



LECTURE 7

DATABASE RECOVERY

(PART 2/2)



NIKON D90 F3.5 3s ISO320

C5000Z F2.8 1/800s ISO50



OUTLINES



Redo Logging



Undo/Redo Logging



Protecting Against Media Failures



OUTLINES



- 17.3 **Redo Logging**
- 17.4 **Undo/Redo Logging**
- 17.5 **Protecting Against Media Failures**



Why redo log?

- Problem for undo logging
 - First write all its changed data to disk then commit a transaction → **too many disk I/O's**
- Save disk I/O's
 - Let changed data reside only in main memory for a while
 - Use *redo logging to fix things up in the event of a crash*



Differences between redo and undo logging

Undo logging	Redo logging
<i>Cancels</i> the effect of incomplete transactions; <i>Ignores</i> committed ones during recovery	<i>Ignores</i> incomplete transactions ; <i>Repeats</i> the changes made by committed transactions
Write changed database elements to disk before the COMMIT log record reaches disk	Requires the COMMIT record appear on disk before any changed values reach disk
The old values of changed database elements	The new values of changed database elements



Redo logging rules

- **R1:** Before modifying any database element X on disk, it is necessary that all log records pertaining to this modification of X , including both the update record $\langle T, X, v \rangle$ and the $\langle \text{COMMIT } T \rangle$ record, must appear on disk.
 1. The log records indicating changed database elements.
 2. The COMMIT log record.
 3. The changed database elements themselves

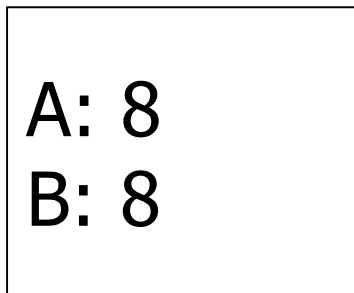
Write-ahead logging rule



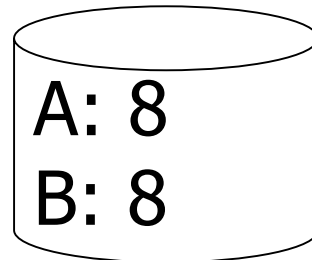
Redo logging (deferred modification)



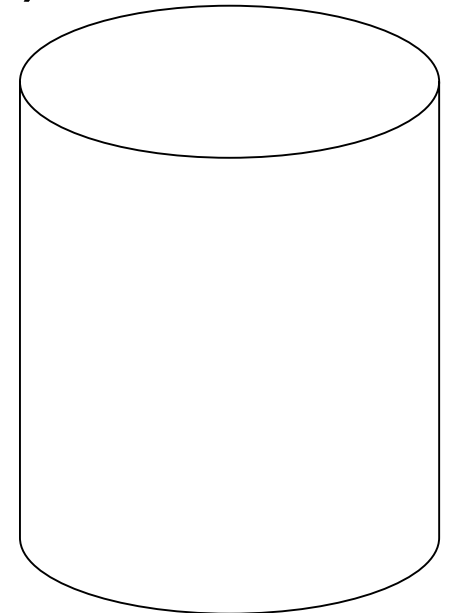
T: Read(A,t); $t \leftarrow t \times 2$; write (A,t);
 Read(B,t); $t \leftarrow t \times 2$; write (B,t);
 Output(A); Output(B);



memory



DB



LOG

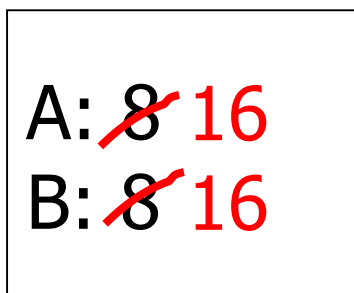


Redo logging (deferred modification)

