Chapter 16: Recovery System

Database System Concepts, 6th Ed.

Chapter 16: Recovery System

- Failure Classification
- Storage Structure
- Recovery and Atomicity
- Recovery Algorithms
- Buffer Management
- Failure with Loss of Nonvolatile Strorage
- Early Lock Release and Logical Undo Operations
- ARIES
- Remote Backup Systems

Failure Classification

- **Transaction failure**:
  - **Logical errors**: transaction cannot complete due to some internal error condition (e.g., data 부재, overflow, resource limit 초과)
  - **System errors**: the database system must terminate an active transaction due to an error condition (e.g., deadlock)
- **System crash**: a power failure, other HW/SW 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
  - Copies of data on other disks or Archival backups on tertiary media such as DVD or tapes are needed

Recovery Algorithms

- Consider transaction T_i that transfers \$50 from account *A* to account *B*
  - Two updates: subtract \$50 from A and add \$50 to B
- Transaction T_i requires updates to A and B to be output to the database
  - A failure may occur after one of these modifications have been made but before both of them are made
  - Modifying the database without ensuring that the transaction will commit may leave the database in an inconsistent state
  - Failures that occur just after transaction commits may result in lost updates
- Recovery algorithms have two parts
  1. Actions taken during normal transaction processing to ensure enough information exists to recover from failures (정보기록행위)
  2. Actions taken after a failure to recover the database contents to a state that ensures atomicity, consistency and durability (DB복구행위)

Chapter 16: Recovery System

- Failure Classification
- Storage Structure
- Recovery and Atomicity
- Recovery Algorithms
- Buffer Management
- Failure with Loss of Nonvolatile Strorage
- Early Lock Release and Logical Undo Operations
- ARIES
- Remote Backup Systems

Storage Structure

- **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
  - but may still fail, losing data
- **Stable storage**:
  - a mythical form of storage that survives all failures
  - approximated by maintaining multiple copies on distinct nonvolatile media
  - See book for more details on how to implement stable storage

Stable-Storage Implementation [1/2]

- Mirrored disks: Maintain multiple copies of each block on separate disks (RAID)
- Remote backup: Maintain remote sites to protect against disasters (fire, flooding)
- Block transfer can result in
  - Successful completion
  - Partial failure: destination block has incorrect information
  - Total failure: destination block was never updated
- Data-transfer failures can still result in inconsistent copies
- Protecting storage media from data-transfer failure:
  - 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 with the following check
       * If either copy has an error (bad checksum), overwrite it by the other copy
       * If both have no error, but are different, overwrite the 2nd block by the 1st block
    3. The output is completed only after the second write successfully completes

  - If system failures during the above steps → Recovery procedure in next slide

Stable-Storage Implementation [2/2]

- Copies of a block may differ due to failure during output operation

  To recover from system failure during output operation:

  1. First find inconsistent blocks:
     1. *Expensive solution*: Compare the two copies of every disk block
     2. *Better solution*: (used in hardware RAID systems)
        - Record in-progress disk writes on non-volatile RAM
        - Use this information during recovery to find blocks that may be inconsistent, and only compare copies of these
  2. Second fix the inconsistent blocks:

     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 2nd block by the 1st block

Data Access during Transaction [1/2]

- **Physical blocks** are those blocks residing on the disk
- **Buffer blocks** are the blocks residing temporarily in main memory
- Disk buffer: the area of memory where blocks reside temporarily
- 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
- We assume, for simplicity, that each data item fits in, and is stored inside, a single block

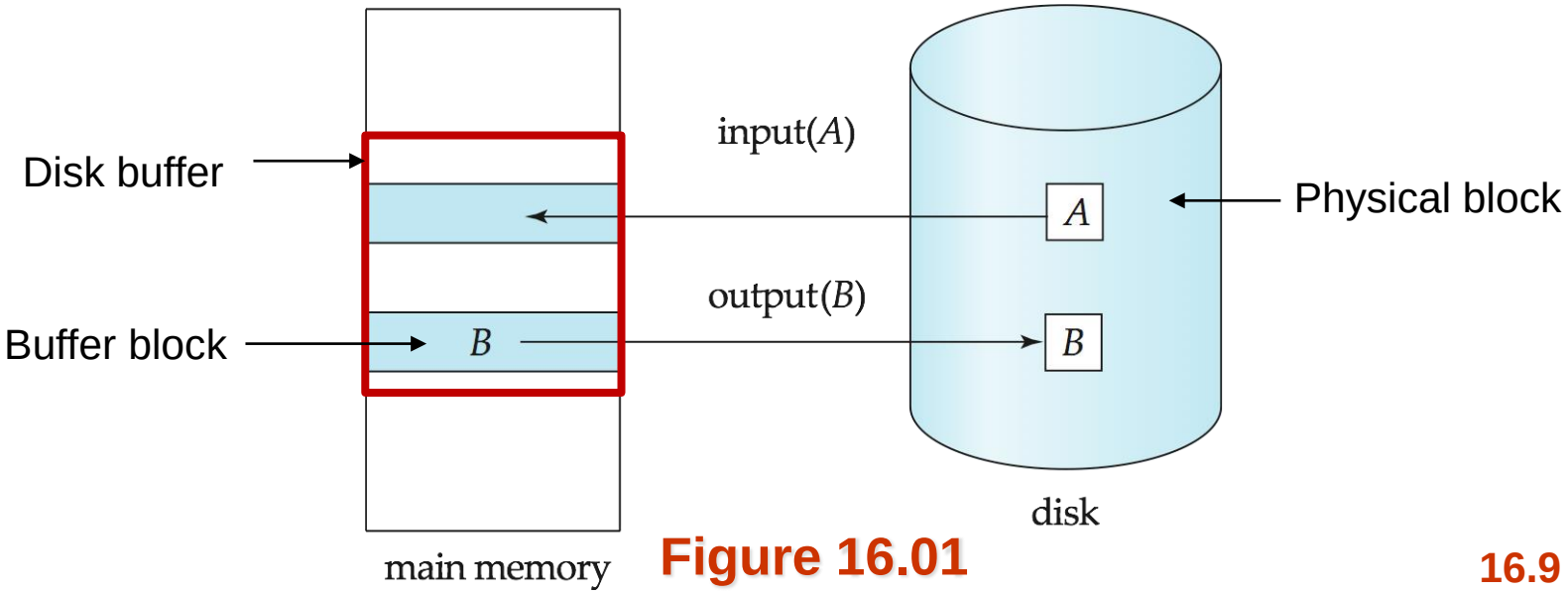
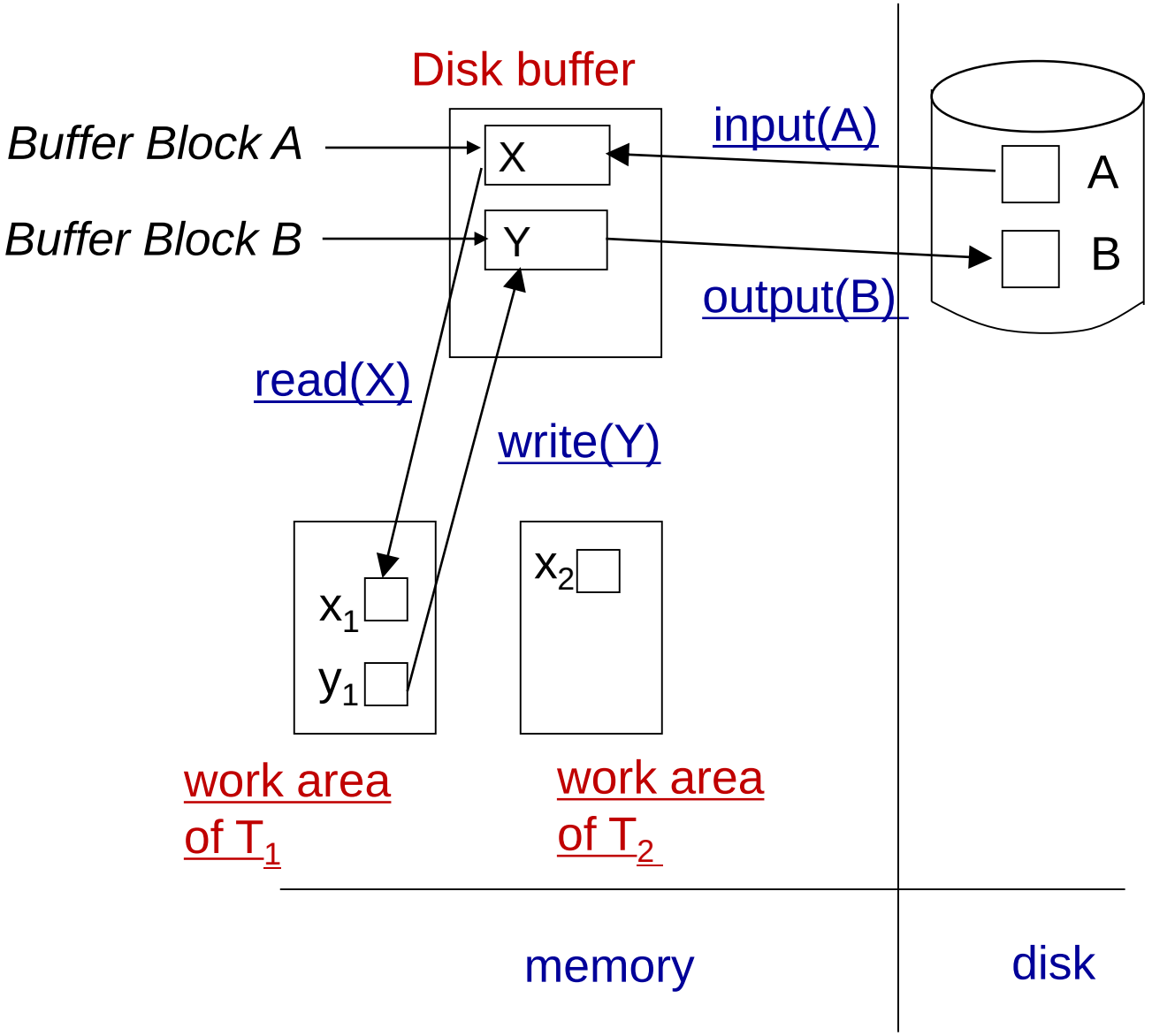


Figure 16.01

Data Access during Transaction [2/2]

- 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
- Transferring data items between system buffer blocks and its private work-area done by:
  - **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
  - **Note: output**(B_X) need not immediately follow **write**(X). System can perform the **output** operation when it deems fit
- Transactions
  - Must perform **read**(X) before accessing X for the first time (subsequent reads can be from local copy)
  - **write**(X) can be executed at any time before the transaction commits

Example of Data Access during Transaction



Chapter 16: Recovery System

- Failure Classification
- Storage Structure
- Recovery and Atomicity
- Recovery Algorithms
- Buffer Management
- Failure with Loss of Nonvolatile Strorage
- Early Lock Release and Logical Undo Operations
- ARIES
- Remote Backup Systems

Recovery and Atomicity

- To ensure atomicity despite failures
  - we first output information describing the modifications to stable storage before modifying the database itself
- We study **log-based recovery mechanisms** in detail
  - We first present key concepts
  - And then present the actual recovery algorithm
- Less used alternative: **shadow-paging** (brief details in book)

Log-Based Recovery

- 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
- When transaction T_i starts, it registers itself by writing a
  $<T_i$ **start**$>$ log record
- *Before* T_i executes **write**(X), a log record
  $<T_i, X, V_1, V_2>$ is written,

  where V_1 is the value of X before the write (the **old value**), and
  V_2 is the value to be written to X (the **new value**)
- When T_i finishes it last statement, the log record
  $<T_i$ **commi**t$>$ is written

- Two approaches using logs
  - Deferred database modification
  - Immediate database modification

At the moment, Assume serial execution of Transactions T0, T1, T3…

Example of Undo & Redo Log

- 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)

Figure 16.2
Portion of the System Log corresponding to T_0 and T_1

<T_0 start>
<T_0, A, 1000, 950>
<T_0, B, 2000, 2050>
<T_0 commit>
<T_1 start>
<T_1, C, 700, 600>
<T_1 commit>

Figure 16.3
State of System Log and Database corresponding to T_0 and T_1

Log	Database
<T_0 start>	
<T_0, A, 1000, 950>	
<T_0, B, 2000, 2050>	
	$A = 950$
	$B = 2050$
<T_0 commit>	
<T_1 start>	
<T_1, C, 700, 600>	
	$C = 600$
<T_1 commit>	

Database Modification

- The **immediate-modification** scheme allows updates of an uncommitted transaction to be made to the buffer, or the disk itself, before the transaction commits
- Update log record must be written *before* database item is written
  - We assume that the log record is output directly to stable storage
    - Will see later that how to postpone log record output to some extent
  - Output of updated blocks to stable storage 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 in memory
- In the textbook, we cover **the recovery algorithm in immediate-modification**
  - 2PL CC is a natural fit with the immediate-modification scheme

- The **deferred-modification** scheme performs updates to buffer/disk only at the time of transaction commit
  - Simplifies some aspects of recovery
  - But has overhead of storing local copy
  - MVCC , snapshot isolation CC is a natural fit

Immediate Database Modification Example

- 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)

- Note: B_X denotes block containing X

Log	Write	Output
<T_0 **start**>		
<T_0, A, 1000, 950>		
<T_0, B, 2000, 2050		
	A = 950	
	B = 2050	
<T_0 **commit**>		
<T_1 **start**>		
<T_1, C, 700, 600>		
	C = 600	
		B_B , B_C
<T_1 **commit**>		
		B_A

B_C output before T_1 commits

B_A output after T_0 commits

트랜잭션이 커밋 되지 않아도 변경된 블록이 Disk에 Output될 수 있고,
블록을 Output하는 순서는 트랜잭션에서 수행된 순서를 따르지 않음을 보여주고 있음
// **로그는 순서대로, 디스크 반영은 DBMS가 임의로!**

Concurrency Control and Recovery

- Base Assumptions
  - With concurrent transactions, all transactions share a single disk buffer and a single log
    - A buffer block can have data items updated by one or more transactions
  - If a transaction T_i has modified an item, no other transaction can modify the same item until T_i has committed or aborted
    - Otherwise how to perform undo if T1 updates A, then T2 updates A and commits, and finally T1 has to abort?
    - This assumption can be ensured by obtaining exclusive locks on updated items and holding the locks till end of transaction (strict 2PL CC)
      - 그러나 Snapshot isolation CC 나 Validation-based CC 에서도 적용가능
  - Log records of different transactions may be interspersed in the log
    - intersperse 흩어놓다, 산재시키다

Transaction Commit

- A transaction is said to have committed when its commit log record is output to stable storage
  - all previous log records of the transaction must have been output already
- Writes performed by a transaction may still be in the buffer when the transaction commits, and may be output later
  - 변경된 buffer block들이 output 되기전에 failure가 생겨도 log record는 stable storage에 있으므로 redo하면 됨

Undo and Redo Operations

- **Undo** of a log record <T_i, X, V_1, V_2> writes the **old** value V_1 to X
- **Redo** of a log record <T_i, X, V_1, V_2> writes the **new** value V_2 to X
- **Undo and Redo of Transactions**
  - **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
    - each time a data item X is restored to its old value V a special log record <T_i , X, V> is written out
    - when undo of a transaction is complete, a log record <T_i **abort**> is written out.
  - **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
    - No logging is done in this case

Undo and Redo on Recovering from Failure

- When recovering after failure:
  - Transaction T_i needs to be undone if the log
    - contains the record <T_i **start**>,
    - but does not contain either the record <T_i **commit**> *or* <T_i **abort**>.
  - Transaction T_i needs to be redone if the log
    - contains the records <T_i **start**>
    - and contains the record <T_i **commit**> *or* <T_i **abort**>

- Note that If transaction T_i was undone earlier and the <T_i **abort**> record written to the log, and then a failure occurs, on recovery from failure T_i is redone
  - **such a redo redoes all the original actions *including the steps that restored old values***
    - Known as **repeating history**
    - Seems wasteful, but simplifies recovery greatly

