- a mythical form of storage that survives all failures
- approximated by maintaining multiple copies on distinct nonvolatile media

# Stable-Storage Implementation

109

- Maintain multiple copies of each block on separate disks
  - copies can be at remote sites to protect against disasters such as fire or flooding.
- Failure during data transfer can still result in inconsistent copies: Block transfer can result in
  - Successful completion
  - Partial failure: destination block has incorrect information
  - Total failure: destination block was never updated
- Protecting storage media from failure during data transfer (one solution):
  - Execute output operation as follows (assuming two copies of each block):
    1. Write the information onto the first physical block.
    2. When the first write successfully completes, write the same information onto the second physical block.
    3. The output is completed only after the second write successfully completes.

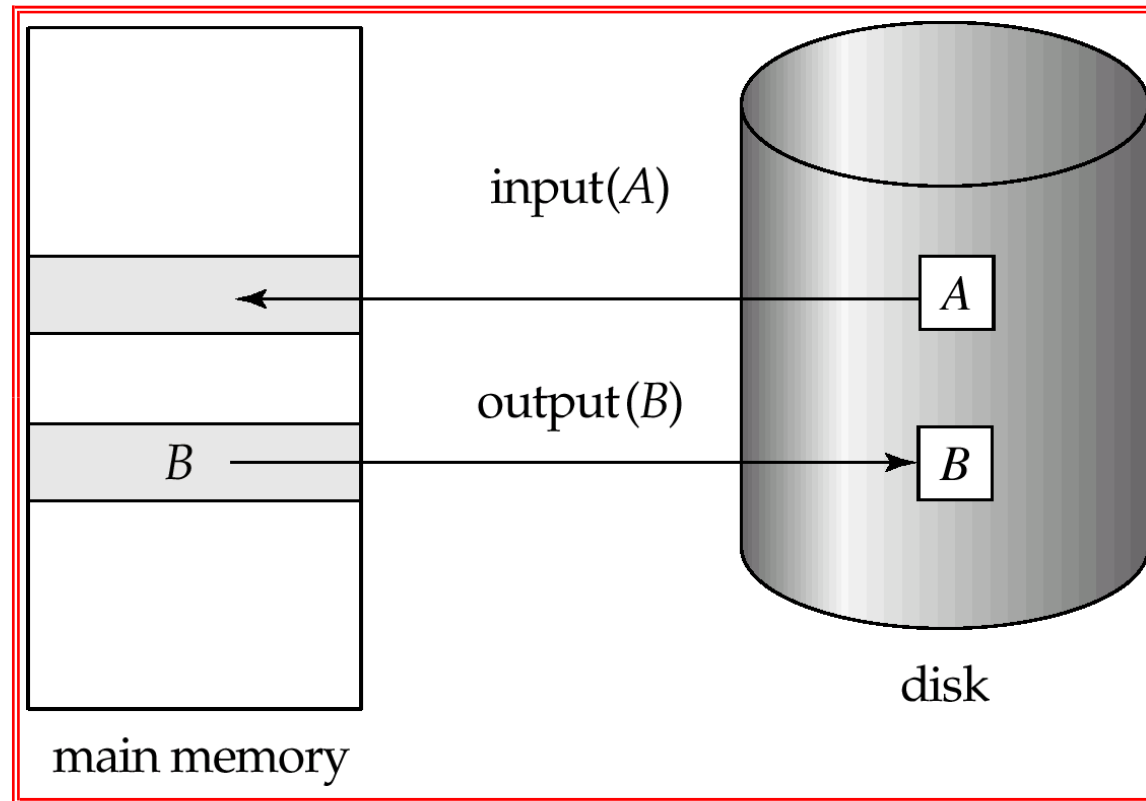
# Stable-Storage Implementation (Cont.)

110

- Protecting storage media from failure during data transfer (cont.):
- Copies of a block may differ due to failure during output operation. To recover from failure:
  1. First find inconsistent blocks:
    1. *Expensive solution*: Compare the two copies of every disk block.
    2. *Better solution*:
      - Record in-progress disk writes on non-volatile storage (Non-volatile RAM or special area of disk).
      - Use this information during recovery to find blocks that may be inconsistent, and only compare copies of these.
      - Used in hardware RAID systems
  2. If either copy of an inconsistent block is detected to have an error (bad checksum), overwrite it by the other copy. If both have no error, but are different, overwrite the second block by the first block.

# Block Storage Operations

111



# Data Access

112

- **Physical blocks** are those blocks residing on the disk.
- **Buffer blocks** are the blocks residing temporarily in main memory.
- Block movements between disk and main memory are initiated through the following two operations:
  - **input**( $B$ ) transfers the physical block  $B$  to main memory.
  - **output**( $B$ ) transfers the buffer block  $B$  to the disk, and replaces the appropriate physical block there.
- Each transaction  $T_i$  has its private work-area in which local copies of all data items accessed and updated by it are kept.
  - $T_i$ 's local copy of a data item  $X$  is called  $x_i$ .
- We assume, for simplicity, that each data item fits in, and is stored inside, a single block.

## Data Access (Cont.)

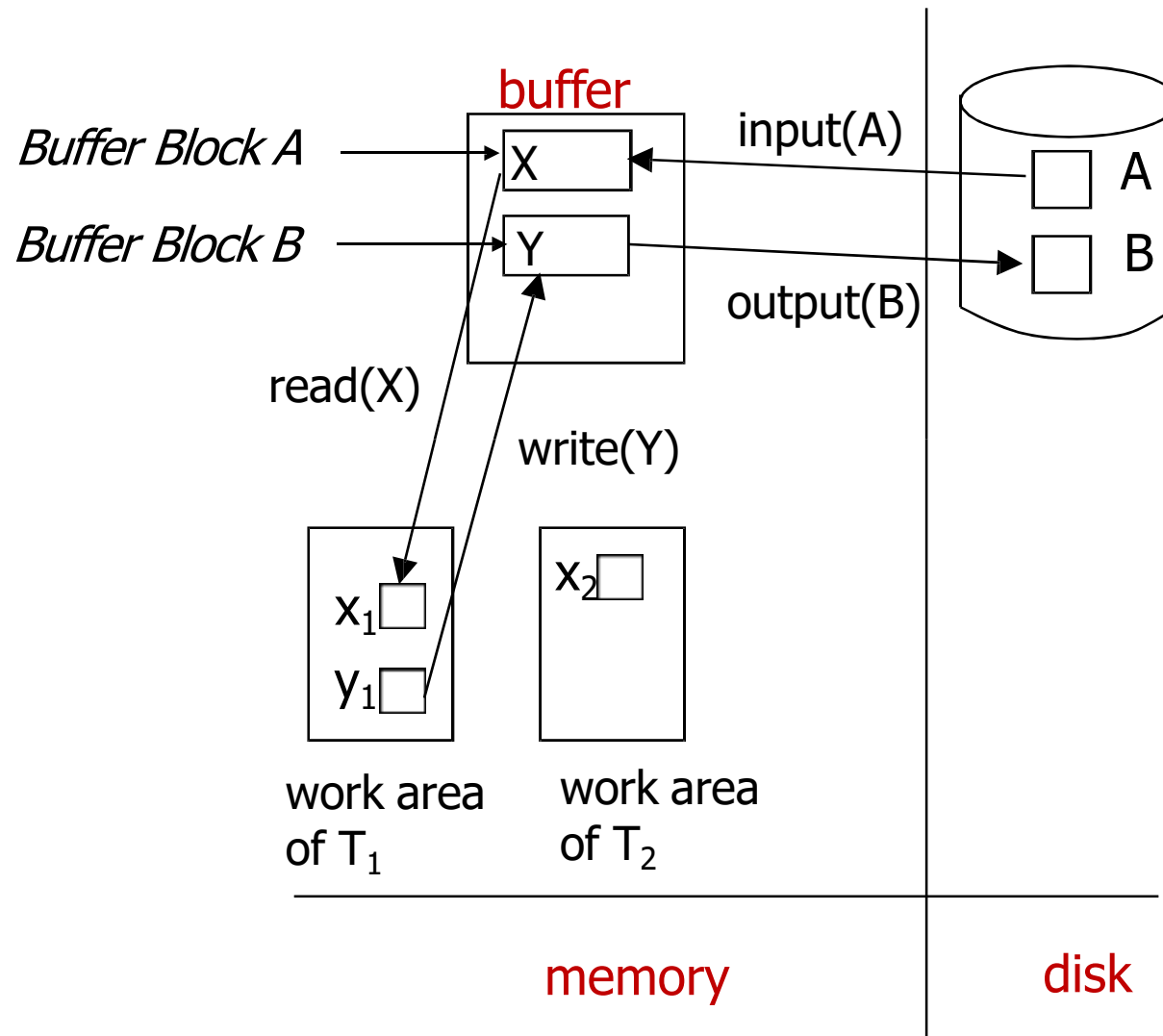
113

- Transaction transfers data items between system buffer blocks and its private work-area using the following operations :
  - **read**( $X$ ) assigns the value of data item  $X$  to the local variable  $x_i$ .
  - **write**( $X$ ) assigns the value of local variable  $x_i$  to data item  $\{X\}$  in the buffer block.
  - both these commands may necessitate the issue of an **input**( $B_X$ ) instruction before the assignment, if the block  $B_X$  in which  $X$  resides is not already in memory.
- Transactions
  - Perform **read**( $X$ ) while accessing  $X$  for the first time;
  - All subsequent accesses are to the local copy.
  - After last access, transaction executes **write**( $X$ ).
- **output**( $B_X$ ) need not immediately follow **write**( $X$ ). System can perform the **output** operation when it deems fit.