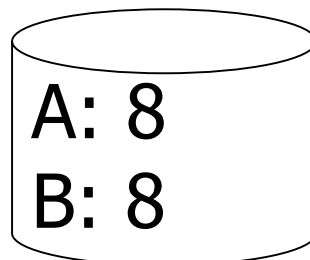
T: Read(A,t); $t \leftarrow t \times 2$; write (A,t);

Read(B,t); $t \leftarrow t \times 2$; write (B,t);

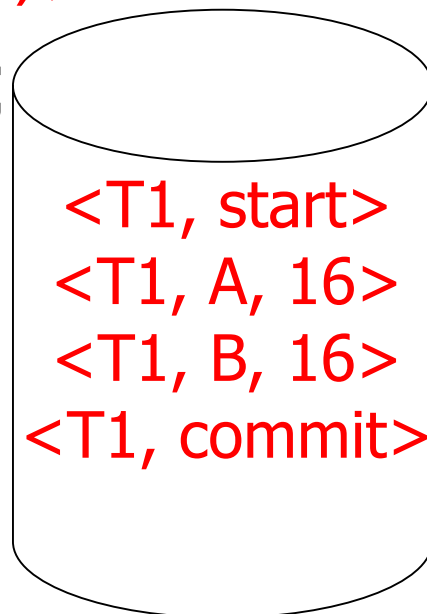
Flush-log; Output(A); Output(B);



memory



DB

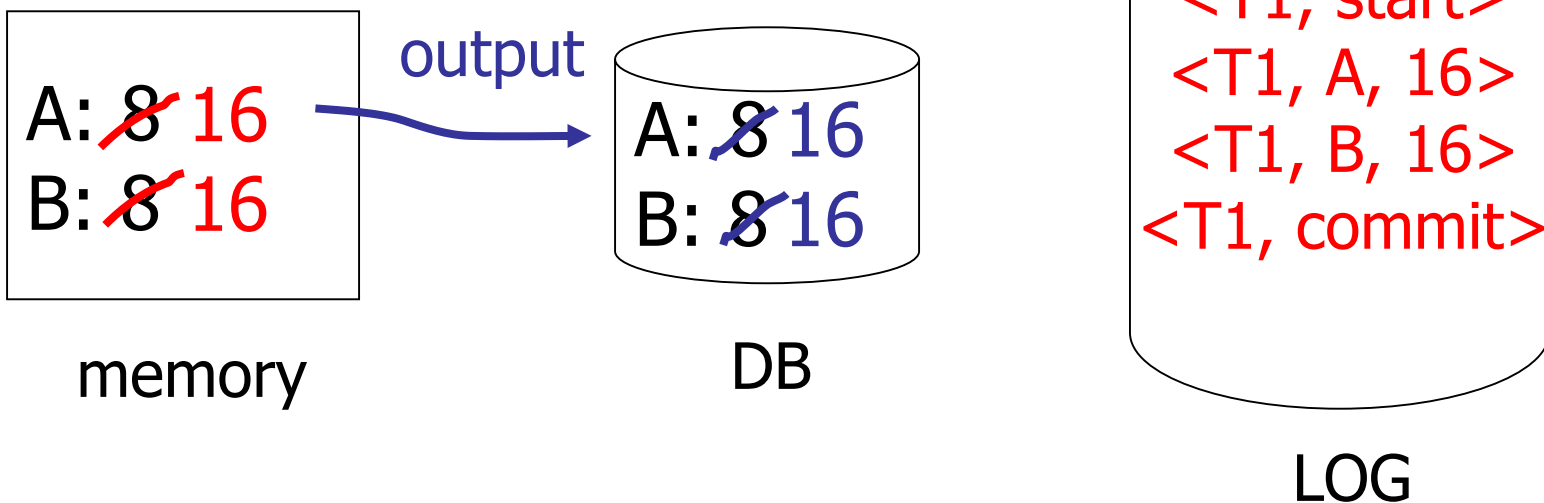


LOG



Redo logging (deferred modification)