Immediate DB Modification Recovery Example

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

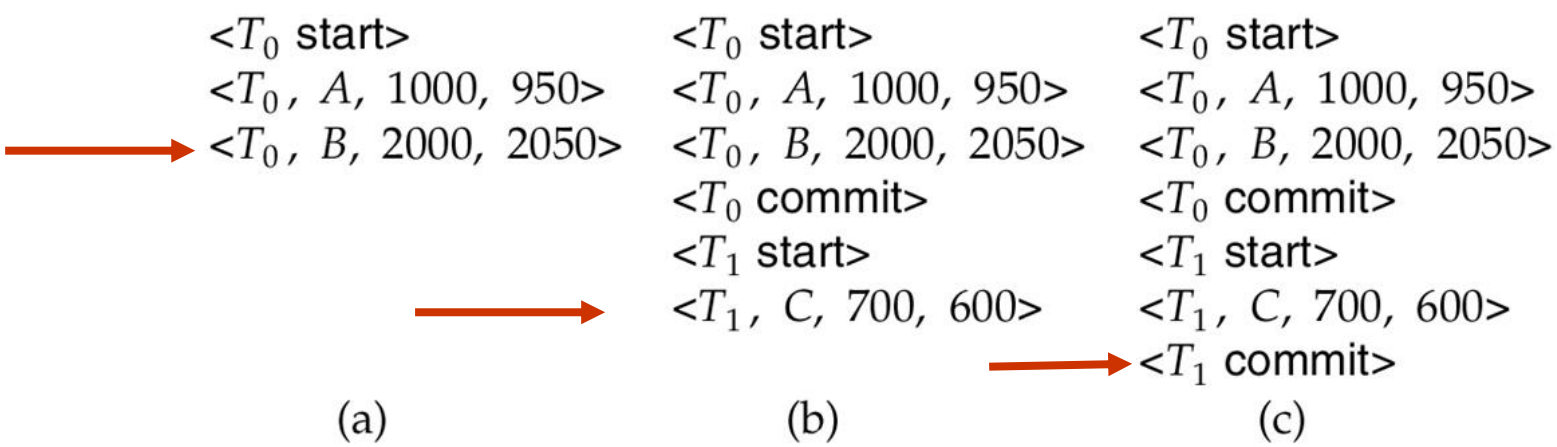


Fig 16.04

Recovery actions in each case above are:

(a) undo (T_0): B is restored to 2000 and A to 1000, and log records
Log records <T_0, B, 2000>, <T_0, A, 1000>, <T_0, **abort**> are written out

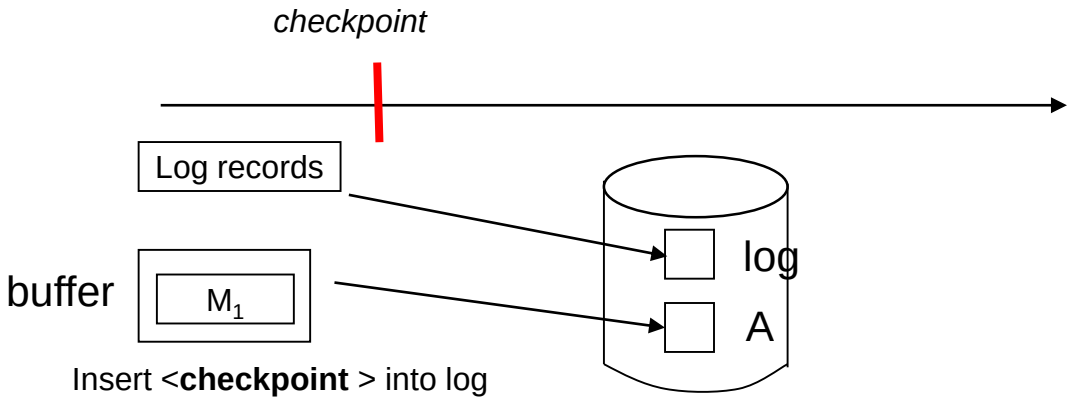
(b) redo (T_0) and undo (T_1): *A* and *B* are set to 950 and 2050 and C is restored to 700
Log records <T_1, C, 700>, <T_1, **abort**> are written out

(c) redo (T_0) and redo (T_1): A and B are set to 950 and 2050 respectively
Then *C* is set to 600

Undo하는 이유: buffer에 반영이 되었었으므로 혹시 DB에도 반영되었을 가능성도 있으므로, 다시 buffer에 예전값을 써주어서 DB에 반영이 되었어도 다시 원상복귀시키기위해

Checkpoints [1/2]

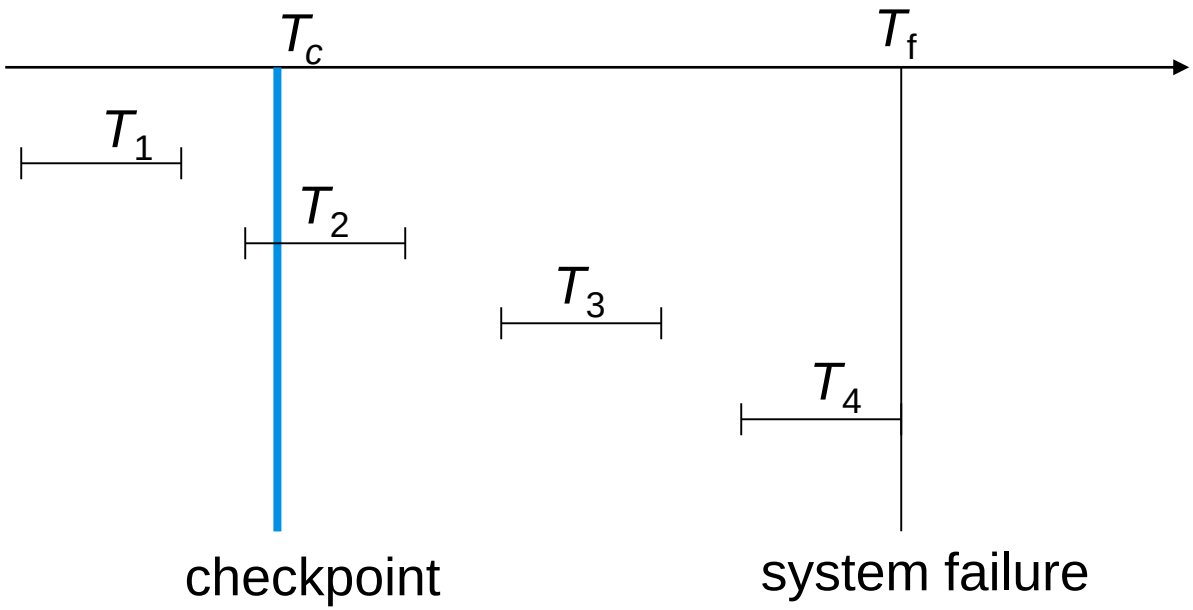
- Redoing/undoing all transactions recorded in the log can be very slow
  1. processing the entire log is time-consuming if the system has run for a long time
  2. we might unnecessarily redo transactions which have already output their updates to the database
- Streamline recovery procedure by periodically 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** *L*> onto stable storage where *L* is a list of all transactions active at the time of checkpoint
     - All updates are stopped while doing checkpointing



Checkpoints [2/2]

- 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
  - Scan backwards from end of log to find the most recent <**checkpoint** *L*> record
    - Only transactions that are in *L* or started after the checkpoint need to be redone or undone
    - Transactions that committed or aborted before the checkpoint already have all their updates output to stable storage
- Some earlier part of the log may be needed for undo operations
  - Continue scanning backwards till a record <T_i **start**> is found for every transaction T_i in *L*
    - Parts of log prior to earliest <T_i **start**> record above are not needed for recovery, and can be erased whenever desired
- Bothersome: All updates are stopped while doing checkpointing
  - Solution: Fuzzy-checkpoint scheme (we cover it later)
    - transactions are allowed updates while buffer blocks are being written

Example of Checkpoints

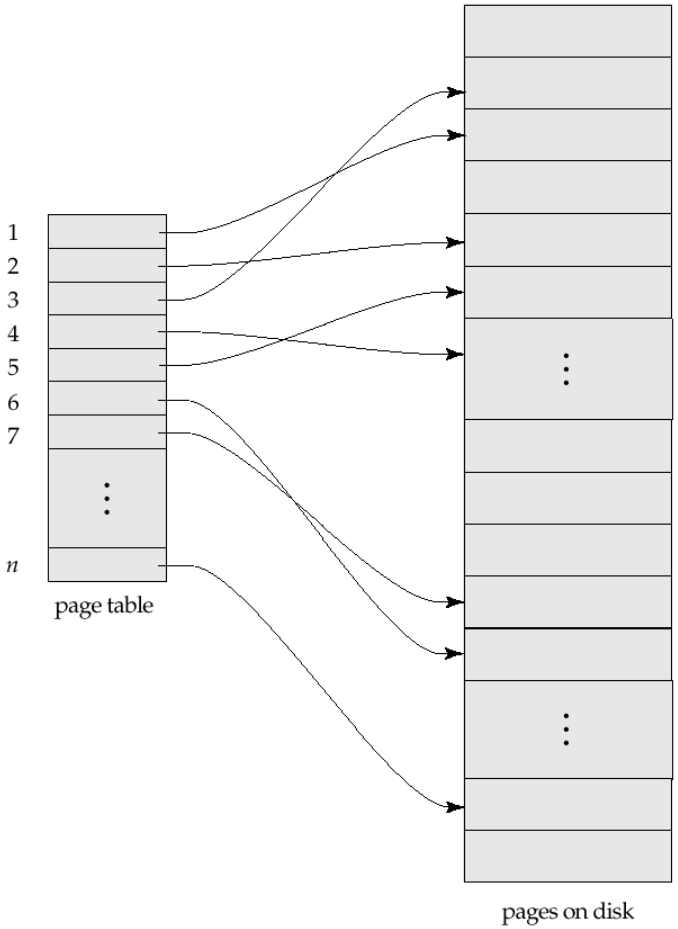


- T_1 can be ignored (updates already output to disk due to checkpoint)
- *Undo* T_4 (remember Undo first, then Redo)
- *Redo* T_2 and T_3
  - Be careful for the order of undo & redo
  - Suppose T1 (x =3); T2 (x = x + 1); T3 (x = x +1); T4 (x = x * 2)

Shadow Paging [1/3]

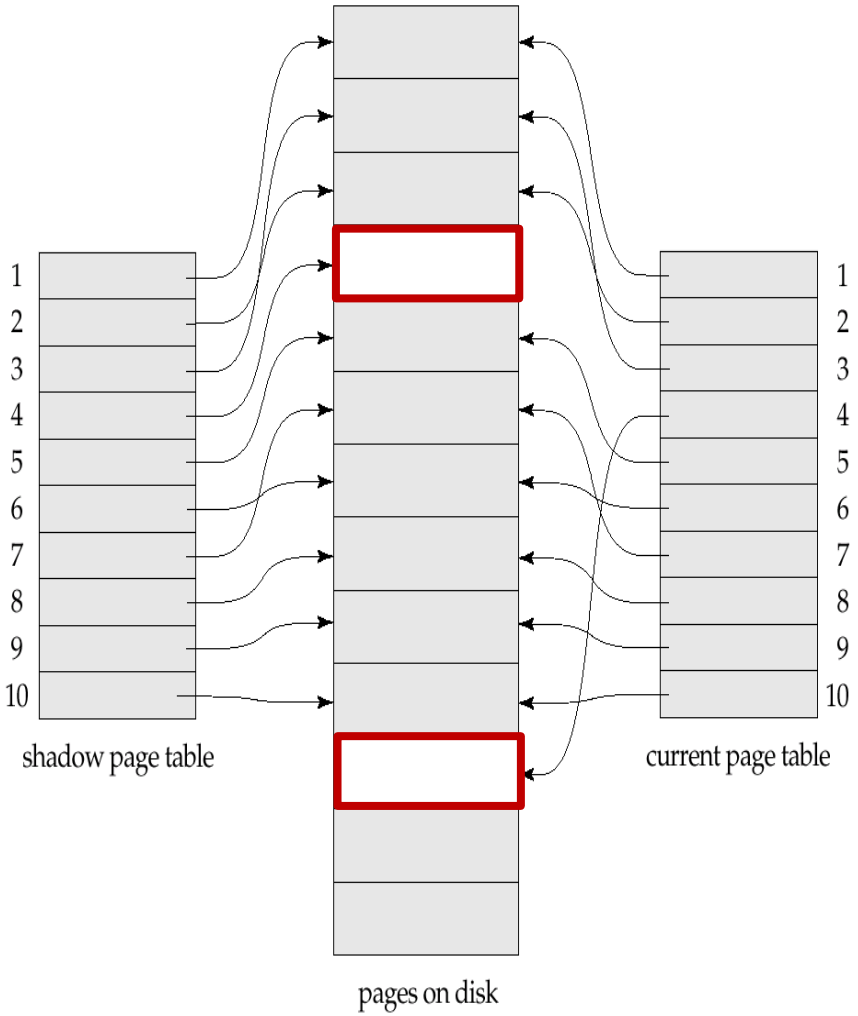
- **Shadow paging** is an alternative to log-based recovery
  - Working only when transactions execute serially
- Idea: maintain *two* page tables during the lifetime of a transaction
  - the **current page table**, and the **shadow page table**
- Store the shadow page table in nonvolatile storage, such 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



Shadow Paging

Shadow and current page tables after write to page 4



Shadow Paging [2/3]

- 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)

Show Paging [3/3]

- 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^+-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

Chapter 16: Recovery System

- Failure Classification
- Storage Structure
- Recovery and Atomicity
- Recovery Algorithms
- Buffer Management
- Failure with Loss of Nonvolatile Strorage
- Early Lock Release and Logical Undo Operations
- ARIES
- Remote Backup Systems

Recovery Algorithm [1/3]

- **Logging** (during normal operation): (log record 생성)
  - <T_i **start**> at transaction start
  - <T_i, X_j, V_1, V_2> for each update
  - <T_i **commit**> at transaction end
- **Transaction rollback (during normal operation): only "undo" phase**
  - Let T_i be the transaction to be rolled back
  - Scan log backwards from the end, and for each log record of T_i of the form <T_i, X_j, V_1, V_2>
    - perform the undo by writing V_1 to X_j
    - write a log record <T_i , X_j, V_1>
      - such log records are called **compensation log records**
  - Once the record <T_i **start**> is found, stop the scan and write the log record <T_i **abort**>

Transaction이 roll-back하는 이유?
Deadlock resolution의 victim으로 선택된 경우, CC Scheme에서 roll-back이 요구되는 경우

Recovery Algorithm [2/3]

- **Recovery from failure**: need 2 phases
  - **Redo phase**: replay updates of all transactions, whether they committed, aborted, or are incomplete
  - **Undo phase**: undo all incomplete transactions

- **Redo (Replay) phase**:
  1. Find last <**checkpoint** *L*> record, and set undo-list to *L*
  2. Scan forward from above <**checkpoint** *L*> record
     1. Whenever a record <T_i, X_j, V_1, V_2> is found, redo it by writing V_2 to X_j
     2. Whenever a log record <T_i **start**> is found, add T_i to undo-list
     3. Whenever a log record <T_i **commit**> *or* <T_i **abort**> is found, remove T_i from undo-list

(redo phase에서는 redo작업에 대한 log record를 만들필요가 없다!)