# Example of Data Access

114



# Recovery and Atomicity

115

- Modifying the database without ensuring that the transaction will commit may leave the database in an **inconsistent state**.
- Consider transaction  $T_i$  that transfers \$50 from account  $A$  to account  $B$ ; goal is either to perform all database modifications made by  $T_i$  or none at all.
- Several output operations may be required for  $T_i$  (to output  $A$  and  $B$ ). A **failure** may occur after one of these modifications have been made but before all of them are made.

## Recovery and Atomicity (Cont.)

116

- To ensure atomicity despite failures, we first output information describing the modifications to stable storage without modifying the database itself.
- We study two approaches:
  - **log-based recovery**, and
  - **shadow-paging**
- We assume (initially) that transactions run **serially**, that is, one after the other.

# Log-Based Recovery

117

- A **log** is kept on stable storage.
  - The log is a sequence of **log records**, and maintains a record of update activities on the database.
  - Each and every **log record** is given a unique id called log sequence number.
- A **log record** is written for each of the following actions:
  1. Updating a page
  2. Commit
  3. Abort
  4. End
  5. Undoing an update

Every **log record** contains the following fields:

Prev LSN	Trans Id	Type	Page ID	Length	Offset	Before Image	After image
-------------	-------------	------	------------	--------	--------	-----------------	----------------

## Log-Based Recovery Cont..

118

- When transaction  $T_i$  starts, it registers itself by writing a  $\langle T_i, \text{start} \rangle$  log record
- Before  $T_i$  executes **write**( $X$ ), a log record  $\langle T_i, X, V_1, V_2 \rangle$  is written, where  $V_1$  is the value of  $X$  before the write, and  $V_2$  is the value to be written to  $X$ .
  - Log record notes that  $T_i$  has performed a write on data item  $X_i$ .  $X_i$  had value  $V_1$  before the write, and will have value  $V_2$  after the write.
- When  $T_i$  finishes its last statement, the log record  $\langle T_i, \text{commit} \rangle$  is written.
- We assume for now that log records are written directly to stable storage (that is, they are not buffered)
- Two approaches using logs
  - Deferred database modification
  - Immediate database modification

# Deferred Database Modification

119

- The **deferred database modification** scheme records all modifications to the log, but defers all the **writes** to after partial commit.
- Assume that transactions execute serially
- Transaction starts by writing  $\langle T_i, \text{start} \rangle$  record to log.
- A **write**(X) operation results in a log record  $\langle T_i, X, V \rangle$  being written, where V is the new value for X
  - Note: old value is not needed for this scheme
- The write is not performed on X at this time, but is deferred.
- When  $T_i$  partially commits,  $\langle T_i, \text{commit} \rangle$  is written to the log
- Finally, the log records are read and used to actually execute the previously deferred writes.

# Deferred Database Modification (Cont.)

120

- During recovery after a crash, a transaction needs to be redone if and only if both  $\langle T_i \text{ start} \rangle$  and  $\langle T_i \text{ commit} \rangle$  are there in the log.
- Redoing a transaction  $T_i$  (**redo** $T_i$ ) sets the value of all data items updated by the transaction to the new values.
- Crashes can occur while
  - the transaction is executing the original updates, or
  - while recovery action is being taken
- example transactions  $T_0$  and  $T_1$  ( $T_0$  executes before  $T_1$ ):

$T_0$ : **read** (A)

A:- A - 50

**Write** (A)

**read** (B)

B:- B + 50

**write** (B)

$T_1$  : **read** (C)

C:- C- 100

**write** (C)

## Deferred Database Modification (Cont.)

121

- Below we show the log as it appears at three instances of time.

$\langle T_0 \text{ start} \rangle$	$\langle T_0 \text{ start} \rangle$	$\langle T_0 \text{ start} \rangle$
$\langle T_0, A, 950 \rangle$	$\langle T_0, A, 950 \rangle$	$\langle T_0, A, 950 \rangle$
$\langle T_0, B, 2050 \rangle$	$\langle T_0, B, 2050 \rangle$	$\langle T_0, B, 2050 \rangle$
	$\langle T_0 \text{ commit} \rangle$	$\langle T_0 \text{ commit} \rangle$
	$\langle T_1 \text{ start} \rangle$	$\langle T_1 \text{ start} \rangle$
	$\langle T_1, C, 600 \rangle$	$\langle T_1, C, 600 \rangle$
		$\langle T_1 \text{ commit} \rangle$
(a)	(b)	(c)

- If log on stable storage at time of crash is as in case:
  - (a) No redo actions need to be taken
  - (b) redo( $T_0$ ) must be performed since  $\langle T_0 \text{ commit} \rangle$  is present
  - (c) redo( $T_0$ ) must be performed followed by redo( $T_1$ ) since  $\langle T_0 \text{ commit} \rangle$  and  $\langle T_1 \text{ commit} \rangle$  are present



# Immediate Database Modification

122

- The **immediate database modification** scheme allows database updates of an uncommitted transaction to be made as the writes are issued
  - since undoing may be needed, update logs must have both old value and new value
- Update log record must be written *before* database item is written
  - We assume that the log record is output directly to stable storage
  - Can be extended to postpone log record output, so long as prior to execution of an **output**( $B$ ) operation for a data block  $B$ , all log records corresponding to items  $B$  must be flushed to stable storage
- Output of updated blocks can take place at any time before or after transaction commit
- Order in which blocks are output can be different from the order in which they are written.

# Immediate Database Modification Example

123

Log	Write	Output
$\langle T_0 \text{ start} \rangle$		
$\langle T_0, A, 1000, 950 \rangle$		
$T_0, B, 2000, 2050$		
	$A = 950$	
	$B = 2050$	
$\langle T_0 \text{ commit} \rangle$		
$\langle T_1 \text{ start} \rangle X_1$		
$\langle T_1, C, 700, 600 \rangle$		
	$C = 600$	
		$B_B, B_C$
$\langle T_1 \text{ commit} \rangle$		
		$B_A$
<p>□ Note: <math>B_X</math> denotes block containing <math>X</math>.</p>		

# Immediate Database Modification (Cont.)

124

- Recovery procedure has two operations instead of one:
  - **undo**( $T_i$ ) restores the value of all data items updated by  $T_i$  to their old values, going backwards from the last log record for  $T_i$
  - **redo**( $T_i$ ) sets the value of all data items updated by  $T_i$  to the new values, going forward from the first log record for  $T_i$
- Both operations must be **idempotent**
  - That is, even if the operation is executed multiple times the effect is the same as if it is executed once
    - Needed since operations may get re-executed during recovery
- When recovering after failure:
  - Transaction  $T_i$  needs to be undone if the log contains the record  $\langle T_i, \mathbf{start} \rangle$ , but does not contain the record  $\langle T_i, \mathbf{commit} \rangle$ .
  - Transaction  $T_i$  needs to be redone if the log contains both the record  $\langle T_i, \mathbf{start} \rangle$  and the record  $\langle T_i, \mathbf{commit} \rangle$ .
- Undo operations are performed first, then redo operations.

# Immediate DB Modification Recovery Example

125

Below we show the log as it appears at three instances of time.