T: Read(A,t); $t \leftarrow t \times 2$; write (A,t);
 Read(B,t); $t \leftarrow t \times 2$; write (B,t);
Flush-log; Output(A); Output(B);





Redo log

T: Read(A,t); $t := t \times 2$; write (A,t);
 Read(B,t); $t := t \times 2$; write (B,t); Output(A); Output(B)

Step	Action	t	M-A	M-B	D-A	D-B	Log
1)							<START T>
2)	READ(A,t)	8	8		8	8	
3)	$t := t * 2$	16	8		8	8	
4)	WRITE(A,t)	16	16		8	8	<T, A, 16>
5)	READ(B,t)	8	16	8	8	8	
6)	$t := t * 2$	16	16	8	8	8	
7)	WRITE(B,t)	16	16	16	8	8	<T, B, 16>
8)							<COMMIT T>
9)	FLUSH LOG						
10)	OUTPUT(A)	16	16	16	16	8	
11)	OUTPUT(B)	16	16	16	16	16	

Figure 17.7: Actions and their log entries using redo logging



Recovery with Redo logging

Consequence of R1: unless the log has a $\langle \text{COMMIT } T \rangle$ record, we know that no changes to the database made by transaction T have been written to disk.

- **Incomplete transactions** may be treated during recovery as if they had never occurred.
- **The completed transactions**, write the new values in redo log to disk regardless of whether they were already there.



Recovery with Redo logging



- To recover, using a redo log, after a system crash:
 1. Identify the committed transactions.
 2. Scan the log forward from the beginning. For each log record $\langle T, X, v \rangle$ encountered:
 - a) If T is not a committed transaction, do nothing.
 - b) If T is committed, write value v for database element X .
 3. For each incomplete transaction T , write an $\langle \text{ABORT } T \rangle$ record to the log and flush the log.



Recovery rules: Redo logging

- (1) Let S = set of transactions with $\langle \text{COMMIT}, T_i \rangle$ in log
- (2) For each $\langle T_i, X, v \rangle$ in log, in forward order (earliest \rightarrow latest) do:
 - if $T_i \in S$ then $\begin{cases} \text{WRITE}(X, v) \\ \text{OUTPUT}(X) \end{cases}$
- (3) For each $T_i \notin S$, write $\langle \text{ABORT } T_i \rangle$



Example 17.7

Step	Action	t	M-A	M-B	D-A	D-B	Log
1)							<START T >
2)	READ(A,t)	8	8		8	8	
3)	$t := t*2$	16	8		8	8	
4)	WRITE(A,t)	16	16		8	8	< $T, A, 16$ >
5)	READ(B,t)	8	16	8	8	8	
6)	$t := t*2$	16	16	8	8	8	
7)	WRITE(B,t)	16	16	16	8	8	< $T, B, 16$ >
8)							<COMMIT T >
9)	FLUSH LOG						
10)	OUTPUT(A)	16	16	16	16	8	
11)	OUTPUT(B)	16	16	16	16	16	

1、The crash occurs any time after step (9)

2、The crash occurs between steps (8) and (9)

3、The crash occurs prior to step (8)



Checkpointing a Redo Log

- Redo logs present a checkpointing problem
 - The database changes made by a committed transaction can be copied to disk much later than the time at which the transaction commits
 - Cannot limit our concern to transactions that are active at the time we decide to create a checkpoint



Checkpointing a Redo Log

- The steps to perform a nonquiescent checkpoint of redo log:
 1. Write a log record **<START CKPT (T_1, \dots, T_k)>**, where T_1, \dots, T_k are all the active (uncommitted) transactions, and flush the log
 2. Write to disk all database elements that were written to buffers but not yet to disk by transactions that had already committed when the **START CKPT** record was written to the log
 3. Write an **<END CKPT>** record to the log and flush the log