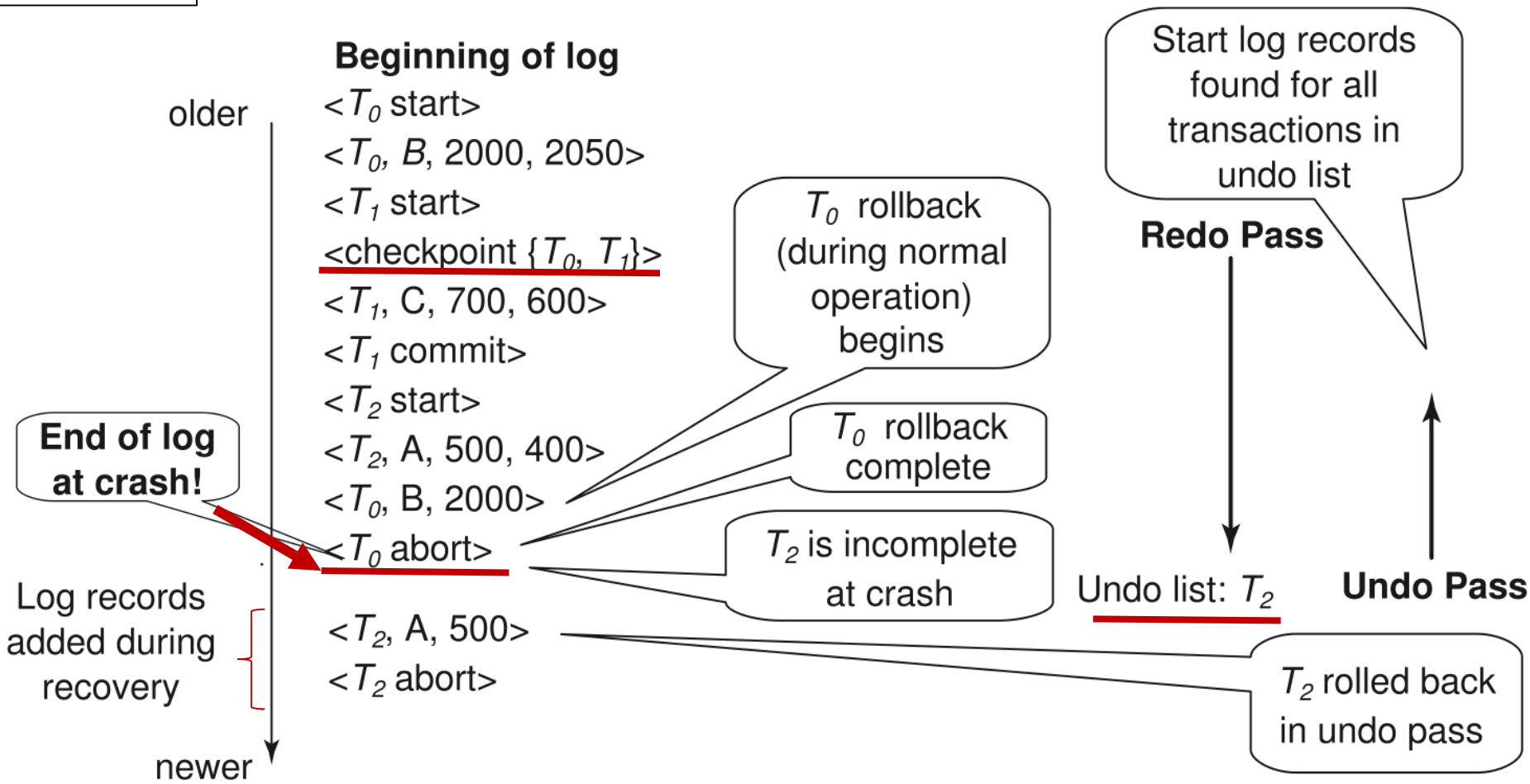
Recovery Algorithm [3/3]

- **Undo phase:**
  1. Scan log backwards from end
     1. Whenever a log record <T_i, X_j, V_1, V_2> is found where T_i is in undo-list perform same actions as for transaction rollback:
        1. perform undo by writing V_1 to X_j
        2. write a log record <T_i , X_j, V_1>
     2. Whenever a log record <T_i **start**> is found where T_i is in undo-list,
        1. Write a log record <T_i **abort**>
        2. Remove T_i from undo-list
     3. Stop when undo-list is empty
        - i.e. <T_i **start**> has been found for every transaction in undo-list

- After undo phase completes, normal transaction processing can commence

Example of Recovery

Fig 16.05



Checkpoint순간에 진행중인 T0, T1을 undo-list넣고, 내려오면서 적절한 redo를 수행하고, start를 만나면 undo-list넣고, commit이나 abort를 만나면 undo-list에 빼고, Crash point까지 와서도 undo-list에 남아있는 transaction들은 undo를 수행

Chapter 16: Recovery System

- Failure Classification
- Storage Structure
- Recovery and Atomicity
- Recovery Algorithms
- Buffer Management
- Failure with Loss of Nonvolatile Strorage
- Early Lock Release and Logical Undo Operations
- ARIES
- Remote Backup Systems

Log Record Buffering

- **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

- **Write-ahead logging** (**WAL)** rule must be followed if log records are buffered:
  - 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
    - Strictly speaking WAL only requires undo information to be output
  - 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 <T_i **commit**> has been output to stable storage

Database Buffering [1/2]

- 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

- By the WAL rule, 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

<T_0 **start**>
<T_0, A, 1000, 950>
Transaction T_0 issues read(B)

Suppose the block (having A) is chosen to be output to disk for the block (having B) to come to the memory. ➔ <T_0, A, 1000, 950> should be output to a stable storage first

- The recovery algorithm supports **the no-force policy**
  - **No-force policy:** updated data blocks need not be written to disk when transaction commits
    - WAL에 의해서 문제가 발생해도 복구 가능
  - **Force policy**: requires updated blocks to be written at commit (more expensive)

Database Buffering [2/2]

- The recovery algorithm supports **the steal policy**
  - blocks containing updates of uncommitted transactions can be written to disk, even before the transaction commits
  - By the WAL rule, log records with undo information for the updates are output to the log on stable storage first
- 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**: X-larch, S-latch
- **To output a block to disk**
  1. First acquire **an X-latch** on the block
     1. Ensures no update can be in progress on the block
  2. Then perform a **log flush**
  3. Then output the block to disk
  4. Finally release the X-latch on the block

OS Role in Buffer Management [1/2]

- Database buffer can be implemented either
  - DBMS reserves part of main memory to serve as a buffer
  - DBMS implements its buffer within in the virtual memory provided by OS

- Implementing database buffer in reserved main-memory has drawbacks:
  - OS가 main-memory의 일부에 대해서 관리능력을 DBMS에 넘겨주는것이므로
  - 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

- Database buffers are implemented in virtual memory in spite of dual paging problem
  - When OS 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!

OS Role in Buffer Management [2/2]

- One or other in the previous slide must be chosen unless the following functionality
  - Ideally when OS needs to evict a page from the buffer,
    - OS should pass control to DBMS
    - OS should output the page to DBMS instead of to swap space (making sure to output log records first), if it is modified
    - Release the page from the buffer, for the OS to use
  - Thus, dual paging can thus be avoided
  - But common OSs do not support such functionality
  - OS for only-database environment may be designed in the future

Evict: 쫓아내다, 퇴거시키다

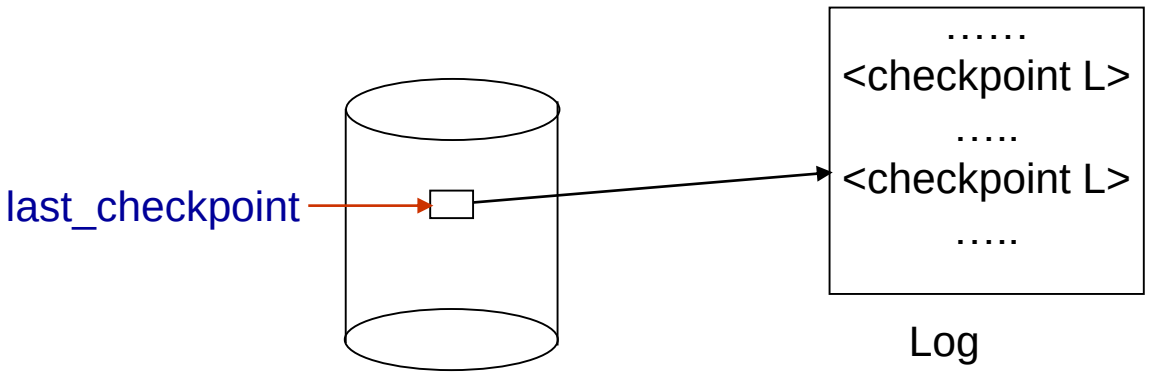
Fuzzy Checkpointing [1/2]

- IBM team (Lindsay et al., Notes on Distributed DB, Cambridge Univ, 1980)
- To avoid long interruption of normal processing during checkpointing
- **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 the list *M* of modified buffer blocks
  4. Now permit transactions to proceed with their actions
  5. Flush buffer: Output to disk all modified buffer blocks in the 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

Fuzzy Checkpointing [2/2]

- Even if there is a failure during checkpointing, log-force guarantee safe recovery

- 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 checkpoint, are handled safely



Normal Checkpointing Example

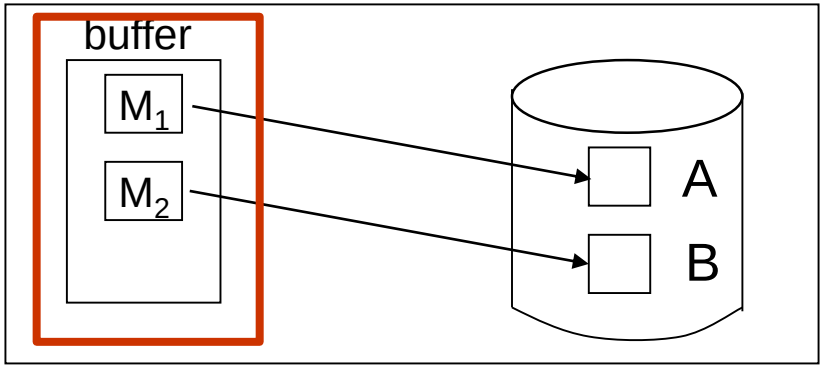
Assume Last_chkpoint L_1

(1) Stop all transactions
(2) Begin chkpoint L_2

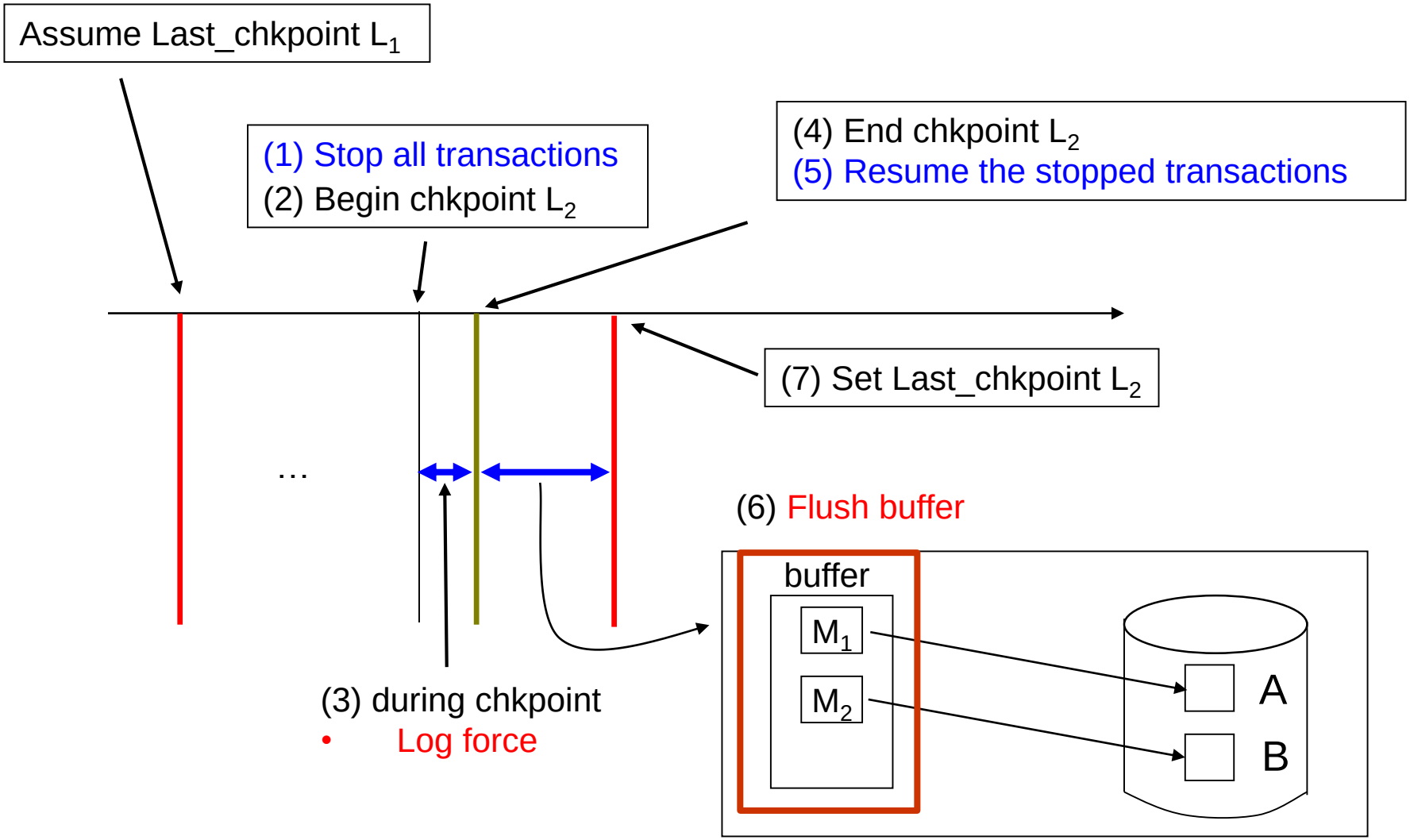
(4) End chkpoint L_2
(5) Set Last_chkpoint L_2
(6) Resume the stopped transactions

…

(3) During checkpoint
- Log force
- Flush buffer



Fuzzy Checkpointing Example



Chapter 16: Recovery System

- Failure Classification
- Storage Structure
- Recovery and Atomicity
- Recovery Algorithms
- Buffer Management
- Failure with Loss of Nonvolatile Storage
- Early Lock Release and Logical Undo Operations
- ARIES
- Remote Backup Systems

Failure with Loss of Nonvolatile Storage

- 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
    - 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
- To recover from disk failure
  - Restore database from the 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** **(**Similar to fuzzy checkpointing)

Chapter 16: Recovery System

- Failure Classification
- Storage Structure
- Recovery and Atomicity
- Recovery Algorithms
- Buffer Management
- Failure with Loss of Nonvolatile Storage
- Early Lock Release and Logical Undo Operations
- ARIES
- Remote Backup Systems

Recovery with Early Lock Release

- The previous recovery schemes assume
  - Locks are released only just before commit (such as Strict 2PL)
  - 트랜잭션의 commt전의 중간 값을 다른 트랜잭션이 볼 수 없는 상황을 전제

- Support for high-concurrency locking techniques, such as those used for B^+-tree concurrency control, which release locks early
  - Remember the crabbing cc protocol or B-link tree cc protocol
  - Commit안된 트랜잭션의 중간 값을 다른 트랜잭션이 볼 수 있는 상황
  - Transactions which have read the data associated with early lock release can not be undone with the log record
  - Need "Logical undo" by compensating transactions

- Logical-undo recovery executes exactly the same actions as normal processing
  - based on "repeating history" (backward log scan)

Logical Undo Logging

- Operations like B+-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+-tree
  - Instead, insertions (resp. deletions) are undone by executing a deletion (resp. insertion) operation (known as **logical undo**)

- Transaction T가 rollback을 할때에 T의 중간 값을 read한 다른 transaction들을 cascadingly rollback시켜야 하는데 ➔ Physical undo를 쓰면 Btree 같은 구조는 too much loss ➔ logical undog하면 data semantics를 유지하며 다른 transaction들을 그냥 진행

- 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 undo operations**
    - delete of tuple, to undo insert of tuple
      - If DBMS allows early lock release on space allocation information
    - subtract amount deposited, to undo deposit
      - If DBMS allows early lock release on bank balance