$\langle T_0 \text{ start} \rangle$	$\langle T_0 \text{ start} \rangle$	$\langle T_0 \text{ start} \rangle$
$\langle T_0, A, 1000, 950 \rangle$	$\langle T_0, A, 1000, 950 \rangle$	$\langle T_0, A, 1000, 950 \rangle$
$\langle T_0, B, 2000, 2050 \rangle$	$\langle T_0, B, 2000, 2050 \rangle$	$\langle T_0, B, 2000, 2050 \rangle$
	$\langle T_0 \text{ commit} \rangle$	$\langle T_0 \text{ commit} \rangle$
	$\langle T_1 \text{ start} \rangle$	$\langle T_1 \text{ start} \rangle$
	$\langle T_1, C, 700, 600 \rangle$	$\langle T_1, C, 700, 600 \rangle$
		$\langle T_1 \text{ commit} \rangle$
(a)	(b)	(c)

Recovery actions in each case above are:

- (a) undo ( $T_0$ ): B is restored to 2000 and A to 1000.
- (b) undo ( $T_1$ ) and redo ( $T_0$ ): C is restored to 700, and then A and B are set to 950 and 2050 respectively.
- (c) redo ( $T_0$ ) and redo ( $T_1$ ): A and B are set to 950 and 2050 respectively. Then C is set to 600

# Checkpoints

126

- Problems in recovery procedure as discussed earlier :
  1. searching the entire log is time-consuming
  2. we might unnecessarily redo transactions which have already
  3. output their updates to the database.
- Streamline recovery procedure by periodically performing **checkpointing**
  1. Output all log records currently residing in main memory onto stable storage.
  2. Output all modified buffer blocks to the disk.
  3. Write a log record < **checkpoint** > onto stable storage.

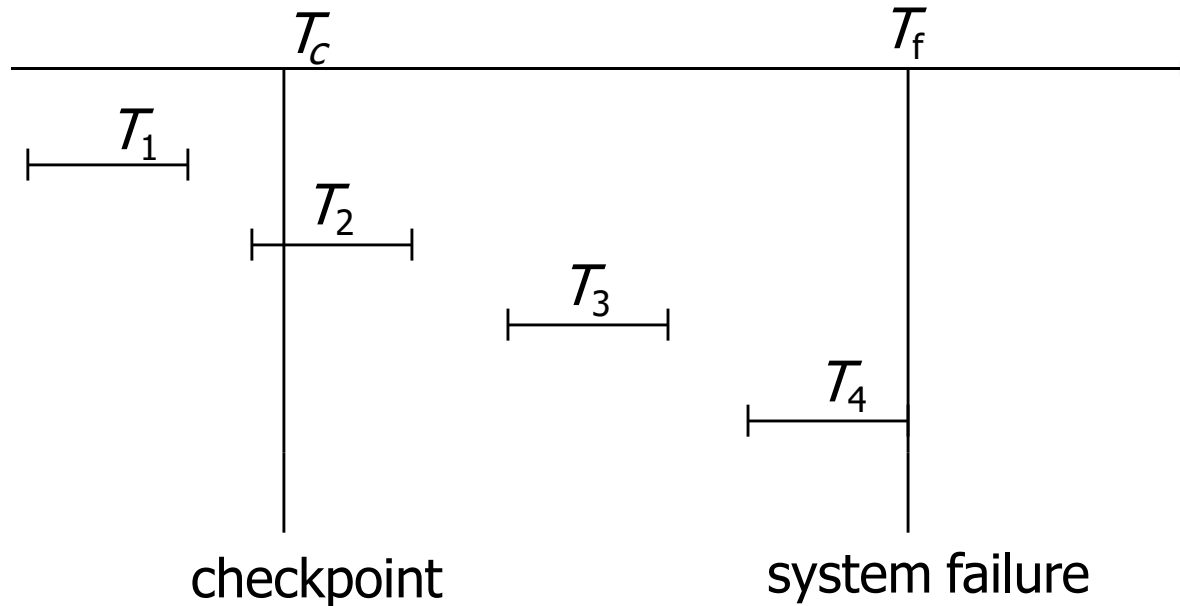
## Checkpoints (Cont.)

127

- During recovery we need to consider only the most recent transaction  $T_i$  that started before the checkpoint, and transactions that started after  $T_i$ .
  1. Scan backwards from end of log to find the most recent **<checkpoint>** record
  2. Continue scanning backwards till a record **< $T_i$  start>** is found.
  3. Need only consider the part of log following above **start** record. Earlier part of log can be ignored during recovery, and can be erased whenever desired.
  4. For all transactions (starting from  $T_i$  or later) with no **< $T_i$  commit>**, execute **undo( $T_i$ )**. (Done only in case of immediate modification.)
  5. Scanning forward in the log, for all transactions starting from  $T_i$  or later with a **< $T_i$  commit>**, execute **redo( $T_i$ )**.

# Example of Checkpoints

128



- $T_1$  can be ignored (updates already output to disk due to checkpoint)
- $T_2$  and  $T_3$  redone.
- $T_4$  undone

# Shadow Paging

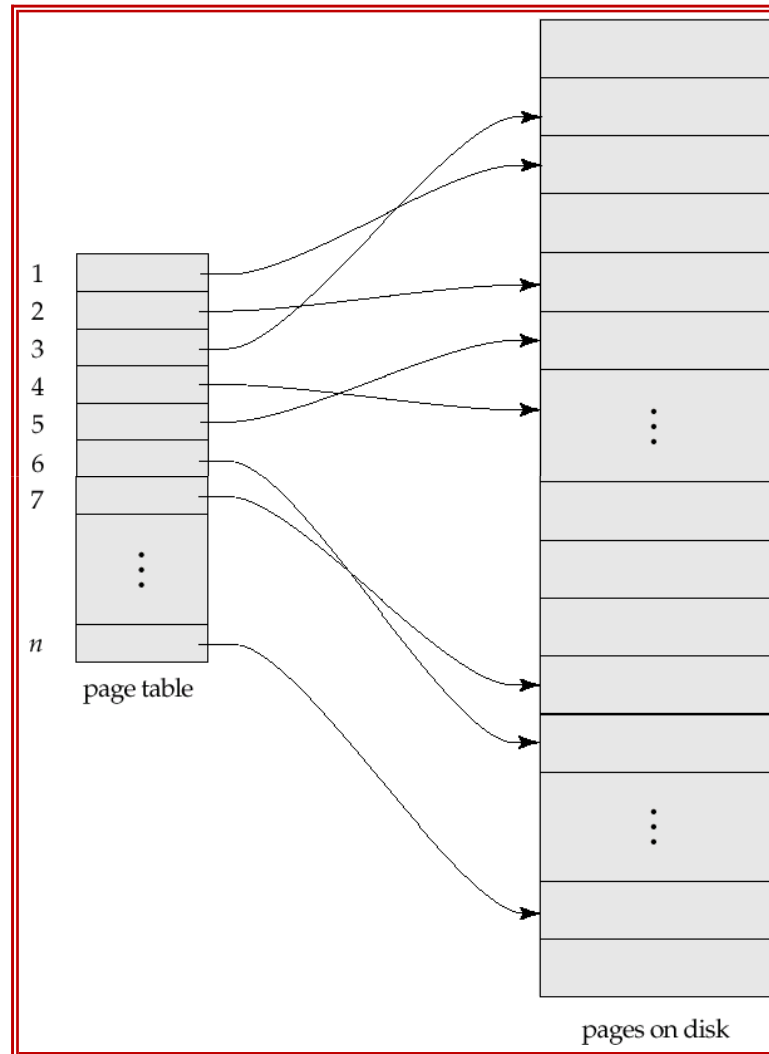
129

- **Shadow paging** is an alternative to log-based recovery; this scheme is useful if transactions execute serially
- **Idea:** maintain *two* page tables during the lifetime of a transaction - the **current page table**, and the **shadow page table**
- Store the shadow page table in nonvolatile storage, such that state of the database prior to transaction execution may be recovered.
  - Shadow page table is never modified during execution
- To start with, both the page tables are identical. Only current page table is used for data item accesses during execution of the transaction.
- Whenever any page is about to be written for the first time
  - A copy of this page is made onto an unused page.
  - The current page table is then made to point to the copy
  - The update is performed on the copy