Checkpointing a Redo Log

- Example

<START T_1 >
< T_1 , A , 5>
<START T_2 >
<COMMIT T_1 >
< T_2 , B , 10>
<START CKPT (T_2)>
< T_2 , C , 15>
<START T_3 >
< T_3 , D , 20>
<END CKPT>
<COMMIT T_2 >
<COMMIT T_3 >

- When we start the checkpoint, only T_2 is active, but the value of A written by T_1 may have reached disk.
- If not, then we must copy A to disk before the checkpoint can end



Recovery With a Checkpointed Redo Log

- Two cases:
 1. The last checkpoint record on the log before a crash is **<END CKPT>**
 - a) Every value written by a transaction that committed before the corresponding **<START CKPT (T_1, \dots, T_k)>** has had its changes written to disk
 - b) Limit our attention to the transactions that are either one of the T_i 's mentioned in the **<START CKPT (T_1, \dots, T_k)>** or that started after the **<START CKPT (T_1, \dots, T_k)>**
 - c) We do not have to look further back than the earliest of the **<START T_i >** records. Notice that these **START** records could appear prior to any number of checkpoints



Recovery With a Checkpointed Redo Log

- Two cases:
 2. The last checkpoint record on the log before a crash is $\langle \text{START CKPT } (T_1, \dots, T_k) \rangle$
 - a) We cannot be sure that committed transactions prior to the start of this checkpoint had their changes written to disk
 - b) We must search back to the **previous** $\langle \text{END CKPT} \rangle$ record, find its matching $\langle \text{START CKPT } (S_1, \dots, S_m) \rangle$ record
 - c) Redo all those committed transactions that either started after $\langle \text{START CKPT } (S_1, \dots, S_m) \rangle$ or are among the S_i 's



Recovery With a Checkpointed Redo Log

<START T_1 >
< T_1 , A, 5>
<START T_2 >
<COMMIT T_1 >
< T_2 , B, 10>
<START CKPT (T_2)>
< T_2 , C, 15>
<START T_3 >
< T_3 , D, 20>
<END CKPT>
<COMMIT T_2 >
<COMMIT T_3 >





A crash occurs

- Example
 - Case 1
 - Redo all those transactions that either started after **<START CKPT (T_2)>** or that are on its list
 - Search the log as far back as the **<START T_2 >**



Recovery With a Checkpointed Redo Log

<START T_1 >
< T_1 , A, 5>
<START T_2 >
<COMMIT T_1 >
< T_2 , B, 10>
<START CKPT (T_2)>
< T_2 , C, 15>
 <START T_3 >
 < T_3 , D, 20>

<END CKPT>
<COMMIT T_2 >
<COMMIT T_3 >

A crash occurs

- Example
 - Case 2
 - There is no previous checkpoints, and we must go all the way to the beginning of the log
 - We identify T_1 as the only committed transaction



Recovery With a Checkpointed Redo Log

- Transactions may be active during several checkpoints
 - To include in $\langle \text{START CKPT } (T_1, \dots, T_k) \rangle$ pointers to the place on the log where T_1, \dots, T_k started
 - By doing so, we know when it is safe to delete early portions of the log
 - When we write an $\langle \text{END CKPT} \rangle$, we know that we shall never need to look back further than the earliest of the $\langle \text{START } T_i \rangle$ for the active transactions T_i
 - Thus, anything prior to that START record may be deleted



OUTLINES



17.3 Redo Logging



17.4 Undo/Redo Logging



17.5 Protecting Against Media Failures



Key drawbacks:



- *Undo logging*: requires that data be written to disk immediately after a transaction finishes, increasing the number of disk I/O's
- *Redo logging*: requires to keep all modified blocks in buffers until the transaction commits and the log records have been flushed, increasing the average number of buffers required by transactions



Undo/Redo Logging



- Provides increased flexibility, at the expense of maintain more information on the log
- The update log record: $\langle T_i, X, v, w \rangle$
 - Transaction T changed the value of database element X
 - its former value was v
 - its new value is w