Physical Redo

- Physical undo를 logical undo로 교체하더라고, redo는 예전모델처럼 Physical redo
- 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

Operation Logging [1/2]

- Operation logging is done as follows:
  1. When operation starts, <T_i, O_j, **operation-begin**> is logged, where O_j 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, <T_i, O_j, **operation-end**, U> 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>

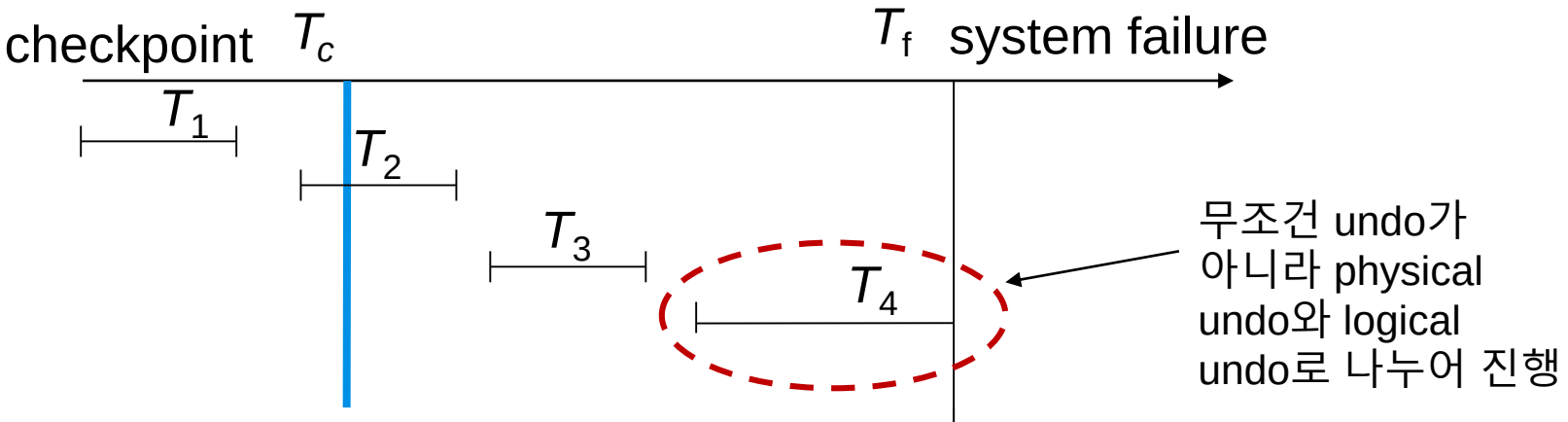
<T1, Y, 45, RID7>

Physical redo of steps in insert

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

Operation Logging [2/2]

- Undo 를 해야 하는 경우
  - If crash/rollback occurs before operation completes: (T_4 에서 진행중인 operation)
    - 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:(T_4에서 완료된 operation 이지만 early-lock-release때문에 rollback해야 하는데 physical undo가 너무 부담)
    - the **operation-end** log record is found, and in this case
    - logical undo is performed using *U*
- Redo 를 해야 하는 경우 (T_2, T_3)
  - Redo of operation (after crash) still uses physical redo information



Recovery from Transaction Rollback with Logical Undo [1/2]

- Rollback of transaction T_i , Scan the log backwards:

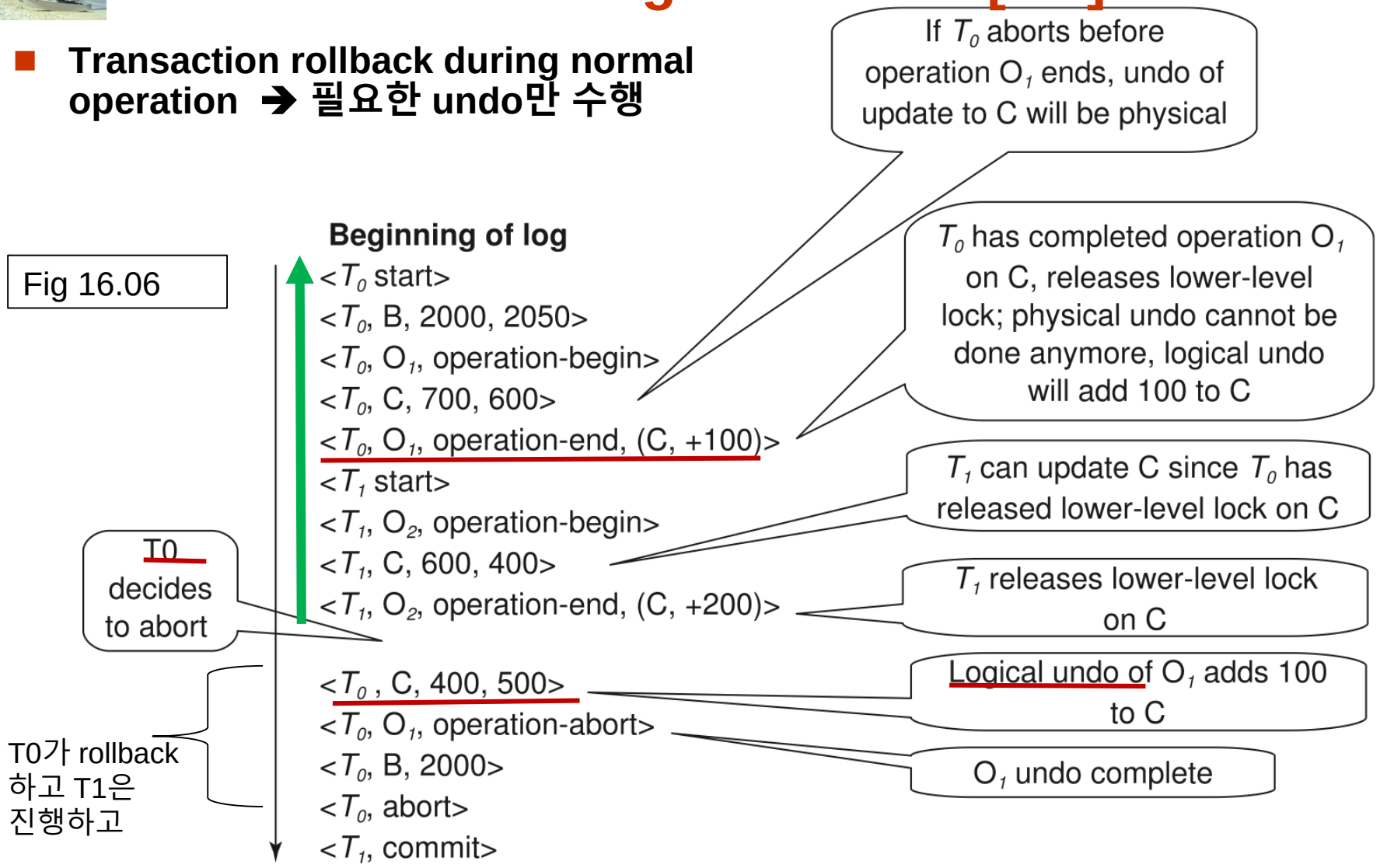
1. If a log record $<T_i, X, V_1, V_2>$ is found, perform the (physical) undo and log a special redo-only log record $<T_i, X, V_1>$
2. If a $<T_i, O_j,$ **operation-end**, $U>$ record is found
   - Rollback the operation logically (logical undo) using the undo information U
     - Updates performed 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 $<T_i, O_j,$ **operation-abort**$>$
   - Skip all preceding log records for T_i until $<T_i, O_j$ **operation-begin**$>$ is found
3. If a redo-only log record is found, ignore it
4. If a $<T_i, O_j,$ **operation-abort**$>$ record is found:
   - skip all preceding log records for T_i until $<T_i, O_j,$ **operation-begin**$>$ is found
5. Stop the scan when the record $<T_i,$ start$>$ is found
6. Add a $<T_i,$ **abort**$>$ record to the log

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

Recovery from Transaction Rollback with Logical Undo [2/2]

- **Transaction rollback during normal operation ➔ 필요한 undo만 수행**

Fig 16.06



** T0는 abort를 해야하는것이고 early-lock-release로 T0의 중간값을 읽은 transaction들이 정상수행이 되도록 수습하기위해서 logical undo를 행하는것

Transaction Rollback: Another Example

- Example with a complete and an incomplete operation

<T1, start>

<T1, O1, operation-begin>

….

<T1, X, 10, K5>

<T1, Y, 45, RID7>

} insert에 관련된 undo목적의 log records

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

<T1, O2, operation-begin>

<T1, Z, 45, 70>

← T1 Rollback begins here

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

<T1, Y, .., ..>

… } ← Normal redo records for logical undo of O1 (delete I9, K5, RID7)

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

T1 roll-back으로 생성된 log-records들을 가지고 다시 recovery

<T1, abort>

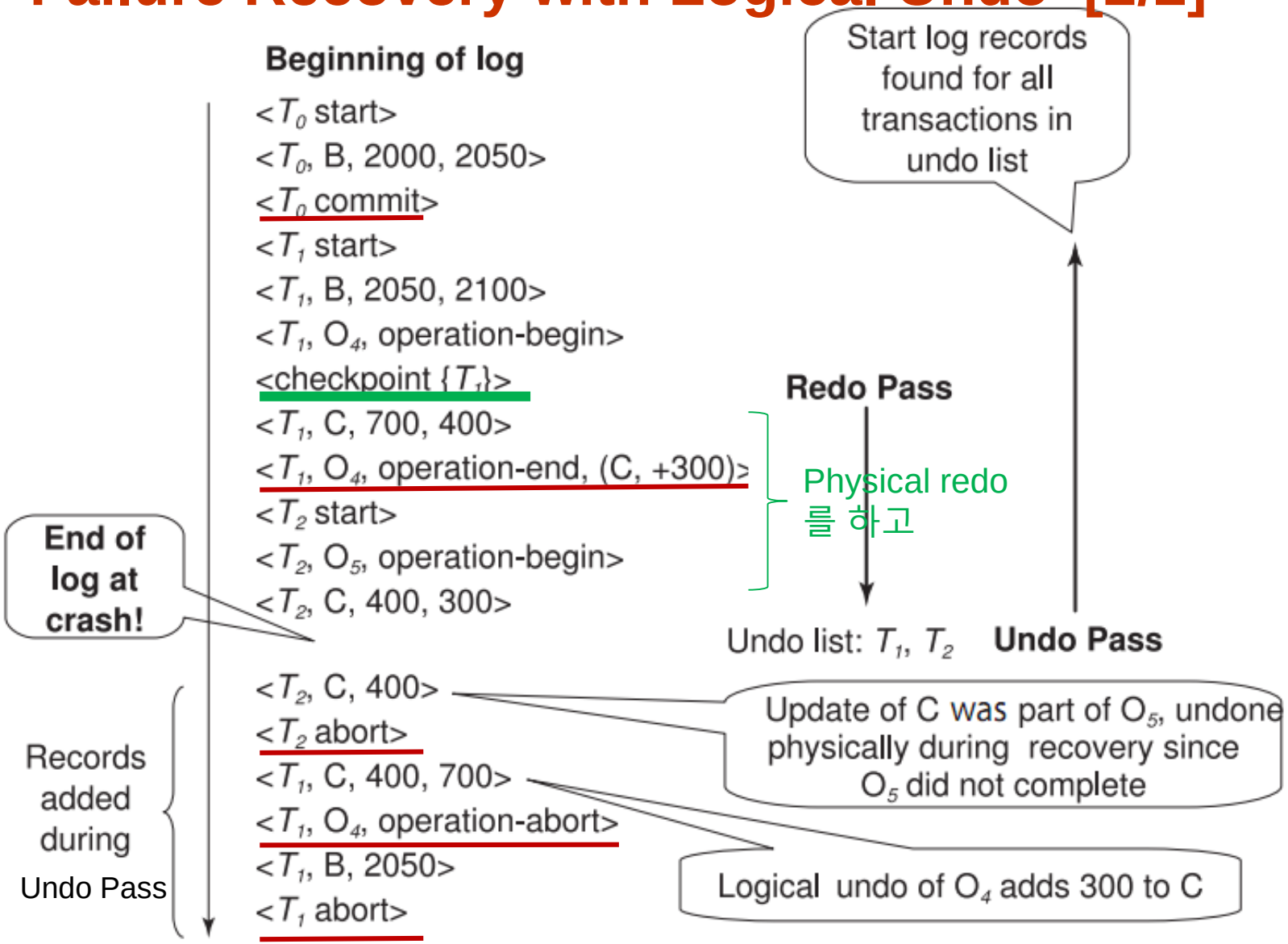
Failure Recovery with Logical Undo [1/2]

- Transaction이 자발적으로 rollback을 할때의 상황이 아니라
  Failure가 생겨서 recovery 를 하는 상황이므로 log의 checkpoint에 의존하여 recovery작업을 시작

- Re-Do phase: Checkpoint이후에 log record를 순서대로 보면서
  - Commit된 transaction들은 log record를 이용하여 physical redo 를 수행하고 (physical redo행위를 log record로 남기는데, 나중에 혹시 모를 redo용도로)
  - <commit> log record가 없는 transaction들은 undo-list에 담고

- Un-Do phase: log record를 역순으로 보면서 Undo-List에 있으면 undo 작업을 함
  - Operation-end가 없는 operation관련 log record 면 physical undo
  - Operation-end가 있는 operation이면 logical undo
  - Undo의 작업은 log에 새로운 undo관련 log record를 기입해야 함

Failure Recovery with Logical Undo [2/2]



- Redo-Pass에서는 physical redo를 하고 undo-list 만들고
- Undo-Pass에서 physical undo와 logical undo가 수행하면서 관련 log record 생성

Recovery from Failure Algorithm with Logical Undo [1/2]

Basically same as earlier algorithm, except for changes described earlier for transaction rollback

Recovery 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_i **start**> is found T_i is added to *undo-list*
      - Whenever <T_i **commit**> or <T_i **abort**> is found, T_i 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

Recovery from Failure Algorithm with Logical Undo [2/2]

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 <T_i **start**> is found for a transaction T_i in *undo-list*, write a <T_i **abort**> log record
   - Stop scan when <T_i **start**> 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

Chapter 16: Recovery System

- Failure Classification
- Storage Structure
- Recovery and Atomicity
- Recovery Algorithms
- Buffer Management
- Failure with Loss of Nonvolatile Storage
- Early Lock Release and Logical Undo Operations
- ARIES
- Remote Backup Systems

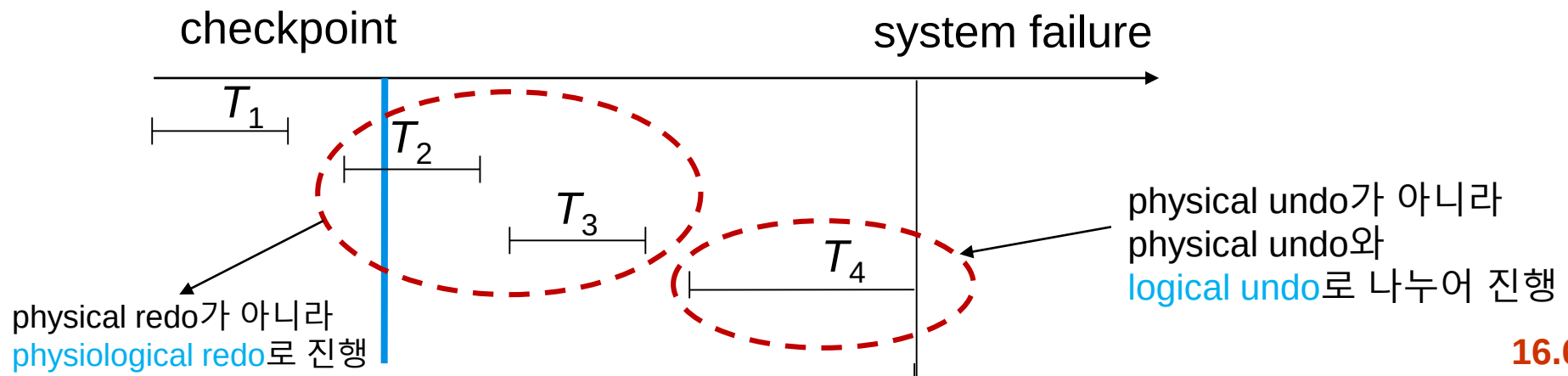
ARIES "State-of-the-Art" Recovery Algorithm