# Sample Page Table

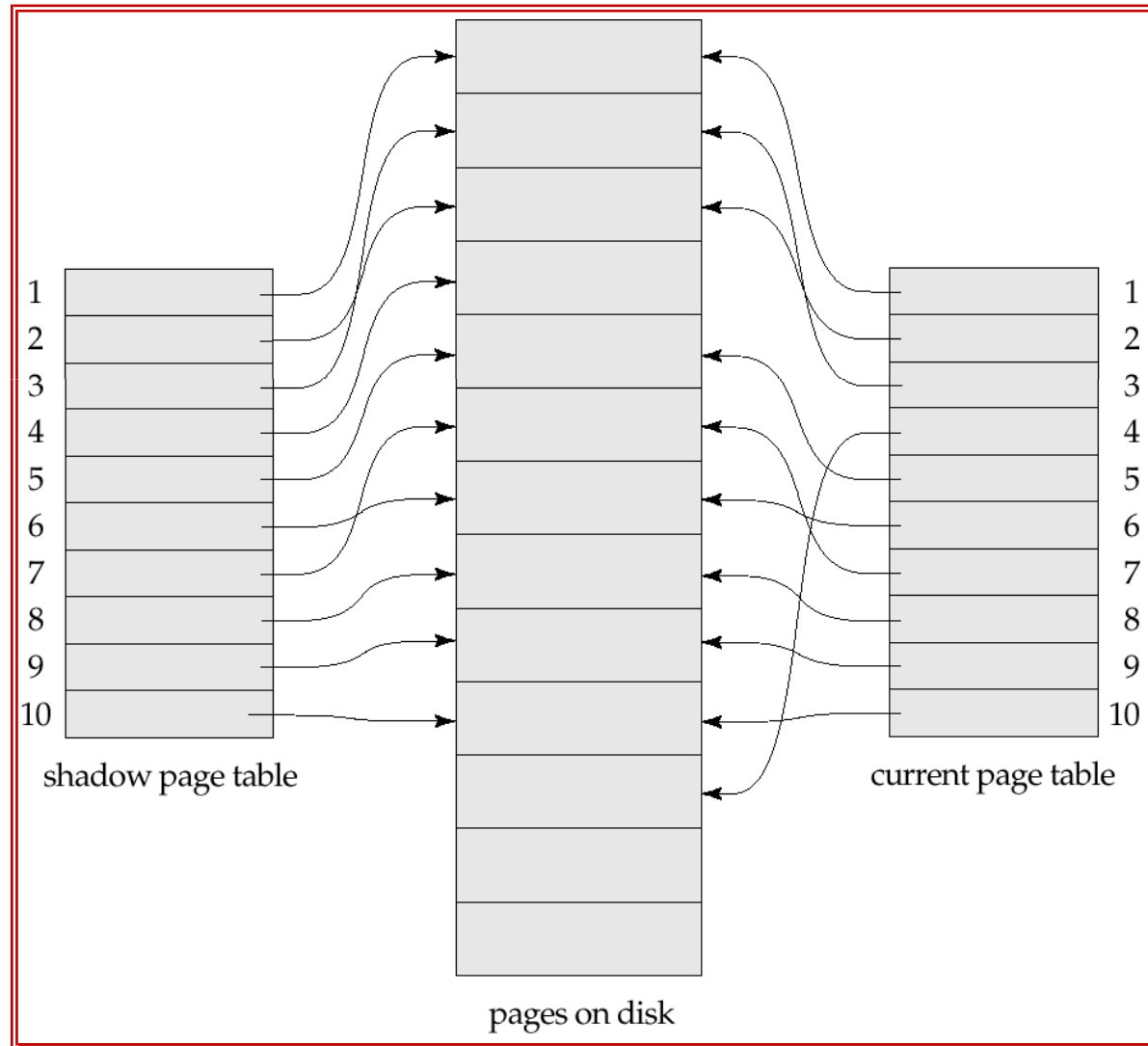
130



# Example of Shadow Paging

131

Shadow and current page tables after write to page 4



## Shadow Paging (Cont.)

132

- To commit a transaction :

1. Flush all modified pages in main memory to disk
2. Output current page table to disk
3. Make the current page table the new shadow page table, as follows:

- keep a pointer to the shadow page table at a fixed (known) location on disk.
  - to make the current page table the new shadow page table, simply update the pointer to point to current page table on disk
- Once pointer to shadow page table has been written, transaction is committed.
  - No recovery is needed after a crash - new transactions can start right away, using the shadow page table.
  - Pages not pointed to from current/shadow page table should be freed (garbage collected).

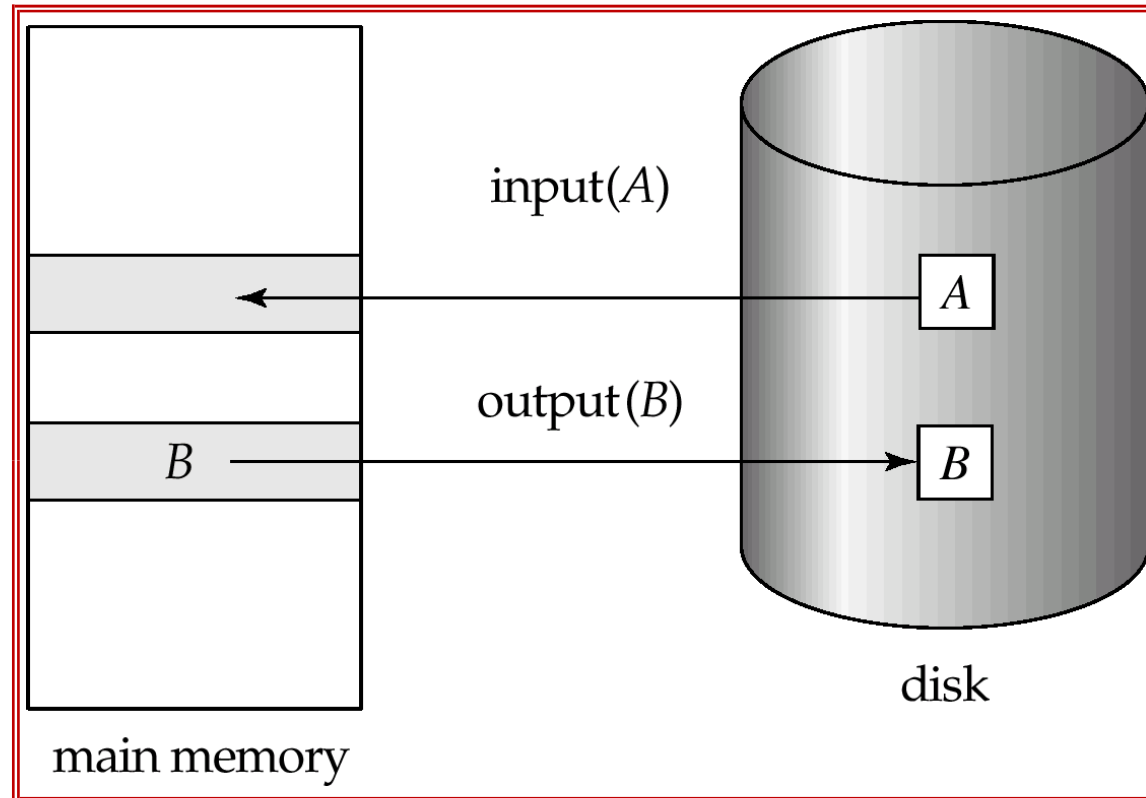
# Shadow Paging (Cont.)

133

- Advantages of shadow-paging over log-based schemes
  - no overhead of writing log records
  - recovery is trivial
- Disadvantages :
  - Copying the entire page table is very expensive
    - Can be reduced by using a page table structured like a B<sup>+</sup>-tree
      - No need to copy entire tree, only need to copy paths in the tree that lead to updated leaf nodes
  - Commit overhead is high even with above extension
    - Need to flush every updated page, and page table
  - Data gets fragmented (related pages get separated on disk)
  - After every transaction completion, the database pages containing old versions of modified data need to be garbage collected
  - Hard to extend algorithm to allow transactions to run concurrently
    - Easier to extend log based schemes

# Block Storage Operations

134



## Portion of the Database Log Corresponding to $T_0$ and $T_1$

135

$\langle T_0 \text{ start} \rangle$

$\langle T_0, A, 950 \rangle$

$\langle T_0, B, 2050 \rangle$

$\langle T_0 \text{ commit} \rangle$

$\langle T_1 \text{ start} \rangle$

$\langle T_1, C, 600 \rangle$

$\langle T_1 \text{ commit} \rangle$

## State of the Log and Database Corresponding to $T_0$ and $T_1$

136

Log	Database
$\langle T_0 \text{ start} \rangle$	
$\langle T_0, A, 950 \rangle$	
$\langle T_0, B, 2050 \rangle$	
$\langle T_0 \text{ commit} \rangle$	
	$A = 950$
	$B = 2050$
$\langle T_1 \text{ start} \rangle$	
$\langle T_1, C, 600 \rangle$	
$\langle T_1 \text{ commit} \rangle$	
	$C = 600$

## Portion of the System Log Corresponding to $T_0$ and $T_1$

137

$\langle T_0 \text{ start} \rangle$   
 $\langle T_0, A, 1000, 950 \rangle$   
 $\langle T_0, B, 2000, 2050 \rangle$   
 $\langle T_0 \text{ commit} \rangle$   
 $\langle T_1 \text{ start} \rangle$   
 $\langle T_1, C, 700, 600 \rangle$   
 $\langle T_1 \text{ commit} \rangle$