Undo/redo Rules



- UR_1 Before modifying any database element X on disk because of changes made by some transaction T , it is necessary that the update record $\langle T, X, v, w \rangle$ appear on disk
 - Rule UR_1 for undo/redo logging enforces only the constraints enforced by **both** undo logging and redo logging
 - The $\langle \text{COMMIT } T \rangle$ log record can precede or follow any of the changes to the database elements on disk



Undo/redo log

Step	Action	t	M-A	M-B	D-A	D-B	Log
1)							<START T >
2)	READ(A, t)	8	8		8	8	
3)	$t := t * 2$	16	8		8	8	
4)	WRITE(A, t)	16	16		8	8	< $T, A, 8, 16$ >
5)	READ(B, t)	8	16	8	8	8	
6)	$t := t * 2$	16	16	8	8	8	
7)	WRITE(B, t)	16	16	16	8	8	< $T, B, 8, 16$ >
8)	FLUSH LOG						
9)	OUTPUT(A)	16	16	16	16	8	
10)							<COMMIT T >
11)	OUTPUT(B)	16	16	16	16	16	

Notice: Step (10) could also have appeared before step (8) or step (9), or after step (11).



Undo/redo Rules



- Problem with delayed commitment
 - A transaction appears to the user to have been completed, and yet because the $\langle \text{COMMIT } T \rangle$ record was not flushed to disk
 - A subsequent crash causes the transaction to be undone rather than redone
- UR_2 (Additional rule)
 - A $\langle \text{COMMIT } T \rangle$ record must be flushed to disk as soon as it appears in the log.

Add **FLUSH LOG** after step (10) in the previous example



Recovery With Undo/Redo Logging

- The undo/redo recovery policy is:
 1. Redo all the committed transactions in the order earliest-first.
 2. Undo all the incomplete transactions in the order latest-first.



Recovery With Undo/Redo Logging

- Necessary to do both undo and redo

Reason: The relative order in which COMMIT log records and the database changes are copied to disk

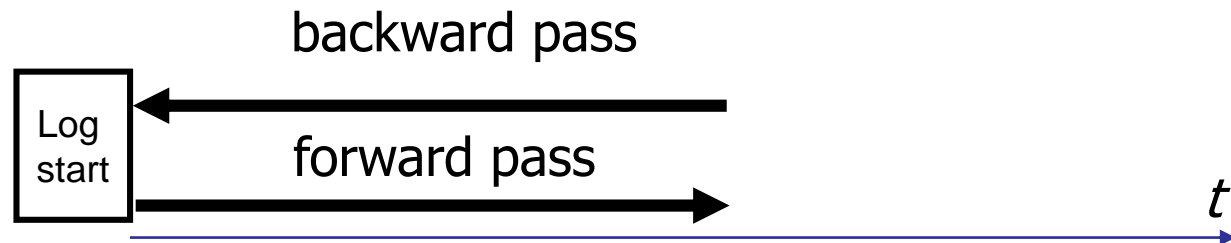
- Either a committed transaction with some or all of its changes not on disk
- Or an uncommitted transaction with some or all of its changes on disk



Recovery process:



- Backwards pass (end of log \rightarrow start; latest-first)
 - construct set S of committed transactions
 - undo actions of transactions not in S
- Forward pass (start \rightarrow end of log; earliest-first)
 - redo actions of transactions in S





Recovery process: Example



Step	Action	t	M-A	M-B	D-A	D-B	Log
1)							<START T >
2)	READ(A, t)	8	8		8	8	
3)	$t := t * 2$	16	8		8	8	
4)	WRITE(A, t)	16	16		8	8	< $T, A, 8, 16$ >
5)	READ(B, t)	8	16	8	8	8	
6)	$t := t * 2$	16	16	8	8	8	
7)	WRITE(B, t)	16	16	16	8	8	< $T, B, 8, 16$ >
8)	FLUSH LOG						
9)	OUTPUT(A)	16	16	16	16	8	
10)							<COMMIT T >
11)	OUTPUT(B)	16	16	16	16	16	