- **Algorithms for Recovery and Isolation Exploiting Semantics** (ARIES)
  - ARIES simplifies the earlier recovery algorithm
    - incorporates numerous optimizations to reduce overheads during normal processing and to speed up recovery
  - C. Mohan, "ARIES/KVL", VLDB 1990 / C. Mohan, "Commit LSN", VLDB 1990
    - IBM DB2 and MS SQL Server support ARIES
  - Oracle: Lahiri et al, "Fast-Start: Quick-Fault Recovery in Oracle", SIGMOD 2001

- Unlike the recovery algorithm described earlier, ARIES uses
  1. 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 logging
  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
     - More coming up on each of the above …

ARIES Optimizations

- **Physiological redo logging**
  - Affected page is physically identified, action within page can be logical
    - Used to reduce physical redo logging overheads
  - When a record is deleted and all other records have to be moved to fill the deleted hole, the physical redo would require logging of old and new values for much of the page
    - Physiological redo can log just the record deletion
  - Requires page to be output to disk atomically
    - Incomplete page output can be detected by checksum techniques,
      - But extra actions are required for recovery
      - Treated as a media failure



Psysiological Redo Logging for Deletion

1개 record를 delete해서 많은 record들의 이동이 있었다면

- Physical logging: 영향받은 페이지들의 before image와 after image를 다 log에 저장하는 기법
  - 블록 레벨 수준으로 기록하기 때문에 redo가 빠름
  - 로그 저장공간이 많이 필요

- Logical logging: delete record의 operation 내용만 log에 저장하는 기법
  - 로그 저장공간이 적게 사용하지만, 디스크 주소 레벨 수준의 기록이 없기 때문에 영향받은 페이지를 일일히 찾아서 redo를 해야 함 ➔ redo 시간 증가

- Physiological logging : 위 1~2 방식을 절충함
  - 영향받은 페이지주소는 기록하고 영향받은 페이지의 before image나 after image를 저장하지 않고 수행된 operation을 기록
  - Redo를 할때 영향받은 페이지들은 찾을 필요 없음
  - 페이지 내에서는 logical logging을 사용하므로 physical logging보다 로그 저장 공간을 덜 사용함

Physiological Redo Logging: 특정 Attribute값을 update했을때

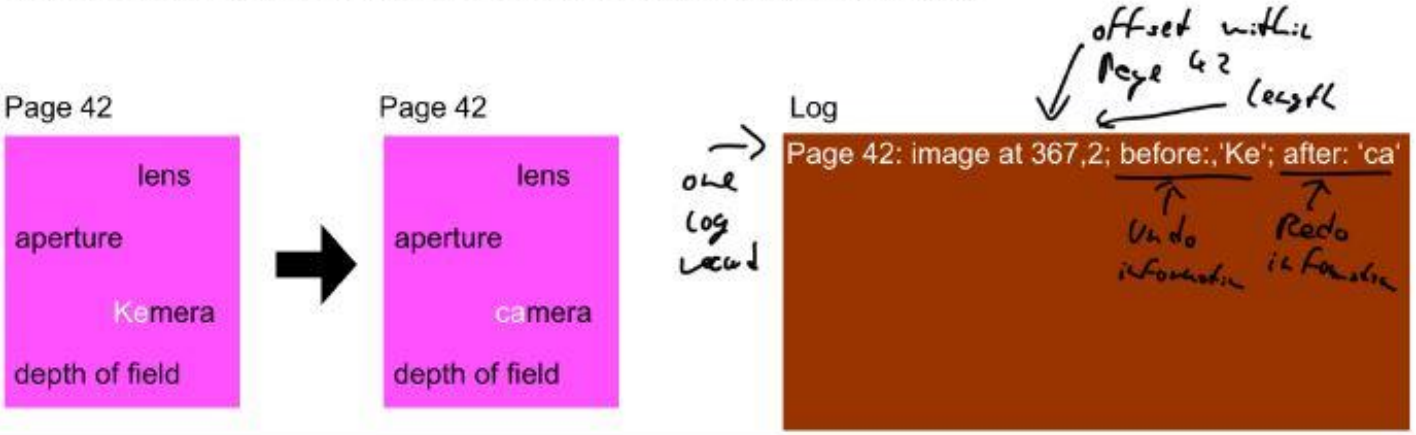
참고자료

Physical Logging

states (byte images) are logged

before image contains state **before** change was performed

after image contains state **after** change was performed



만약 Kemera를 가지고 있는 recor들이 많다면
관련 log recor도 많이 생길것이다.
Redo는 빨리 이루어 진다.

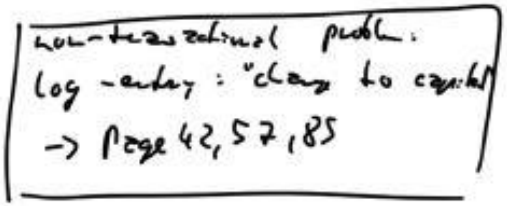
Physiological Redo Logging: 특정 Attribute값을 update했을때

참고자료

Logical Logging

high-level operations are logged

not necessarily limited to a single page



CameraLingo	
termID	term
0	Kemera
1	lens
2	aperture
3	depth of field
4	aperture
5	body
~~6~~	shutter

Page 42

Page 42

Log

만약 Kemera를 가지고 있는 record들이 많다해도
log record는 1개만 있으면 된다.
그러나 redo과정이 시간이 많이 걸린다.

Physiological Redo Logging: 특정 Attribute값을 update했을때

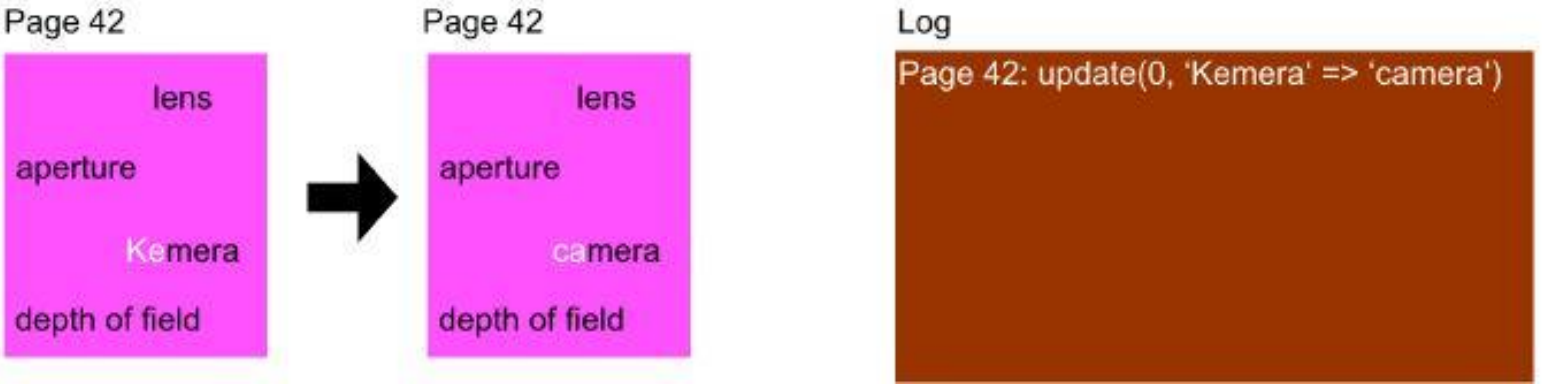
참고자료

Physiological Logging

like logical logging, but:

log entry **may only affect a single page**

example for capital character
=> 3 log records



만약 Kamera를 가지고 있는 record들이 많다해도
physical redo log보다 simple한 physiological log record 를 기록한다.
그러나 redo과정이 시간이 logical redo보다 빠르게 진행된다 생길것이다.

ARIES Data Structures

- 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
      - 로그화일번호, 로그화일내 오프셋
  - Page LSN (특정 page에 마지막 update내용을 log에 기록한 log record의 LSN)
  - Log records of several different types
  - Dirty page table (page ID, pageLSN, recLSN)

ARIES Data Structures: Page LSN

- 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

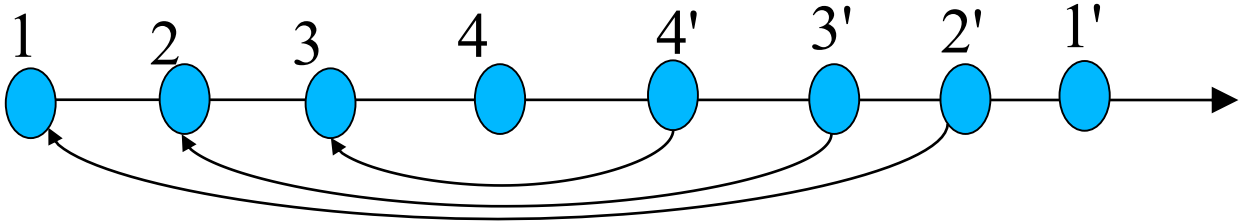
ARIES Data Structures: Log Record

- 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 earlier 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

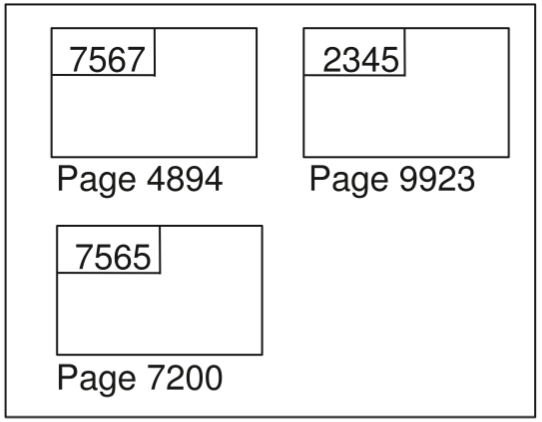
- **DirtyPageTable (아직 disk에 반영이 안된 page의 집합이란 의미)**
  - List of pages in the buffer that have been updated
  - Contains, for each such page
    - **PageID**
    - **PageLSN** of the page
    - **RecLSN** is an LSN such that log records before this LSN have already been applied to the page version on 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
  - 어떤 page가 output to disk이 되면 dirty page table에서 삭제
  - RecLSN 필드의 의미는 해당페이지가 disk에서 memory로 들어올때의 LSN

PageID	PageLSN	RecLSN
4894	7567	7564
7200	7565	7565

Dirty Page Table

참고자료

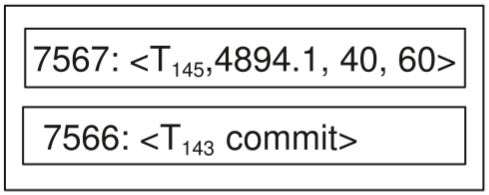
ARIES Data Structures



Database Buffer

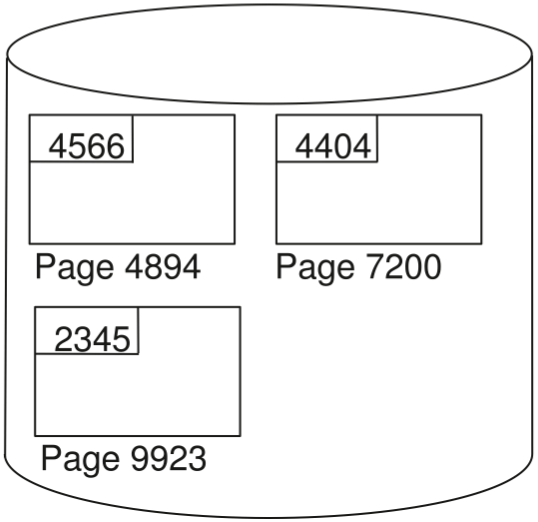
PageID	PageLSN	RecLSN
4894	7567	7564
7200	7565	7565

Dirty Page Table



Log Buffer (PrevLSN and UndoNextLSN fields not shown)

Stable data



Stable log

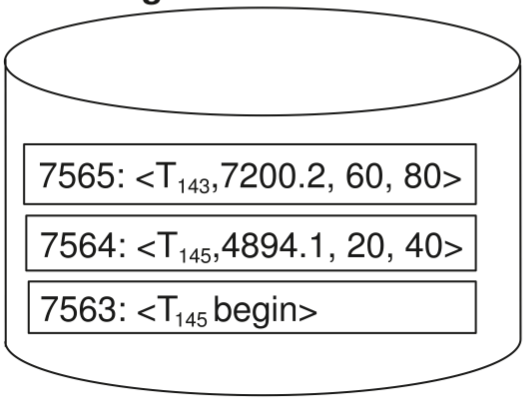


Fig 16.08

ARIES Data Structures: Checkpoint Log

- **Checkpoint log record**
  - Contains:
    - DirtyPageTable and a 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 the 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

참고자료

7568: checkpoint

Txn	lastLSN
T145	7567

PageID	PageLSN	RecLSN
4894	7567	7564
7200	7565	7565

Maintaining ARIES Data Structures [1/13]

RedoInfo UndoInfo <LSN TransId PrevLSN>

Active Transaction List

Trans. ID	LastLSN

Dirty Page Table

PageID	PageLSN	RecLSN

Transaction Schedule

1. T1 write page 1
2. T2 write page 2
3. T1 write page 1
4. T3 write page 4

(Page 1 flushed to disk)

5. T2 commits
6. Begin Checkpoint
7. End Checkpoint
8. T4 write page 3

(Page 4 flushed to disk)

9. T3 write page 2
10. T3 commits
11. T1 writes page 4

Crash!

Log Buffer

// LastLSN: 해당 트랜잭션에 의해 마지막으로 수행된 로그의 LSN
// PageLSN : 해당 페이지를 가장 마지막으로 쓴 로그의 LSN
// RecLSN : 해당 페이지가 Dirty page Table로 새로 들어올 시점의 로그 버퍼의 가장 마지막 LSN

Maintaining ARIES Data Structures [2/13]

Active Transaction List

Trans. ID	LastLSN
T1	1

Dirty Page Table

Page #	PageLSN	RecLSN
1	1	0

RedoInfo UndoInfo <LSN TransId PrevLSN>

1. T1 write page 1
2. T2 write page 2
3. T1 write page 1
4. T3 write page 4

(Page 1 flushed to disk)

5. T2 commits
6. Begin Checkpoint
7. End Checkpoint
8. T4 write page 3

(Page 4 flushed to disk)

9. T3 write page 2
10. T3 commits
11. T1 writes page 4

Crash!

< 1, T1, - >

Log Buffer

Maintaining ARIES Data Structures [3/13]

Active Transaction List

Trans. ID	LastLSN
T1	1
T2	**2**

Dirty Page Table

Page #	PageLSN	RecLSN
1	1	0
2	**2**	**1**

RedoInfo UndoInfo <LSN TransId PrevLSN>

1. T1 write page 1
2. **T2 write page 2**
3. T1 write page 1
4. T3 write page 4

(Page 1 flushed to disk)

5. T2 commits
6. Begin Checkpoint
7. End Checkpoint
8. T4 write page 3

(Page 4 flushed to disk)

9. T3 write page 2
10. T3 commits
11. T1 writes page 4

Crash!

< 1, T1, - >
< 2, T2, - >

Log Buffer

Maintaining ARIES Data Structures [4/13]

Active Transaction List

Trans. ID	LastLSN
T1	**3**
T2	2

Dirty Page Table

Page #	PageLSN	RecLSN
1	**3**	0
2	2	1

RedoInfo UndoInfo <LSN TransId PrevLSN>

1. T1 write page 1
2. T2 write page 2
3. **T1 write page 1**
4. T3 write page 4

(Page 1 flushed to disk)

5. T2 commits
6. Begin Checkpoint
7. End Checkpoint
8. T4 write page 3

(Page 4 flushed to disk)

9. T3 write page 2
10. T3 commits
11. T1 writes page 4

Crash!