## State of System Log and Database Corresponding to $T_0$ and $T_1$

138

Log	Database
$\langle T_0 \text{ start} \rangle$	
$\langle T_0, A, 1000, 950 \rangle$	
$\langle T_0, B, 2000, 2050 \rangle$	
	$A = 950$
	$B = 2050$
$\langle T_0 \text{ commit} \rangle$	
$\langle T_1 \text{ start} \rangle$	
$\langle T_1, C, 700, 600 \rangle$	
	$C = 600$
$\langle T_1 \text{ commit} \rangle$	

# Recovery With Concurrent Transactions

139

- We modify the **log-based recovery** schemes to allow multiple transactions to execute concurrently.
  - All transactions share a single disk buffer and a single log
  - A buffer block can have data items updated by one or more transactions
- We assume **concurrency control** using **strict two-phase locking**;
  - i.e. the updates of uncommitted transactions should not be visible to other transactions
    - Otherwise how to perform undo if T1 updates A, then T2 updates A and commits, and finally T1 has to abort?
- **Logging** is done as described earlier.
  - Log records of different transactions may be interspersed in the log.
- The **checkpointing** technique and actions taken on recovery have to be changed
  - since several transactions may be active when a checkpoint is performed.

# Recovery With Concurrent Transactions (Cont.)

140

- **Checkpoints** are performed as before, except that the checkpoint log record is now of the form

**< checkpoint  $L$  >**

where  $L$  is the list of transactions active at the time of the checkpoint

- We assume no updates are in progress while the checkpoint is carried out (will relax this later)
- When the system recovers from a crash, it first does the following:
  1. Initialize *undo-list* and *redo-list* to empty
  2. Scan the log backwards from the end, stopping when the first **<checkpoint  $L$  >** record is found.  
For each record found during the backward scan:
    - if the record is **< $T_i$  commit>**, add  $T_i$  to *redo-list*
    - if the record is **< $T_i$  start>**, then if  $T_i$  is not in *redo-list*, add  $T_i$  to *undo-list*
  3. For every  $T_i$  in  $L$ , if  $T_i$  is not in *redo-list*, add  $T_i$  to *undo-list*

# Recovery With Concurrent Transactions (Cont.)

141

- At this point *undo-list* consists of incomplete transactions which must be undone, and *redo-list* consists of finished transactions that must be redone.
- Recovery now continues as follows:
  1. Scan log backwards from most recent record, stopping when  $\langle T_i \text{ start} \rangle$  records have been encountered for every  $T_i$  in *undo-list*.
    - During the scan, perform **undo** for each log record that belongs to a transaction in *undo-list*.
  2. Locate the most recent  $\langle \text{checkpoint } L \rangle$  record.
  3. Scan log forwards from the  $\langle \text{checkpoint } L \rangle$  record till the end of the log.
    - During the scan, perform **redo** for each log record that belongs to a transaction on *redo-list*

## Example of Recovery

142

- Go over the steps of the recovery algorithm on the following log:

$\langle T_0 \text{ start} \rangle$

$\langle T_0, A, 0, 10 \rangle$

$\langle T_0 \text{ commit} \rangle$

$\langle T_1 \text{ start} \rangle$  /\* Scan at step 1 comes up to here \*/

$\langle T_1, B, 0, 10 \rangle$

$\langle T_2 \text{ start} \rangle$

$\langle T_2, C, 0, 10 \rangle$

$\langle T_2, C, 10, 20 \rangle$

$\langle \text{checkpoint } \{T_1, T_2\} \rangle$

$\langle T_3 \text{ start} \rangle$

$\langle T_3, A, 10, 20 \rangle$

$\langle T_3, D, 0, 10 \rangle$

$\langle T_3 \text{ commit} \rangle$

# Log Record Buffering

143

- **Log record buffering:** log records are buffered in main memory, instead of being output directly to stable storage.
  - Log records are output to stable storage when a block of log records in the buffer is full, or a **log force** operation is executed.
- **Log force** is performed to commit a transaction by forcing all its log records (including the commit record) to stable storage.
- Several log records can thus be output using a single output operation, reducing the I/O cost.

## Log Record Buffering (Cont.)

144

- The rules below must be followed if log records are buffered:
  - Log records are output to stable storage in the order in which they are created.
  - Transaction  $T_i$  enters the commit state only when the log record  **$\langle T_i, \text{commit} \rangle$**  has been output to stable storage.
  - Before a block of data in main memory is output to the database, all log records pertaining to data in that block must have been output to stable storage.
    - This rule is called the **write-ahead logging or WAL** rule
      - Strictly speaking WAL only requires undo information to be output

# Database Buffering

145

- Database maintains an **in-memory buffer** of data blocks
  - When a new block is needed, if buffer is full an existing block needs to be removed from buffer
  - If the block chosen for removal has been updated, it must be output to disk
- If a block with uncommitted updates is output to disk, log records with undo information for the updates are output to the log on stable storage first
  - (**Write ahead logging**)
- No updates should be in progress on a block when it is output to disk. Can be ensured as follows.
  - Before writing a data item, transaction acquires exclusive lock on block containing the data item
  - Lock can be released once the write is completed.
    - Such locks held for short duration are called **latches**.
  - Before a block is output to disk, the system acquires an exclusive latch on the block
    - Ensures no update can be in progress on the block



## Buffer Management (Cont.)

146

- Database buffer can be implemented either
  - in an area of real **main-memory** reserved for the database, or
  - in **virtual memory**
- Implementing buffer in reserved **main-memory** has drawbacks:
  - Memory is partitioned before-hand between database buffer and applications, limiting flexibility.
  - Needs may change, and although operating system knows best how memory should be divided up at any time, it cannot change the partitioning of memory.

# Buffer Management (Cont.)

147

- **Database buffers** are generally implemented in **virtual memory** in spite of some **drawbacks**:
  - When operating system needs to evict a page that has been modified, the page is written to swap space on disk.
  - When database decides to write buffer page to disk, buffer page may be in swap space, and may have to be read from swap space on disk and output to the database on disk, resulting in extra I/O!
    - Known as **dual paging** problem.
  - Ideally when OS needs to evict a page from the buffer, it should pass control to database, which in turn should
    1. Output the page to database instead of to swap space (making sure to output log records first), if it is modified
    2. Release the page from the buffer, for the OS to useDual paging can thus be avoided, but common operating systems do not support such functionality.

# Failure with Loss of Nonvolatile Storage

148

- So far we assumed no loss of **non-volatile** storage
- Technique similar to checkpointing used to deal with loss of non-volatile storage
  - Periodically **dump** the entire content of the database to stable storage
  - No transaction may be active during the dump procedure; a procedure similar to checkpointing must take place
    - Output all log records currently residing in main memory onto stable storage.
    - Output all buffer blocks onto the disk.
    - Copy the contents of the database to stable storage.
    - Output a record **<dump>** to log on stable storage.

# Recovering from Failure of Non-Volatile Storage

149

- To recover from disk failure
  - restore database from most recent dump.
  - Consult the log and redo all transactions that committed after the dump
- Can be extended to allow transactions to be active during dump; known as **fuzzy dump or online dump**
  - Will study **fuzzy checkpointing** later

# Advanced Recovery: Key Features

150

- Support for high-concurrency locking techniques, such as those used for B<sup>+</sup>-tree concurrency control, which release locks early
  - Supports “logical undo”
- Recovery based on “repeating history”, whereby recovery executes exactly the same actions as normal processing
  - including redo of log records of incomplete transactions, followed by subsequent undo
  - Key benefits
    - supports logical undo
    - easier to understand/show correctness

# Advanced Recovery: Logical Undo Logging

151

- Operations like **B<sup>+</sup>-tree** insertions and deletions release locks early.
  - They cannot be undone by restoring old values (**physical undo**), since once a lock is released, other transactions may have updated the **B<sup>+</sup>-tree**.
  - Instead, insertions (resp. deletions) are undone by executing a deletion (resp. insertion) operation (known as **logical undo**).
- For such operations, undo log records should contain the undo operation to be executed
  - Such logging is called **logical undo logging**, in contrast to **physical undo logging**
    - Operations are called **logical operations**
  - Other examples:
    - delete of tuple, to undo insert of tuple
      - allows early lock release on space allocation information
    - subtract amount deposited, to undo deposit
      - allows early lock release on bank balance

# Advanced Recovery: Physical Redo

152

- Redo information is logged **physically** (that is, new value for each write) even for operations with logical undo
  - **Logical redo** is very complicated since database state on disk may not be “operation consistent” when recovery starts
  - **Physical redo** logging does not conflict with early lock release

# Advanced Recovery: Operation Logging

153

□ **Operation logging** is done as follows:

1. When operation starts, log  $\langle T_i, O_i, \text{operation-begin} \rangle$ . Here  $O_i$  is a unique identifier of the operation instance.
2. While operation is executing, normal log records with physical redo and physical undo information are logged.
3. When operation completes,  $\langle T_i, O_i, \text{operation-end}, U \rangle$  is logged, where  $U$  contains information needed to perform a logical undo information.