1. The crash occurs after the <COMMIT T > record is flushed to disk
2. The crash occurs prior to the <COMMIT T > record is flushed to disk



Checkpointing an Undo/Redo Log

- A nonquiescent checkpoint is somewhat simpler for undo/redo logging than for other logging methods
 1. Write a **<START CKPT (T_1, \dots, T_k)>** record to the log, where T_1, \dots, T_k are all the active transactions, and flush the log
 2. Write to disk all the buffers that are **dirty**; i.e., they contain one or more changed database elements. **Unlike** redo logging, we **flush all dirty buffers**, not just those written by committed transactions
 3. Write an **<END CKPT>** record to the log and flush the log



Checkpointing an Undo/Redo Log

• Example

- During the checkpoint, we shall surely flush **A** and **B** to disk
- Redo T_2 and T_3 and ignore T_1
- When we redo T_2 , we do not need to look prior to the $\langle \text{START CKPT}(T_2) \rangle$ record, even though T_2 was active at that time. **Why?**

$\langle \text{START } T_1 \rangle$
 $\langle T_1, A, 4, 5 \rangle$
 $\langle \text{START } T_2 \rangle$
 $\langle \text{COMMIT } T_1 \rangle$
 $\langle T_2, B, 9, 10 \rangle$
 $\langle \text{START CKPT}(T_2) \rangle$
 $\langle T_2, C, 14, 15 \rangle$
 $\langle \text{START } T_3 \rangle$
 $\langle T_3, D, 19, 20 \rangle$
 $\langle \text{END CKPT} \rangle$
 $\langle \text{COMMIT } T_2 \rangle$
 $\langle \text{COMMIT } T_3 \rangle$

How to recover? A crash occurs

→

A crash occurs → **✗**



OUTLINES



17.3 Redo Logging



17.4 Undo/Redo Logging



17.5 Protecting Against Media Failures



Protecting Against Media Failures

- Log
 - Protect us against system failures, where nothing is lost from disk, but temporary data in main memory is lost
- Archiving
 - More serious media failures involve the loss of one or more disks



Protecting Against Media Failures

- Archiving
 - maintaining a copy of the database separate from the database itself
 - Shut down the database for a while
 - Make a backup copy on some storage medium
 - Store the copy in some remote secure location from the database
 - The backup preserve the database state as it existed at the time of the backup
 - If there were a media failure, the database could be restored to this state



Protecting Against Media Failures

- Archiving
 - To advance to a more recent state
 - Provided **the log** had been preserved since the archive copy was made, and the log survived the failure
 - We can use the archive plus remotely stored log to recover, at least up to the point that the log was transmitted to the remote site



Protecting Against Media Failures

- Two levels of archiving
 1. A *full dump*, in which the entire database is copied
 2. An *incremental dump*, in which only those database elements changed since the previous full or incremental dump are copied
- We can restore the database from a **full dump** and its subsequent **incremental dumps**, in a process much like the way a redo or undo/redo log can be used to repair damage due to a system failure
- We copy the **full dump** back to the database, and then in an **earliest-first order**, make the changes recorded by the later **incremental dumps**



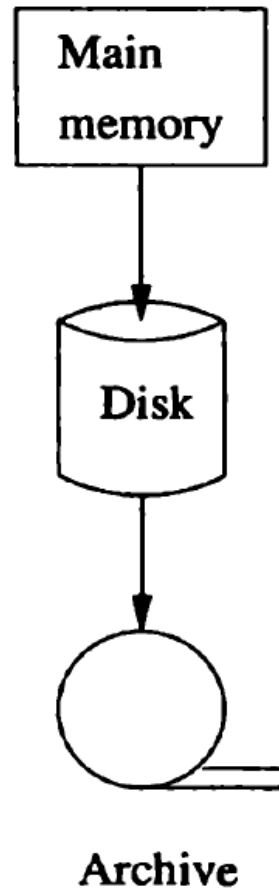
Nonquiescent Archiving

- **The problem**
 - Most databases cannot be shut down for the period of time (possibly hours) needed to make a backup copy
- **Nonquiescent archiving**
 - Analogous to nonquiescent checkpointing
 - Tries to make a copy of the database that existed when the dump began, but database activity may change many database elements on disk during the minutes or hours that the dump takes