< 1, T1, - >
< 2, T2, - >
< 3, T1, 1 >

Maintaining ARIES Data Structures [5/13]

Active Transaction List

Trans. ID	LastLSN
T1	3
T2	2
T3	**4**

Dirty Page Table

Page #	PageLSN	RecLSN
1	3	0
2	2	1
4	**4**	**3**

RedoInfo UndoInfo <LSN TransId PrevLSN>

1. T1 write page 1
2. T2 write page 2
3. T1 write page 1
4. **T3 write page 4**

(Page 1 flushed to disk)

5. T2 commits
6. Begin Checkpoint
7. End Checkpoint
8. T4 write page 3

(Page 4 flushed to disk)

9. T3 write page 2
10. T3 commits
11. T1 writes page 4

Crash!

< 1, T1, - >
< 2, T2, - >
< 3, T1, 1 >
< 4, T3, - >

Maintaining ARIES Data Structures [6/13]

RedoInfo UndoInfo <LSN TransId PrevLSN>

Active Transaction List

Trans. ID	LastLSN
T1	3
T2	2
T3	4

Dirty Page Table

Page #	PageLSN	RecLSN
~~1~~	~~3~~	~~0~~
2	2	1
4	4	3

1. T1 write page 1
2. T2 write page 2
3. T1 write page 1
4. T3 write page 4

(Page 1 flushed to disk)

5. T2 commits
6. Begin Checkpoint
7. End Checkpoint
8. T4 write page 3

(Page 4 flushed to disk)

9. T3 write page 2
10. T3 commits
11. T1 writes page 4

Crash!

< 1, T1, - >
< 2, T2, - >
< 3, T1, 1 >
< 4, T3, - >

Maintaining ARIES Data Structures [7/13]

Active Transaction List

Trans. ID	LastLSN
T1	3
~~T2~~	~~2~~
T3	4

Dirty Page Table

Page #	PageLSN	RecLSN
2	2	1
4	4	3

RedoInfo UndoInfo <LSN TransId PrevLSN>

1. T1 write page 1
2. T2 write page 2
3. T1 write page 1
4. T3 write page 4

(Page 1 flushed to disk)

5. **T2 commits**
6. Begin Checkpoint
7. End Checkpoint
8. T4 write page 3

(Page 4 flushed to disk)

9. T3 write page 2
10. T3 commits
11. T1 writes page 4

Crash!

< 1, T1, - >
< 2, T2, - >
< 3, T1, 1 >
< 4, T3, - >

< 5, T2 commit >

Maintaining ARIES Data Structures [8/13]

Active Transaction List

Trans. ID	LastLSN
T1	3
T3	4

Dirty Page Table

Page #	PageLSN	RecLSN
2	2	1
4	4	3

RedoInfo UndoInfo <LSN TransId PrevLSN>

1. T1 write page 1
2. T2 write page 2
3. T1 write page 1
4. T3 write page 4

(Page 1 flushed to disk)

5. T2 commits
6. **Begin Checkpoint**
7. **End Checkpoint**
8. T4 write page 3

(Page 4 flushed to disk)

9. T3 write page 2
10. T3 commits
11. T1 writes page 4

Crash!

< 1, T1, - >
< 2, T2, - >
< 3, T1, 1 >
< 4, T3, - >

< 5, T2 commit >
< begin chkpt >
< end chkpt >

* Save this shot at CheckPoint

Maintaining ARIES Data Structures [9/13]

Active Transaction List

Trans. ID	LastLSN
T1	3
T3	4
T4	**8**

Dirty Page Table

Page #	PageLSN	RecLSN
2	2	1
4	4	3
3	**8**	**7**

RedoInfo UndoInfo <LSN TransId PrevLSN>

1. T1 write page 1
2. T2 write page 2
3. T1 write page 1
4. T3 write page 4

(Page 1 flushed to disk)

5. T2 commits
6. Begin Checkpoint
7. End Checkpoint
8. **T4 write page 3**

(Page 4 flushed to disk)

9. T3 write page 2
10. T3 commits
11. T1 writes page 4

Crash!

< 1, T1, - >
< 2, T2, - >
< 3, T1, 1 >
< 4, T3, - >

< 5, T2 commit >
< begin chkpt >
< end chkpt >
< 8, T4, - >

Maintaining ARIES Data Structures [10/13]

Active Transaction List

Trans. ID	LastLSN
T1	3
T3	4
T4	8

Dirty Page Table

Page #	PageLSN	RecLSN
2	2	1
~~4~~	~~4~~	~~3~~
3	8	7

RedoInfo UndoInfo <LSN TransId PrevLSN>

1. T1 write page 1
2. T2 write page 2
3. T1 write page 1
4. T3 write page 4

(Page 1 flushed to disk)

5. T2 commits
6. Begin Checkpoint
7. End Checkpoint
8. T4 write page 3

(Page 4 flushed to disk)

9. T3 write page 2
10. T3 commits
11. T1 writes page 4

Crash!

< 1, T1, - >
< 2, T2, - >
< 3, T1, 1 >
< 4, T3, - >

< 5, T2 commit >
< begin chkpt >
< end chkpt >
< 8, T4, - >

ARIES Running Example [11/13]

Active Transaction List

Trans. ID	LastLSN
T1	3
T3	**9**
T4	8

Dirty Page Table

Page #	PageLSN	RecLSN
2	**9**	1
3	8	7

RedoInfo UndoInfo <LSN TransId PrevLSN>

1. T1 write page 1
2. T2 write page 2
3. T1 write page 1
4. T3 write page 4

(Page 1 flushed to disk)

5. T2 commits
6. Begin Checkpoint
7. End Checkpoint
8. T4 write page 3

(Page 4 flushed to disk)

9. **T3 write page 2**
10. T3 commits
11. T1 writes page 4

Crash!

< 1, T1, - >
< 2, T2, - >
< 3, T1, 1 >
< 4, T3, - >

< 5, T2 commit >
< begin chkpt >
< end chkpt >
< 8, T4, - >

< 9, T3, 4 >

Maintaining ARIES Data Structures [12/13]

Active Transaction List

Trans. ID	LastLSN
T1	3
~~T3~~	~~9~~
T4	8

Dirty Page Table

Page #	PageLSN	RecLSN
2	9	1
3	8	7

RedoInfo UndoInfo <LSN TransId PrevLSN>

1. T1 write page 1 — **< 1, T1, - >**
2. T2 write page 2 — **< 2, T2, - >**
3. T1 write page 1 — **< 3, T1, 1 >**
4. T3 write page 4 — **< 4, T3, - >**

(Page 1 flushed to disk)

5. T2 commits — **< 5, T2 commit >**
6. Begin Checkpoint — **< begin chkpt >**
7. End Checkpoint — **< end chkpt >**
8. T4 write page 3 — **< 8, T4, - >**

(Page 4 flushed to disk)

9. T3 write page 2 — **< 9, T3, 4 >**
10. **T3 commits** — **< 10, T3 commit >**
11. T1 writes page 4

Crash!

Maintaining ARIES Data Structures [13/13]

Active Transaction List

Trans. ID	LastLSN
T1	**11**
T4	8

Dirty Page Table

Page #	PageLSN	RecLSN
4	**11**	**10**
2	9	1
3	8	7

RedoInfo UndoInfo <LSN TransId PrevLSN>

1. T1 write page 1 — **< 1, T1, - >**
2. T2 write page 2 — **< 2, T2, - >**
3. T1 write page 1 — **< 3, T1, 1 >**
4. T3 write page 4 — **< 4, T3, - >**

(Page 1 flushed to disk)

5. T2 commits — **< 5, T2 commit >**
6. Begin Checkpoint — **< begin chkpt >**
7. End Checkpoint — **< end chkpt >**
8. T4 write page 3 — **< 8, T4, - >**

(Page 4 flushed to disk)

9. T3 write page 2 — **< 9, T3, 4 >**
10. T3 commits — **< 10, T3 commit >**
11. **T1 writes page 4** — **< 11, T1, 3 >**

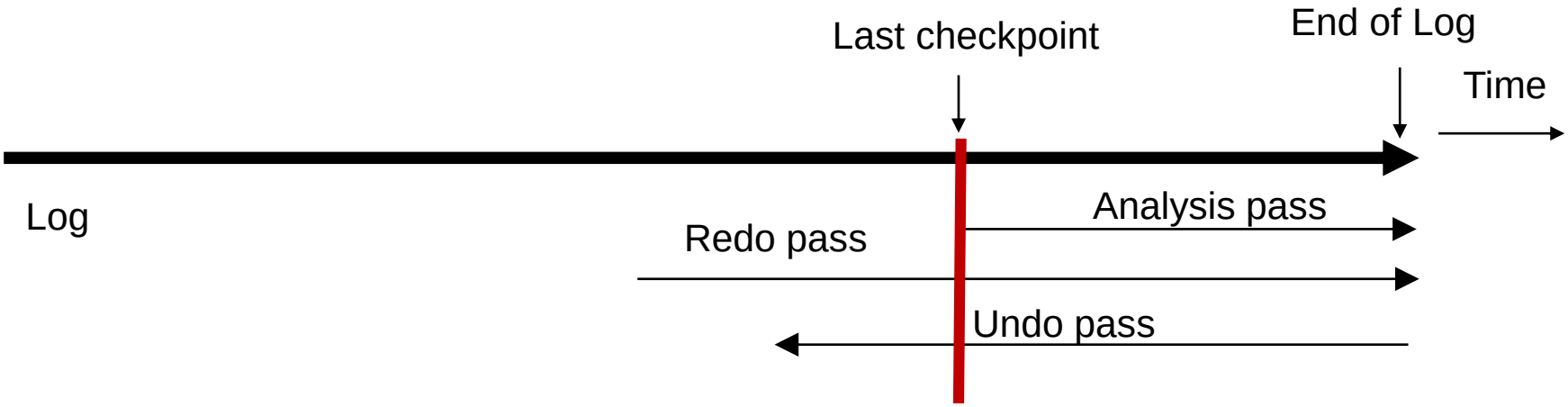
Crash!

3 Passes of ARIES Recovery Algorithm

- 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 Structure

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



Fuzzy CheckPointing에서는 checkpoint작업중에 stop했던
transaction들이 resume하는 시간이 빨라 졌기때문에
Redo나 Undo의 scope가 checkpoint이전까지 넘어가야 한다!

Recovery Actions in ARIES

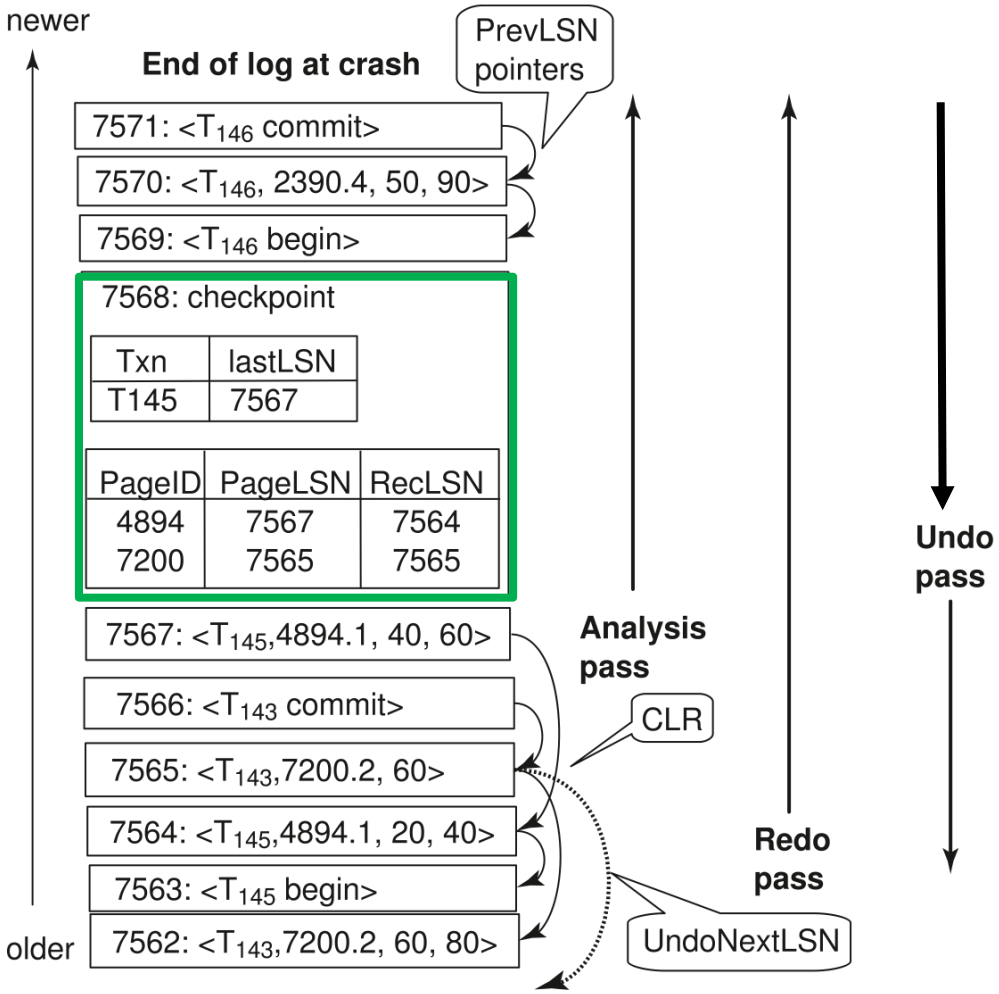


Fig 16.09

ARIES Recovery: Analysis [1/2]

Analysis pass

- Starts from the 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 (… on next page …)

ARIES Recovery: Analysis [2/2]

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 the corresponding page is not in DirtyPageTable, it is added with RecLSN set to LSN of the update log record
  - If “transaction end” log record is 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 Recovery – Analysis Pass [1/5]

- Checkpoint마다 아래의 table을 저장해놓고 있다

RedoInfo UndoInfo <LSN TransId PrevLSN>

Active Transaction List
from the last checkpoint

Trans. ID	LastLSN
T1	3
T3	4

Dirty Page Table
from the last checkpoint

Page #	PageLSN	RecLSN
2	2	**1**
4	4	**3**

Log at crash

1.	T1 write page 1	**< 1, T1, - >**
2.	T2 write page 2	**< 2, T2, - >**
3.	T1 write page 1	**< 3, T1, 1 >**
4.	T3 write page 4	**< 4, T3, - >**
5.	T2 commits	**< 5, T2 commit >**
6.	Begin Checkpoint	**< begin chkpt >**
7.	End Checkpoint	**< end chkpt >**
8.	T4 write page 3	**< 8, T4, - >**
9.	T3 write page 2	**< 9, T3, 4 >**
10.	T3 commits	**< 10, T3 commit >**
11.	T1 writes page 4	**< 11, T1, 3 >**
	crash	

- RedoLSN = min of RecLSN = min(1, 3) = 1
- Undo-list = {T1, T3}

ARIES Recovery – Analysis Pass [2/5]

- Analysis Pass를 진행하면서 적절히 table을 update한다

RedoInfo UndoInfo <LSN TransId PrevLSN>

Active Transaction List from the last checkpoint

Trans. ID	LastLSN
T1	3
T4	**8**
T3	4

Dirty Page Table from the last checkpoint

Page #	PageLSN	RecLSN
3	**8**	**8**
2	2	1
4	4	3

Log at crash

1.	T1 write page 1	**< 1, T1, - >**
2.	T2 write page 2	**< 2, T2, - >**
3.	T1 write page 1	**< 3, T1, 1 >**
4.	T3 write page 4	**< 4, T3, - >**
5.	T2 commits	**< 5, T2 commit >**
6.	Begin Checkpoint	**< begin chkpt >**
7.	End Checkpoint	**< end chkpt >**
8.	**T4 write page 3**	**< 8, T4, - >**
9.	T3 write page 2	**< 9, T3, 4 >**
10.	T3 commits	**< 10, T3 commit >**
11.	T1 writes page 4	**< 11, T1, 3 >**

crash

- RedoLSN = min(1, 3, 8) = 1
- Undo-list = {T1, T3, T4}

ARIES Recovery – Analysis Pass [3/5]