**Example:** insert of (key, record-id) pair (K5, RID7) into index I9

```
<T1, O1, operation-begin>
....
<T1, X, 10, K5>
<T1, Y, 45, RID7>
<T1, O1, operation-end, (delete I9, K5, RID7)>
```

} Physical redo of steps in insert



# Advanced Recovery: Operation Logging (Cont.)

154

- If **crash/rollback** occurs before operation completes:
  - the **operation-end** log record is not found, and
  - the physical undo information is used to undo operation.
- If crash/rollback occurs after the operation completes:
  - the **operation-end** log record is found, and in this case
  - logical undo is performed using  $U$ ; the physical undo information for the operation is ignored.
- Redo of operation (after crash) still uses physical redo information.

# Advanced Recovery: Txn Rollback

155

Rollback of transaction  $T_i$  is done as follows:

□ **Scan the log backwards**

1. If a log record  $\langle T_i, X, V_1, V_2 \rangle$  is found, perform the undo and log a special **redo-only log record**  $\langle T_i, X, V_1 \rangle$ .
2. If a  $\langle T_i, O_j, \text{operation-end}, U \rangle$  record is found
  - Rollback the operation logically using the undo information  $U$ .
    - Updates performed during roll back are logged just like during normal operation execution.
    - At the end of the operation rollback, instead of logging an **operation-end** record, generate a record  $\langle T_i, O_j, \text{operation-abort} \rangle$ .
  - Skip all preceding log records for  $T_i$  until the record  $\langle T_i, O_j, \text{operation-begin} \rangle$  is found

# Advanced Recovery: Txn Rollback (Cont.)

156

- Scan the log backwards (cont.):
  3. If a redo-only record is found ignore it
  4. If a  $\langle T_i, O_i, \text{operation-abort} \rangle$  record is found:
    - skip all preceding log records for  $T_i$  until the record  $\langle T_i, O_i, \text{operation-begin} \rangle$  is found.
  5. Stop the scan when the record  $\langle T_i, \text{start} \rangle$  is found
- 6. Add a  $\langle T_i, \text{abort} \rangle$  record to the log

Some points to note:

- Cases 3 and 4 above can occur only if the database crashes while a transaction is being rolled back.
- Skipping of log records as in case 4 is important to prevent multiple rollback of the same operation.

# Advanced Recovery: Txn Rollback Example

157

- Example with a complete and an incomplete operation

<T1, start>

<T1, O1, operation-begin>

....

<T1, X, 10, K5>

<T1, Y, 45, RID7>

<T1, O1, operation-end, (delete I9, K5, RID7)>

<T1, O2, operation-begin>

<T1, Z, 45, 70>

① T1 Rollback begins here

<T1, Z, 45>      ① redo-only log record during physical undo (of incomplete O2)

<T1, Y, ., .>      ① Normal redo records for logical undo of O1

...

<T1, O1, operation-abort>      ① What if crash occurred immediately after this?

<T1, abort>

# Advanced Recovery: Crash Recovery

158

The following actions are taken when recovering from system crash

1. (**Redo phase**): Scan log forward from last **< checkpoint L >** record till end of log
  1. **Repeat history** by physically redoing all updates of all transactions,
  2. Create an **undo-list** during the scan as follows
    - **undo-list** is set to *L* initially
    - Whenever **<T<sub>i</sub> start>** is found *T<sub>i</sub>* is added to **undo-list**
    - Whenever **<T<sub>i</sub> commit>** or **<T<sub>i</sub> abort>** is found, *T<sub>i</sub>* is deleted from **undo-list**

This brings database to state as of crash, with committed as well as uncommitted transactions having been redone.

Now **undo-list** contains transactions that are **incomplete**, that is, have neither committed nor been fully rolled back.

# Advanced Recovery: Crash Recovery (Cont.)

159

## Recovery from system crash (cont.)

2. (**Undo phase**): Scan log backwards, performing undo on log records of transactions found in *undo-list*.
  - Log records of transactions being rolled back are processed as described earlier, as they are found
    - Single shared scan for all transactions being undone
  - When  $\langle T_i \text{ start} \rangle$  is found for a transaction  $T_i$  in *undo-list*, write a  $\langle T_i \text{ abort} \rangle$  log record.
  - Stop scan when  $\langle T_i \text{ start} \rangle$  records have been found for all  $T_i$  in *undo-list*
- This undoes the effects of incomplete transactions (those with neither **commit** nor **abort** log records). Recovery is now complete.

# Advanced Recovery: Checkpointing

160

- **Checkpointing** is done as follows:
  1. Output all log records in memory to stable storage
  2. Output to disk all modified buffer blocks
  3. Output to log on stable storage a **< checkpoint L >** record.

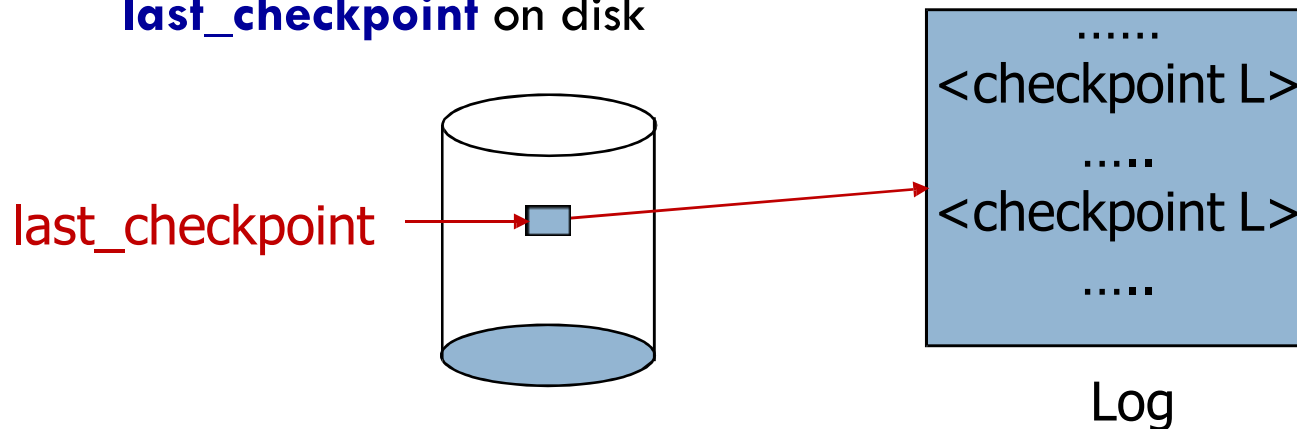
Transactions are not allowed to perform any actions while checkpointing is in progress.

- **Fuzzy checkpointing** allows transactions to progress while the most time consuming parts of checkpointing are in progress

# Advanced Recovery: Fuzzy Checkpointing

161

- **Fuzzy checkpointing** is done as follows:
  1. Temporarily stop all updates by transactions
  2. Write a **<checkpoint L>** log record and force log to stable storage
  3. Note list *M* of modified buffer blocks
  4. Now permit transactions to proceed with their actions
  5. Output to disk all modified buffer blocks in list *M*
    - blocks should not be updated while being output
    - **Follow WAL:** all log records pertaining to a block must be output before the block is output
  6. Store a pointer to the **checkpoint** record in a fixed position **last\_checkpoint** on disk





# Advanced Rec: Fuzzy Checkpointing (Cont.)

162

- When recovering using a **fuzzy checkpoint**, start scan from the **checkpoint** record pointed to by **last\_checkpoint**
  - Log records before **last\_checkpoint** have their updates reflected in database on disk, and need not be redone.
  - Incomplete checkpoints, where system had crashed while performing a checkpoint, are handled safely