Nonquiescent Archiving

- Nonquiescent archiving
 - To restore the database from the archive, the log entries made during the dump can be used to sort things out and get the database to a consistent state



Checkpoint gets data from memory to disk; log allows recovery from system failure

Dump gets data from disk to archive; archive plus log allows recovery from media failure



Nonquiescent Archiving

- Nonquiescent archiving
 - A nonquiescent dump copies the database elements in some fixed order
 - Possibly while those elements are being changed by executing transactions
 - The value of a database element that is copied to the archive may or may not be the value that existed when the dump began
 - As long as the log for the duration of the dump is preserved, the discrepancies can be corrected from the log



Nonquiescent Archiving: Example

Disk	Archive
	Copy <i>A</i>
$A := 5$	
	Copy <i>B</i>
$C := 6$	
	Copy <i>C</i>
$B := 7$	
	Copy <i>D</i>

- Example

- When the dump begins, $A=1, B=2, C=3, D=4$
- At the end of the dump, $A=5, B=7, C=6, D=4$
- The copy of the database in the archive has values $A=1, B=2, C=6, D=4$



Nonquiescent Archiving

- The steps (using either redo or undo/redo, undo is not suitable for use with archiving)
 1. Write a log record **<START DUMP>**
 2. Perform a checkpoint appropriate for whichever logging method is being used
 3. Perform a full or incremental dump of the data disk(s), as desired, making sure that the copy of the data has reached the secure, remote site
 4. Make sure that enough of the log has been copied to the secure, remote site that at least the prefix of the log up to and including the checkpoint in item 2 will survive a media failure of the database
 5. Write a log record **<END DUMP>**



Nonquiescent Archiving

1. Write a log record **<START DUMP>**
 2. Perform a checkpoint appropriate for whichever logging method is being used
 3. Perform a full or incremental dump of the data disk(s), as desired, making sure that the copy of the data has reached the secure, remote site
 4. Make sure that enough of the log has been copied to the secure, remote site that at least the prefix of the log up to and including the checkpoint in item 2 will survive a media failure of the database
 5. Write a log record **<END DUMP>**
- At the completion of the dump, it is safe to throw away log prior to the beginning of the checkpoint *previous* to the one performed in item 2 above



Nonquiescent Archiving: Example

<START DUMP>
<START CKPT (T_1, T_2)>
< $T_1, A, 1, 5$ >
< $T_2, C, 3, 6$ >
<COMMIT T_2 >
< $T_1, B, 2, 7$ >
<END CKPT>
Dump completes
<END DUMP>



A media failure occurs

- Example

- The log survives
- The database is first restored to the values in the archive,
A=1, B=2, C=6, D=4
- Redo T_2
- Undo T_1



Recovery Using an Archive and Log

- Suppose that a media failure occurs
 1. Restore the database from the archive
 - a) Find the most recent full dump and reconstruct the database from it (i.e., copy the archive into the database)
 - b) If there are later incremental dumps, modify the database according to each, earliest first
 2. Modify the database using the surviving log.
Use the method of recovery appropriate to the log method being used



Recovery Using an Archive and Log

- Example
 - A possible undo/redo log

<START DUMP>

<START CKPT (T_1, T_2)>

< $T_1, A, 1, 5$ >

< $T_2, C, 3, 6$ >

<COMMIT T_2 >

< $T_1, B, 2, 7$ >

<END CKPT>

Dump completes

<END DUMP>



...The End of This Lecture...



Q&A