- Analysis Pass를 진행하면서 적절히 table을 update한다

RedoInfo UndoInfo <LSN TransId PrevLSN>

Active Transaction List from the last checkpoint

Trans. ID	LastLSN
T1	3
T4	8
T3	**4 → 9**

Dirty Page Table from the last checkpoint

Page #	PageLSN	RecLSN
3	8	8
2	2	1
4	4	3

Log at crash

1.	T1 write page 1	**< 1, T1, - >**
2.	T2 write page 2	**< 2, T2, - >**
3.	T1 write page 1	**< 3, T1, 1 >**
4.	T3 write page 4	**< 4, T3, - >**
5.	T2 commits	**< 5, T2 commit >**
6.	Begin Checkpoint	**< begin chkpt >**
7.	End Checkpoint	**< end chkpt >**
8.	T4 write page 3	**< 8, T4, - >**
9.	**T3 write page 2**	**< 9, T3, 4 >**
10.	T3 commits	**< 10, T3 commit >**
11.	T1 writes page 4	**< 11, T1, 3 >**
crash		

- RedoLSN = min(1, 3, 8) = 1
- Undo-list = {T1, T3, T4}

ARIES Recovery – Analysis Pass [4/5]

- Analysis Pass를 진행하면서 적절히 table을 update한다

RedoInfo UndoInfo <LSN TransId PrevLSN>

Active Transaction List
from the last checkpoint

Trans. ID	LastLSN
T1	3
T4	8
~~T3~~	~~9~~

Dirty Page Table
from the last checkpoint

Page #	PageLSN	RecLSN
3	8	8
2	2	1
4	4	3

Log at crash

1.	T1 write page 1	**< 1, T1, - >**
2.	T2 write page 2	**< 2, T2, - >**
3.	T1 write page 1	**< 3, T1, 1 >**
4.	T3 write page 4	**< 4, T3, - >**
5.	T2 commits	**< 5, T2 commit >**
6.	Begin Checkpoint	**< begin chkpt >**
7.	End Checkpoint	**< end chkpt >**
8.	T4 write page 3	**< 8, T4, - >**
9.	T3 write page 2	**< 9, T3, 4 >**
10.	**T3 commits**	**< 10, T3 commit >**
11.	T1 writes page 4	**< 11, T1, 3 >**
crash		

- RedoLSN = min(1, 3, 8) = 1
- Undo-list = {T1, ~~T3~~, T4}

ARIES Recovery – Analysis Pass [5/5]

- Analysis Pass를 진행하면서 적절히 table을 update한다

RedoInfo UndoInfo <LSN TransId PrevLSN>

Active Transaction List from the last checkpoint

Trans. ID	LastLSN
T1	**3 → 11**
T4	8

Dirty Page Table from the last checkpoint

Page #	PageLSN	RecLSN
3	8	8
2	2	1
4	4	3

Log at crash

1.	T1 write page 1	**< 1, T1, - >**
2.	T2 write page 2	**< 2, T2, - >**
3.	T1 write page 1	**< 3, T1, 1 >**
4.	T3 write page 4	**< 4, T3, - >**
5.	T2 commits	**< 5, T2 commit >**
6.	Begin Checkpoint	**< begin chkpt >**
7.	End Checkpoint	**< end chkpt >**
8.	T4 write page 3	**< 8, T4, - >**
9.	T3 write page 2	**< 9, T3, 4 >**
10.	T3 commits	**< 10, T3 commit >**
11.	**T1 writes page 4**	**< 11, T1, 3 >**
	crash	

- RedoLSN = min(1, 3, 8) = 1
- Undo-list = {T1, T4}

ARIES Redo Pass

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

     otherwise fetch the page from disk

  2. If the PageLSN of the page fetched from disk is less than the LSN of the log record, then 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 Recovery – Redo Pass [1/4]

- Analysis Pass상태에서 관련 able을 그대로 받아서

■ RedoLSN = min(1, 3, 8) = 1

Active Transaction List
from the last checkpoint

Trans. ID	LastLSN
T1	11
T4	8

Dirty Page Table
from the last checkpoint

Page #	PageLSN	RecLSN
3	8	8
2	2	1
4	4	3

Log at crash

1.	**T1 write page 1**	**< 1, T1, - >**
2.	T2 write page 2	**< 2, T2, - >**
3.	T1 write page 1	**< 3, T1, 1 >**
4.	T3 write page 4	**< 4, T3, - >**
5.	T2 commits	**< 5, T2 commit >**
6.	Begin Checkpoint	**< begin chkpt >**
7.	End Checkpoint	**< end chkpt >**
8.	T4 write page 3	**< 8, T4, - >**
9.	T3 write page 2	**< 9, T3, 4 >**
10.	T3 commits	**< 10, T3 commit >**
11.	T1 writes page 4	**< 11, T1, 3 >**

crash

1 : No redo
page 1 is not in dirty page table

ARIES Recovery – Redo Pass [2/4]

- Analysis Pass상태에서
  그대로 관련 able을 받아서

■RedoLSN = min(1, 3, 8) = 1

Active Transaction List
from the last checkpoint

Trans. ID	LastLSN
T1	11
T4	8

Dirty Page Table
from the last checkpoint

Page #	PageLSN	RecLSN
3	8	8
2	2	1
4	4	3

Log at crash

1.	T1 write page 1	**< 1, T1, - >**
2.	**T2 write page 2**	**< 2, T2, - >**
3.	T1 write page 1	**< 3, T1, 1 >**
4.	T3 write page 4	**< 4, T3, - >**
5.	T2 commits	**< 5, T2 commit >**
6.	Begin Checkpoint	**< begin chkpt >**
7.	End Checkpoint	**< end chkpt >**
8.	T4 write page 3	**< 8, T4, - >**
9.	T3 write page 2	**< 9, T3, 4 >**
10.	T3 commits	**< 10, T3 commit >**
11.	T1 writes page 4	**< 11, T1, 3 >**
	crash	

2 : current LSN “2” >= RecLSN of Page2 “1” → read page 2
PageLSN of the fetched page “0” < current LSN “2” → redo

PageLSN of the page fetched from disk (not dirty page table)

ARIES Recovery – Redo Pass [3/4]

- Analysis Pass상태에서
  그대로 관련 able을 받아서

■RedoLSN = min(1, 3, 8) = 1

Active Transaction List
from the last checkpoint

Trans. ID	LastLSN
T1	11
T4	8

Dirty Page Table
from the last checkpoint

Page #	PageLSN	RecLSN
3	8	8
2	2	1
4	4	3

Log at crash	
1. T1 write page 1	**< 1, T1, - >**
2. T2 write page 2	**< 2, T2, - >**
3. T1 write page 1	**< 3, T1, 1 >**
4. T3 write page 4	**< 4, T3, - >**
5. T2 commits	**< 5, T2 commit >**
6. Begin Checkpoint	**< begin chkpt >**
7. End Checkpoint	**< end chkpt >**
8. T4 write page 3	**< 8, T4, - >**
9. T3 write page 2	**< 9, T3, 4 >**
10. T3 commits	**< 10, T3 commit >**
11. T1 writes page 4	**< 11, T1, 3 >**
crash	

3 : No redo

4 : LSN 4 >= RecLSN 3 → read page 4
PageLSN of the fetched page 4 >= 4,
thus no redo

ARIES Recovery – Redo Pass [4/4]

- Analysis Pass상태에서
  그대로 관련 able을 받아서

Active Transaction List
from the last checkpoint

Trans. ID	LastLSN
T1	11
T4	8

Dirty Page Table
from the last checkpoint

Page #	PageLSN	RecLSN
3	8	8
2	2	1
4	4	3

■RedoLSN = min(1, 3, 8) = 1

Log at crash

1.	T1 write page 1	**< 1, T1, - >**
2.	T2 write page 2	**< 2, T2, - >**
3.	T1 write page 1	**< 3, T1, 1 >**
4.	T3 write page 4	**< 4, T3, - >**
5.	T2 commits	**< 5, T2 commit >**
6.	Begin Checkpoint	**< begin chkpt >**
7.	End Checkpoint	**< end chkpt >**
8.	**T4 write page 3**	**< 8, T4, - >**
9.	**T3 write page 2**	**< 9, T3, 4 >**
10.	T3 commits	**< 10, T3 commit >**
11.	**T1 writes page 4**	**< 11, T1, 3 >**

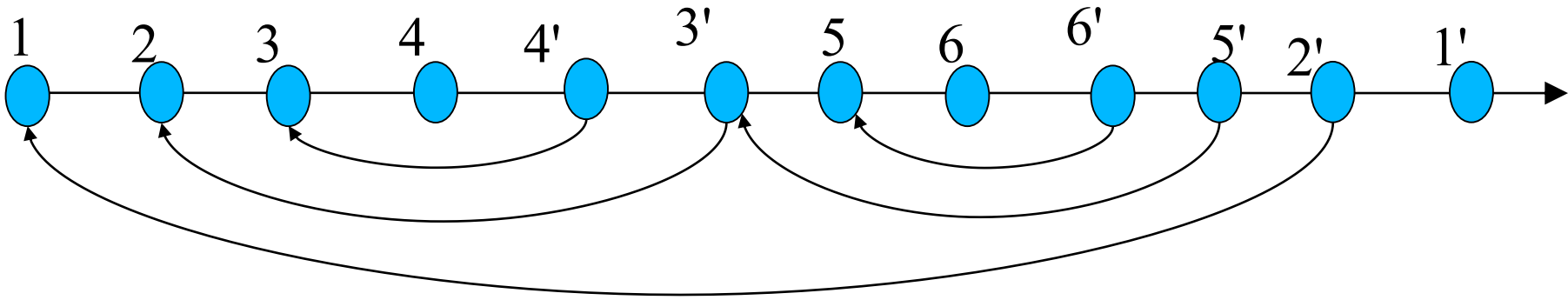
crash

8, 9, 11 : Redo

ARIES Undo Actions

- 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
  - To handle deadlocks by rolling back just enough to release requested locks
  - Figure indicates forward actions after partial rollbacks
    - records 3 and 4 initially, later 5 and 6, then full rollback



ARIES: Undo Pass

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

ARIES Recovery – Undo Pass [1/4]

Active Transaction List

Trans. ID	LastLSN
T1	11
T4	8

- Undo-list = {T1, T4} from Analysis pass

Next record to undo = max(11, 8) = 11

Last LSN of T1
= prevLSN of log record 11 = 3

LSN TransID UndoNextLSN RedoInfo

Log at crash

1.	T1 write page 1	**< 1, T1, - >**
2.	T2 write page 2	**< 2, T2, - >**
3.	T1 write page 1	**< 3, T1, 1 >**
4.	T3 write page 4	**< 4, T3, - >**
5.	T2 commits	**< 5, T2 commit >**
6.	Begin Checkpoint	**< begin chkpt >**
7.	End Checkpoint	**< end chkpt >**
8.	T4 write page 3	**< 8, T4, - >**
9.	T3 write page 2	**< 9, T3, 4 >**
10.	T3 commits	**< 10, T3 commit >**
11.	**T1 writes page 4**	**< 11, T1, 3 >**
12.	**CLR**	**< 11′, T1, 3 >**

ARIES Recovery – Undo Pass [2/4]

LSN TransID UndoNextLSN RedoInfo

Active Transaction List

Trans. ID	LastLSN
T1	**3**
T4	8

- Undo-list = {T1, T4}

Next record to undo
= max(3, 8) = 8

Last LSN T4
= prevLSN of record 8 = "-"
→ Remove T4 from undo-list

Log at crash

1.	T1 write page 1	**< 1, T1, - >**
2.	T2 write page 2	**< 2, T2, - >**
3.	T1 write page 1	**< 3, T1, 1 >**
4.	T3 write page 4	**< 4, T3, - >**
5.	T2 commits	**< 5, T2 commit >**
6.	Begin Checkpoint	**< begin chkpt >**
7.	End Checkpoint	**< end chkpt >**
8.	**T4 write page 3**	**< 8, T4, - >**
9.	T3 write page 2	**< 9, T3, 4 >**
10.	T3 commits	**< 10, T3 commit >**
11.	T1 writes page 4	**< 11, T1, 3 >**
12.	CLR	**< 11′, T1, 3 >**
13.	**CLR**	**< 8′ T4, - >**

ARIES Recovery – Undo Pass [3/4]

Active Transaction List

Trans. ID	LastLSN
T1	3

■Undo-list = {T1}

T4는 active transaction list에서 제거됨

Next record to undo = 3

Last LSN T1
= prevLSN of record 3 = 1

LSN TransID UndoNextLSN RedoInfo

Log at crash

1.	T1 write page 1	< 1, T1, - >
2.	T2 write page 2	< 2, T2, - >
3.	**T1 write page 1**	**< 3, T1, 1 >**
4.	T3 write page 4	< 4, T3, - >
5.	T2 commits	< 5, T2 commit >
6.	Begin Checkpoint	< begin chkpt >
7.	End Checkpoint	< end chkpt >
8.	T4 write page 3	< 8, T4, - >
9.	T3 write page 2	< 9, T3, 4 >
10.	T3 commits	< 10, T3 commit >
11.	T1 writes page 4	< 11, T1, 3 >
12.	CLR	< 11′, T1, 3 >
13.	CLR	< 8′ T4, - >
14.	**CLR**	**< 3′, T1, 1 >**

ARIES Recovery – Undo Pass [4/4]

Active Transaction List

Trans. ID	LastLSN
T1	1

- Undo-list = {T1}

Last LSN T1
= prevLSN of record 1 = "-"
→ Remove T1 from undo-list

Empty undo-list
→ Undo complete

LSN TransID UndoNextLSN RedoInfo

Log at crash

1.	**T1 write page 1**	**< 1, T1, - >**
2.	T2 write page 2	**< 2, T2, - >**
3.	T1 write page 1	**< 3, T1, 1 >**
4.	T3 write page 4	**< 4, T3, - >**
5.	T2 commits	**< 5, T2 commit >**
6.	Begin Checkpoint	**< begin chkpt >**
7.	End Checkpoint	**< end chkpt >**
8.	T4 write page 3	**< 8, T4, - >**
9.	T3 write page 2	**< 9, T3, 4 >**
10.	T3 commits	**< 10, T3 commit >**
11.	T1 writes page 4	**< 11, T1, 3 >**
12.	CLR	**< 11′, T1, 3 >**
13.	CLR	**< 8′ T4, - >**
14.	CLR	**< 3′, T1, 1 >**
15.	**CLR**	**< 1′, T1, - >**

Other ARIES Features [1/2]

- 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

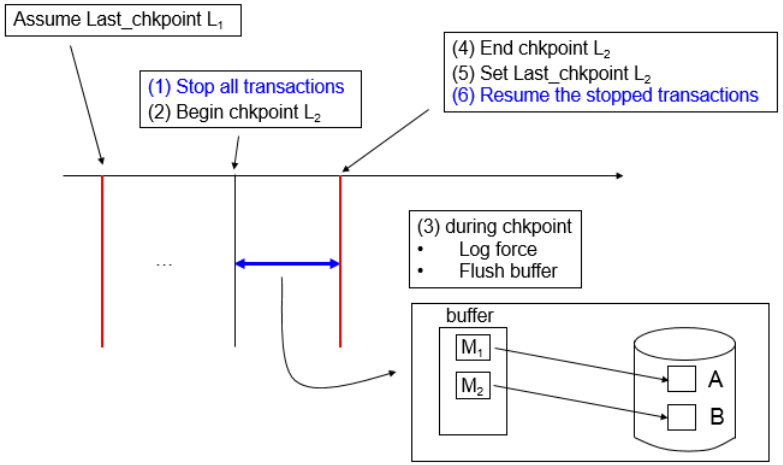
Other ARIES Features [2/2]