# ARIES

## (Algorithms for Recovery and Isolation Exploiting Semantics)

163

- **ARIES** is a state of the art recovery method
  - Incorporates numerous optimizations to reduce overheads during normal processing and to speed up recovery
  - The “**advanced recovery algorithm**” we studied earlier is modeled after **ARIES**, but greatly simplified by removing optimizations
- Unlike the advanced recovery algorithm, **ARIES**
  1. Uses **log sequence number (LSN)** to identify log records
    - Stores LSNs in pages to identify what updates have already been applied to a database page
  2. Physiological redo
  3. Dirty page table to avoid unnecessary **redos** during recovery
  4. Fuzzy checkpointing that only records information about dirty pages, and does not require dirty pages to be written out at checkpoint time

# ARIES Optimizations

164

## □ Physiological redo

- Affected page is physically identified, action within page can be logical

### ■ Used to reduce logging overheads

- e.g. when a record is deleted and all other records have to be moved to fill hole

- Physiological redo can log just the record deletion
- Physical redo would require logging of old and new values for much of the page

### ■ Requires page to be output to disk atomically

- Easy to achieve with hardware **RAID**, also supported by some disk systems
- Incomplete page output can be detected by checksum techniques,
  - But extra actions are required for recovery
  - Treated as a media failure

# ARIES Data Structures

165

- ARIES uses several data structures
  - Log sequence number (LSN) identifies each log record
    - Must be sequentially increasing
    - Typically an offset from beginning of log file to allow fast access
      - Easily extended to handle multiple log files
  - Page LSN
  - Log records of several different types
  - Dirty page table

# ARIES Data Structures: Page LSN

166

- Each page contains a **PageLSN** which is the LSN of the last log record whose effects are reflected on the page
  - **To update a page:**
    - X-latch the page, and write the log record
    - Update the page
    - Record the LSN of the log record in PageLSN
    - Unlock page
  - **To flush page to disk, must first S-latch page**
    - Thus page state on disk is operation consistent
      - Required to support physiological redo
  - **PageLSN is used during recovery to prevent repeated redo**
    - Thus ensuring idempotence

# ARIES Data Structures: Log Record

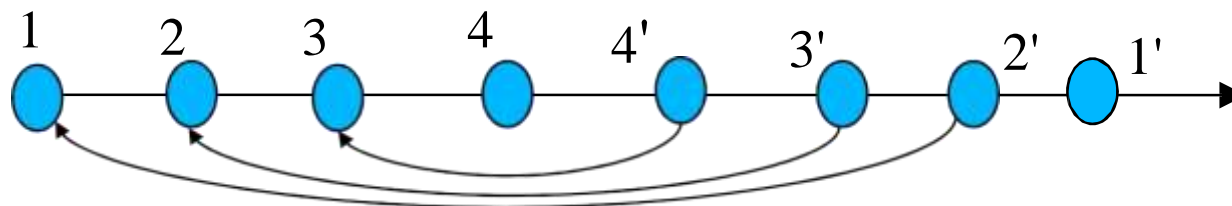
167

- Each log record contains LSN of previous log record of the same transaction

LSN	TransID	PrevLSN	RedoInfo	UndoInfo
-----	---------	---------	----------	----------

- LSN in log record may be implicit
- Special redo-only log record called **compensation log record (CLR)** used to log actions taken during recovery that never need to be undone
  - Serves the role of operation-abort log records used in advanced recovery algorithm
  - Has a field UndoNextLSN to note next (earlier) record to be undone
    - Records in between would have already been undone
    - Required to avoid repeated undo of already undone actions

LSN	TransID	UndoNextLSN	RedoInfo
-----	---------	-------------	----------

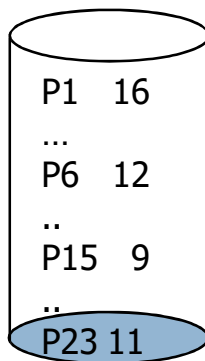


# ARIES Data Structures: DirtyPage Table

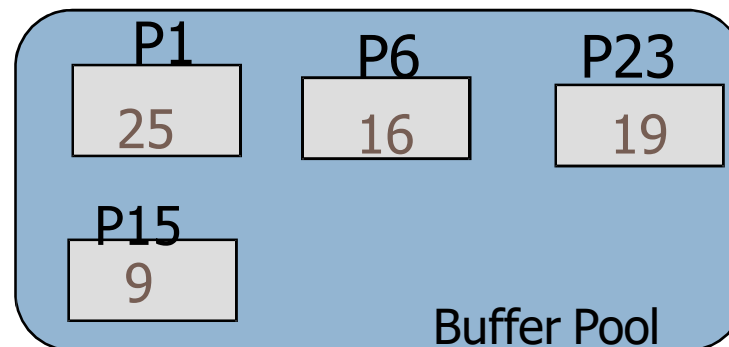
168

## □ DirtyPageTable

- List of pages in the buffer that have been updated
- Contains, for each such page
  - **PageLSN** of the page
  - **RecLSN** is an LSN such that log records before this LSN have already been applied to the page version on disk
    - Set to current end of log when a page is inserted into dirty page table (just before being updated)
    - Recorded in checkpoints, helps to minimize redo work



Page LSNs on disk



Page	PLSN	RLSN
P1	25	17
P6	16	15
P23	19	18

Dirty Page Table

# ARIES Data Structures: Checkpoint Log

169

## □ Checkpoint log record

### □ Contains:

- DirtyPageTable and list of active transactions

- For each active transaction, LastLSN, the LSN of the last log record written by the transaction

- Fixed position on disk notes LSN of last completed checkpoint log record

- Dirty pages are not written out at checkpoint time

- Instead, they are flushed out continuously, in the background

- Checkpoint is thus very low overhead

- can be done frequently



# ARIES Recovery Algorithm

170

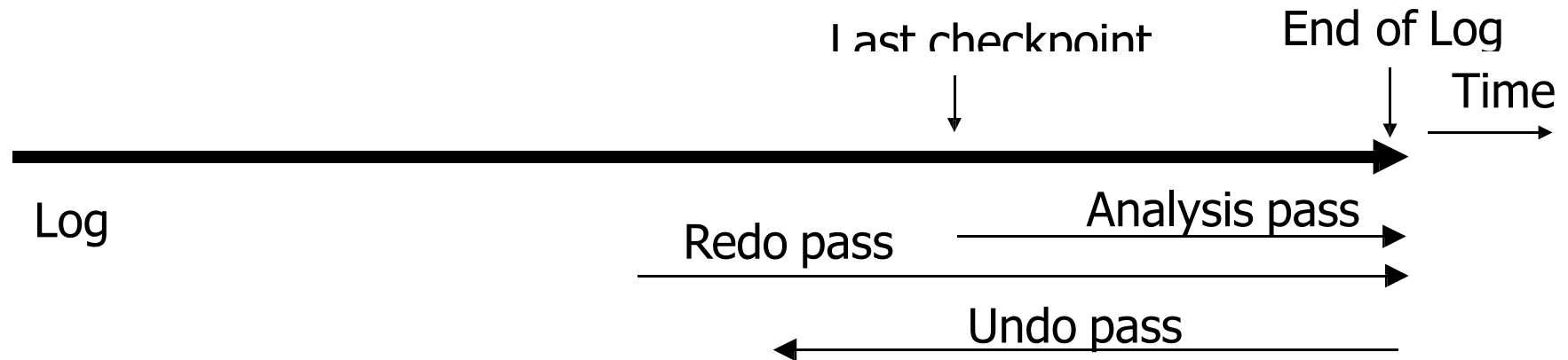
ARIES recovery involves three passes

- **Analysis pass:** Determines
  - Which transactions to undo
  - Which pages were dirty (disk version not up to date) at time of crash
  - **RedoLSN:** LSN from which redo should start
- **Redo pass:**
  - Repeats history, redoing all actions from RedoLSN
    - RecLSN and PageLSNs are used to avoid redoing actions already reflected on page
- **Undo pass:**
  - Rolls back all incomplete transactions
    - Transactions whose abort was complete earlier are not undone
      - **Key idea:** no need to undo these transactions: earlier undo actions were logged, and are redone as required

# Aries Recovery: 3 Passes

171

- Analysis, redo and undo passes
- Analysis determines where redo should start
- Undo has to go back till start of earliest incomplete transaction



# ARIES Recovery: Analysis

172

## Analysis pass

- Starts from last complete checkpoint log record
  - Reads DirtyPageTable from log record
  - Sets RedoLSN = min of RecLSNs of all pages in DirtyPageTable
    - In case no pages are dirty, RedoLSN = checkpoint record's LSN
  - Sets undo-list = list of transactions in checkpoint log record
  - Reads LSN of last log record for each transaction in undo-list from checkpoint log record
- Scans forward from checkpoint
- .. Cont. on next page ...

# ARIES Recovery: Analysis (Cont.)

173

## Analysis pass (cont.)

### □ Scans forward from checkpoint

- If any log record found for transaction not in undo-list, adds transaction to undo-list
- Whenever an update log record is found
  - If page is not in DirtyPageTable, it is added with RecLSN set to LSN of the update log record
- If transaction end log record found, delete transaction from undo-list
- Keeps track of last log record for each transaction in undo-list
  - May be needed for later undo

### □ At end of analysis pass:

- RedoLSN determines where to start redo pass
- RecLSN for each page in DirtyPageTable used to minimize redo work
- All transactions in undo-list need to be rolled back

# ARIES Redo Pass

174

**Redo Pass:** Repeats history by replaying every action not already reflected in the page on disk, as follows:

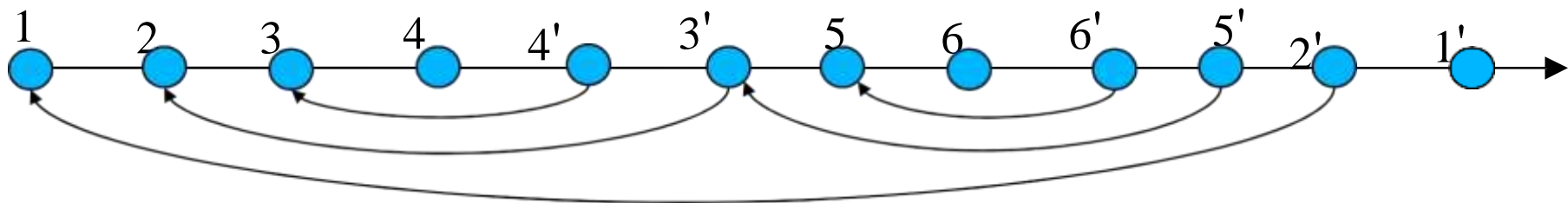
- Scans forward from RedoLSN. Whenever an update log record is found:
  1. If the page is not in DirtyPageTable or the LSN of the log record is less than the RecLSN of the page in DirtyPageTable, then skip the log record
  2. Otherwise fetch the page from disk. If the PageLSN of the page fetched from disk is less than the LSN of the log record, redo the log record

**NOTE:** if either test is negative the effects of the log record have already appeared on the page. First test avoids even fetching the page from disk!

# ARIES Undo Actions

175

- When an undo is performed for an update log record
  - Generate a CLR containing the undo action performed (actions performed during undo are logged physically or physiologically).
    - CLR for record  $n$  noted as  $n'$  in figure below
  - Set UndoNextLSN of the CLR to the PrevLSN value of the update log record
    - Arrows indicate UndoNextLSN value
- ARIES supports partial rollback
  - Used e.g. to handle deadlocks by rolling back just enough to release reqd. locks
  - Figure indicates forward actions after partial rollbacks
    - records 3 and 4 initially, later 5 and 6, then full rollback



# ARIES: Undo Pass

176

## Undo pass:

- Performs backward scan on log undoing all transaction in undo-list
  - Backward scan optimized by skipping unneeded log records as follows:
    - Next LSN to be undone for each transaction set to LSN of last log record for transaction found by analysis pass.
    - At each step pick largest of these LSNs to undo, skip back to it and undo it
    - After undoing a log record
      - For ordinary log records, set next LSN to be undone for transaction to PrevLSN noted in the log record
      - For compensation log records (CLRs) set next LSN to be undo to UndoNextLSN noted in the log record
        - All intervening records are skipped since they would have been undone already
- Undos performed as described earlier

# Other ARIES Features

177

## □ Recovery Independence

- Pages can be recovered independently of others
  - E.g. if some disk pages fail they can be recovered from a backup while other pages are being used

## □ Savepoints:

- Transactions can record savepoints and roll back to a savepoint
  - Useful for complex transactions
  - Also used to rollback just enough to release locks on deadlock



## Other ARIES Features (Cont.)

178

### □ Fine-grained locking:

- Index concurrency algorithms that permit tuple level locking on indices can be used
  - These require logical undo, rather than physical undo, as in advanced recovery algorithm

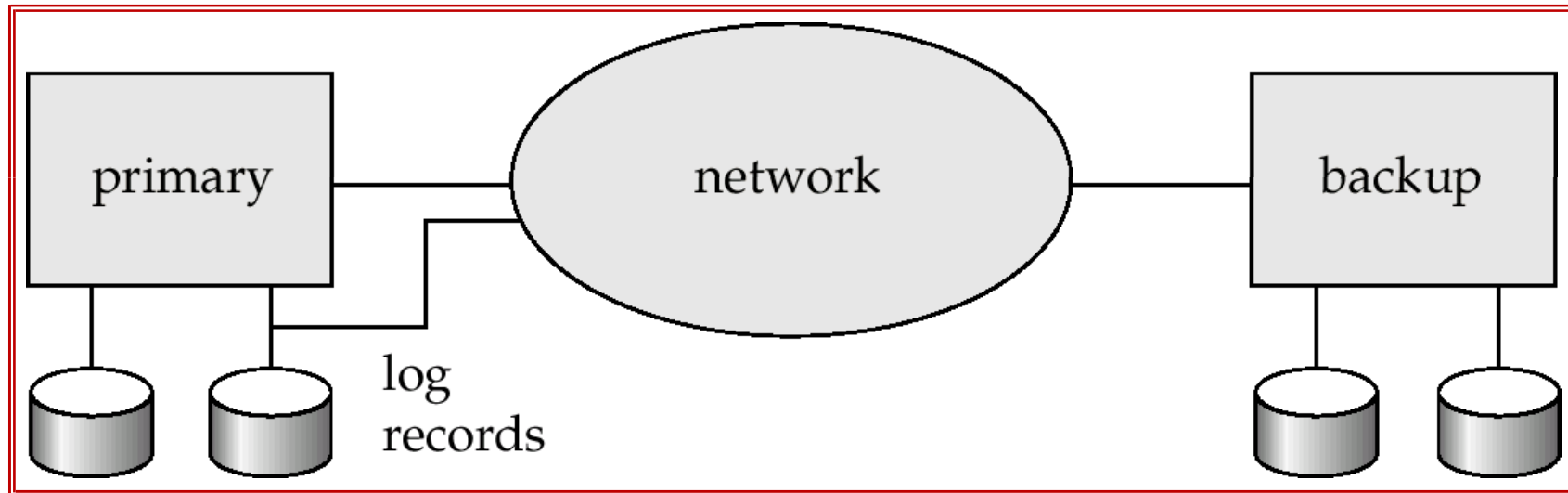
### □ Recovery optimizations: For example:

- Dirty page table can be used to **prefetch** pages during redo
- Out of order redo is possible:
  - redo can be postponed on a page being fetched from disk, and performed when page is fetched.
  - Meanwhile other log records can continue to be processed

# Remote Backup Systems

179

- Remote backup systems provide high availability by allowing transaction processing to continue even if the primary site is destroyed.



# Remote Backup Systems (Cont.)

180

- **Detection of failure:** Backup site must detect when primary site has failed
  - to distinguish primary site failure from link failure maintain several communication links between the primary and the remote backup.
  - Heart-beat messages
- **Transfer of control:**
  - To take over control backup site first perform recovery using its copy of the database and all the log records it has received from the primary.
    - Thus, completed transactions are redone and incomplete transactions are rolled back.
  - When the backup site takes over processing it becomes the new primary
  - To transfer control back to old primary when it recovers, old primary must receive redo logs from the old backup and apply all updates locally.

## Remote Backup Systems (Cont.)

181

- **Time to recover:** To reduce delay in takeover, backup site periodically processes the redo log records (in effect, performing recovery from previous database state), performs a checkpoint, and can then delete earlier parts of the log.
- **Hot-Spare** configuration permits very fast takeover:
  - Backup continually processes redo log record as they arrive, applying the updates locally.
  - When failure of the primary is detected the backup rolls back incomplete transactions, and is ready to process new transactions.
- Alternative to remote backup: distributed database with replicated data
  - Remote backup is faster and cheaper, but less tolerant to failure

## Remote Backup Systems (Cont.)

182

- Ensure durability of updates by delaying transaction commit until update is logged at backup; avoid this delay by permitting lower degrees of durability.
- **One-safe:** commit as soon as transaction's commit log record is written at primary
  - Problem: updates may not arrive at backup before it takes over.
- **Two-very-safe:** commit when transaction's commit log record is written at primary and backup
  - Reduces availability since transactions cannot commit if either site fails.
- **Two-safe:** proceed as in two-very-safe if both primary and backup are active. If only the primary is active, the transaction commits as soon as its commit log record is written at the primary.
  - Better availability than two-very-safe; avoids problem of lost transactions in one-safe.