- 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 earlier 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

Traditional Recovery vs ARIES [1/3] Checkpoint Viewpoint

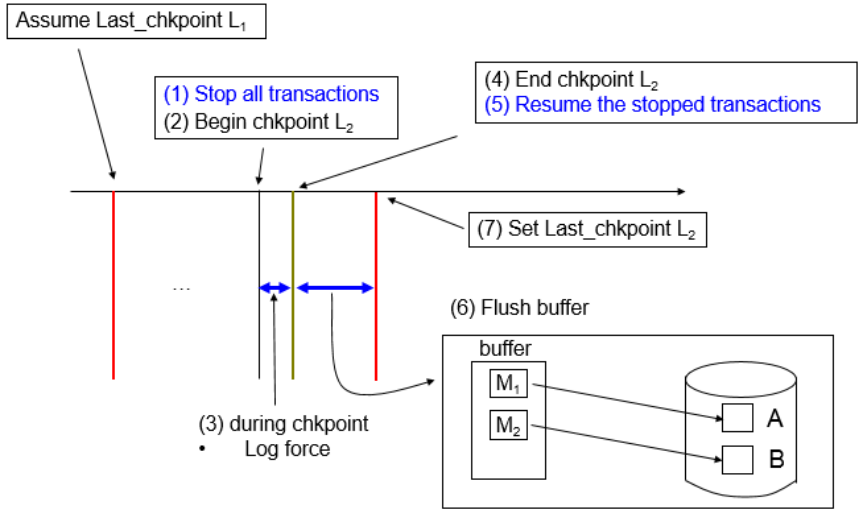
참고자료

Traditional Recovery Checkpointing system



- 체크포인트 수행 시 모든 변경사항을 출력함
- 변경사항이 기록되는 동안 모든 트랜잭션이 중단됨
- 복구과정의 Redo 단계는 checkpoint 이후부터 수행함

ARIES FuzzyCheckpointing system



- 체크포인트 수행 시 모든 변경사항이 출력되지 않음 (성능↑)
- 백그라운드에서 변경사항이 기록되는 동안 진행중인 트랜잭션이 중단되지 않음 (성능↑)
- 복구과정의 Redo 단계는 RedoLSN부터 수행함 (단점 : checkpoint 이전의 로그일 수 있음)

Traditional Recovery vs ARIES [2/3]
Redo Viewpoint

참고자료

■ Redo in Traditional Recovery System

1. T1 write page 1
2. T2 write page 2

(Page 1 flushed to disk)

3. T2 commits
4. Begin Checkpoint
5. End Checkpoint
6. **T3 write page 3**
7. **T3 write page 4**
8. **T3 write page 5**
9. **T3 commit**

(Page 3,4,5 flushed to disk)

10. **T4 write page 3**
11. **T3 write page 4**
12. **T4 commit**

(Page 3,4 flushed to disk)

13. **T5 write page 5**
14. **T5 commit**

Crash!

Checkpoint부터 Crash 이전까지 모두 Redo

■ Redo in ARIES System

1. T1 write page 1
2. **T2 write page 2**

(Page 1 flushed to disk)

3. **T2 commits**
4. Begin Checkpoint
5. End Checkpoint
6. T3 write page 3
7. T3 write page 4
8. T3 write page 5
9. T3 commit

(Page 3,4,5 flushed to disk)

10. T4 write page 3
11. T3 write page 4
12. T4 commit

(Page 3,4 flushed to disk)

13. **T5 write page 5**
14. **T5 commit**

Crash!

불필요한 Redo 생략 (성능↑)

Traditional Recovery vs ARIES [3/3]

참고자료

Undo Viewpoint

■ Undo in Traditional Recovery System

1. **T1 write page 1**
2. T2 write page 2

(Page 1 flushed to disk)

3. T2 commits
4. Begin Checkpoint
5. End Checkpoint
6. T3 write page 3
7. T3 write page 4
8. T3 write page 5
9. T3 commit

(Page 3,4,5 flushed to disk)

10. T4 write page 3
11. T3 write page 4
12. T4 commit

(Page 3,4 flushed to disk)

13. T5 write page 5
14. T5 commit

Crash!

로그를 역순으로 탐색하여 Undo-list 내의 트랜잭션인 경우 Undo를 수행함

■ Undo in ARIES System

1. **T1 write page 1**
2. T2 write page 2

(Page 1 flushed to disk)

3. T2 commits
4. Begin Checkpoint
5. End Checkpoint
6. T3 write page 3
7. T3 write page 4
8. T3 write page 5
9. T3 commit

(Page 3,4,5 flushed to disk)

10. T4 write page 3
11. T3 write page 4
12. T4 commit

(Page 3,4 flushed to disk)

13. T5 write page 5
14. T5 commit

Crash!

불필요한 로그 탐색 생략

Active Transaction List의 LastLSN 필드와 각 로그의 PrevLSN 필드를 이용하여 다음 순서로 Undo 해야 할 로그를 빠르게 찾음 (성능↑)

Undo-list = {T1}

Chapter 16: Recovery System

- Failure Classification
- Storage Structure
- Recovery and Atomicity
- Recovery Algorithms
- Buffer Management
- Failure with Loss of Nonvolatile Storage
- Early Lock Release and Logical Undo Operations
- ARIES
- Remote Backup Systems

Remote Backup Systems [1/4]

- Remote backup systems provide high availability by allowing transaction processing to continue even if the primary site is destroyed
  - 만약 primary site가 완전히 손상이면, backup site가 인계 받아 (1) 복구를 수행하고, (2) 정상적인 operation을 대신 진행하고 (3) 나중에 primary site가 up하면 다시 인계
- Primary site and secondary site should be synchronized
  - by sending all log records from primary to secondary

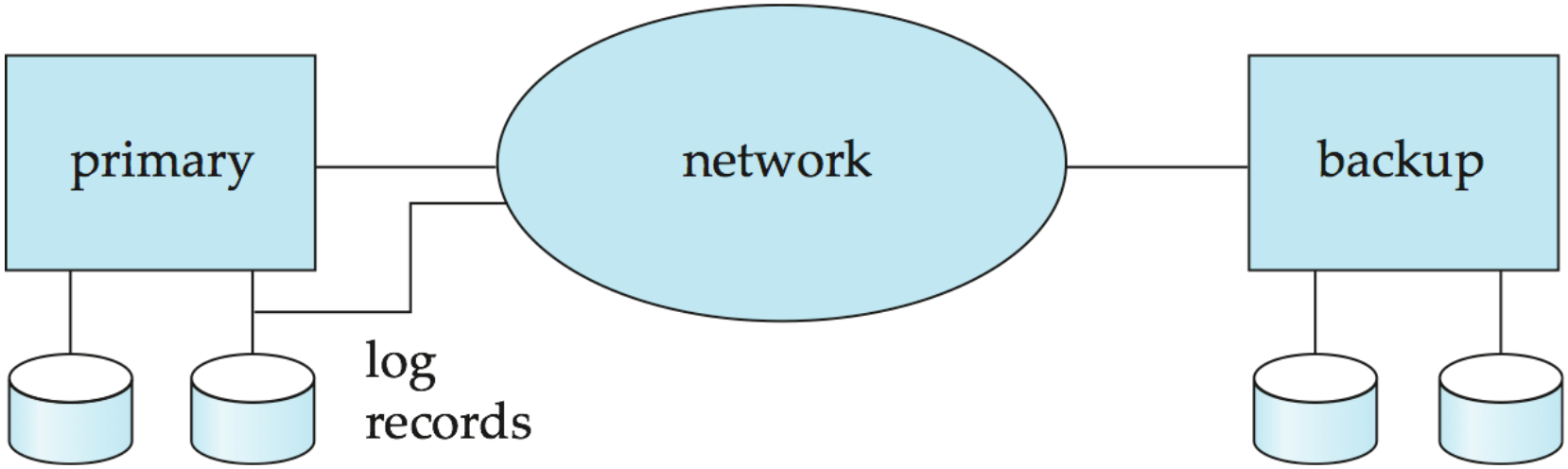


Fig 16.10

Remote Backup Systems [2/4]

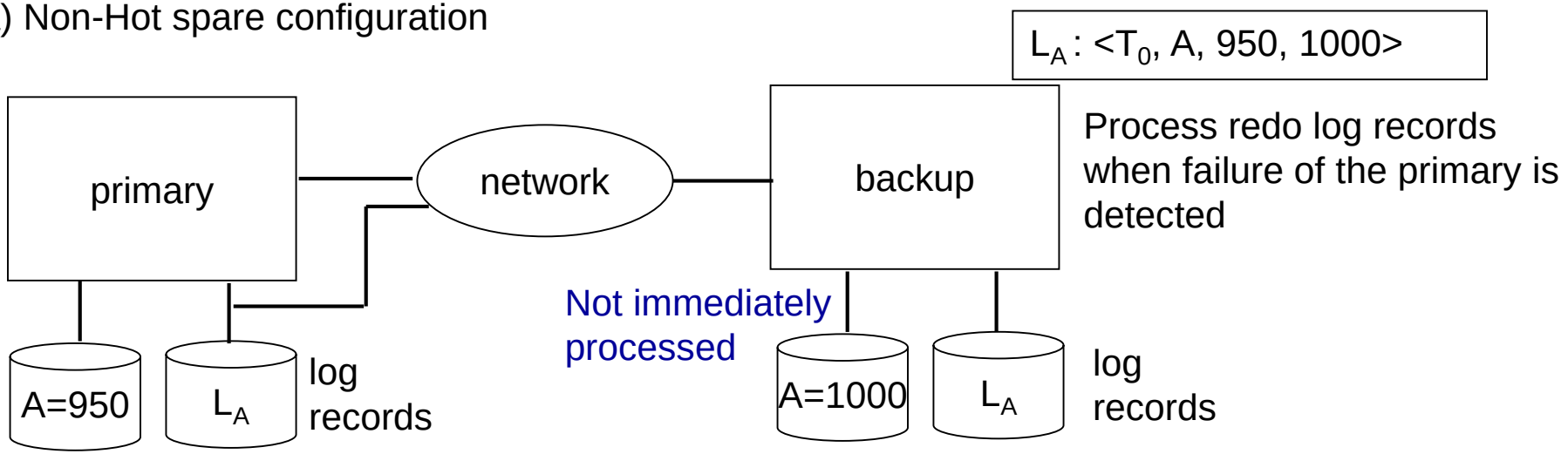
- **Detection of failure**:
  - Backup site must detect when primary site has failed
  - Need to distinguish primary site failure from link failure
    - maintain several communication links between the primary and the remote backup
  - Heart-beat messages
- **Transfer of control**:
  - To take over control the 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
    - DB 전체를 이동하는게 아님!
    - Back-up site에 failure가 발생한것으로 간주하여 Primary site로 넘길수도 있음

Remote Backup Systems [3/4]

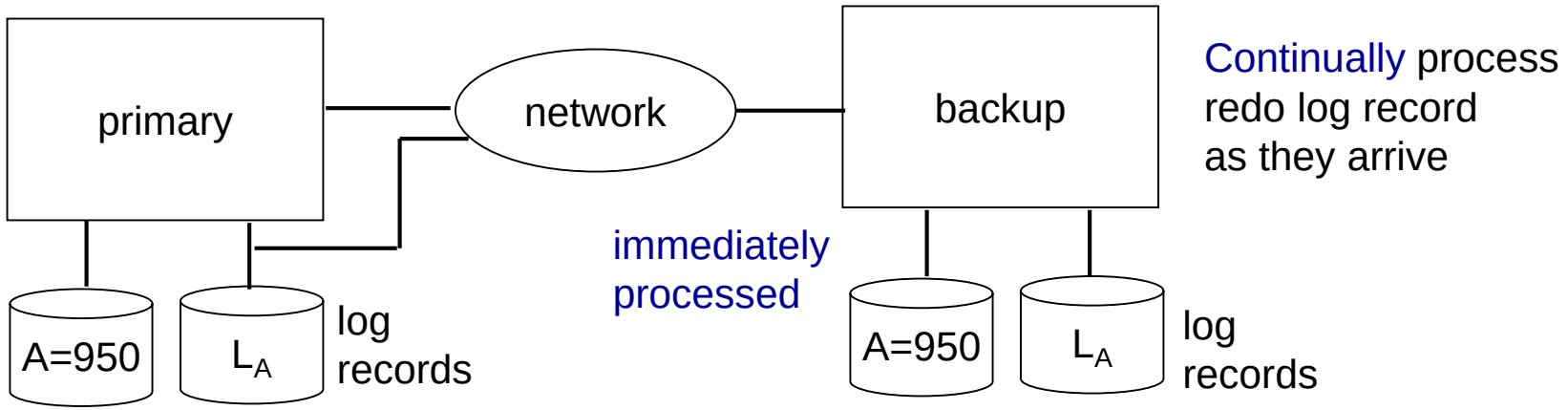
- **Time to recover**
- To reduce time delay in takeover or recover
  - The backup-site periodically processes the redo log records (in effect, performing recovery from previous database state), performs a checkpoint, and can then delete earlier parts of the log
- **Hot-Spare configuration** permits 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-site just rolls back (undo) incomplete transactions, and then, is ready to process new transactions
- Alternative to the remote backup**: distributed database with replicated data**
  - Transactions are required to update all replicas of any data item that they update
    - Transactions are accepted in any database replicas
    - Need expensive protocol like 2 phase commit or 3 phase commit
  - The remote-backup is faster and cheaper, but less tolerant to failure
    - more on this in Chapter 19 (distributed databases)

Hot Spare Configuration Example

(a) Non-Hot spare configuration



(b) Hot spare configuration

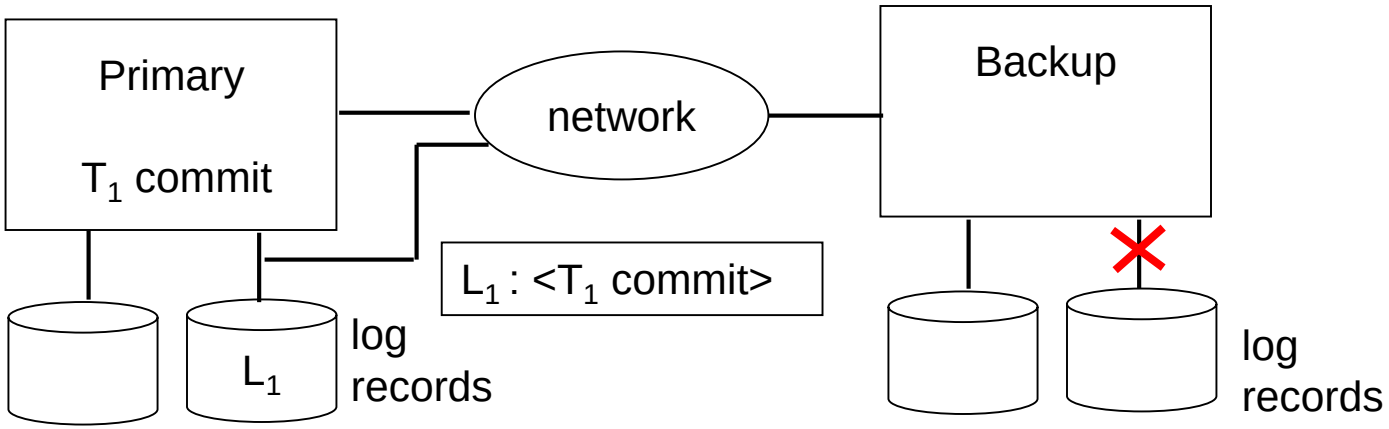


Remote Backup Systems [4/4]

- **Time to commit**
- Durability of updates is guaranteed by delaying transaction commit until update is logged at the backup site (in fact, the two-very-safe scheme below)
  - High durability causes low availability
- **One-safe scheme**
  - Allow commit as soon as transaction's commit log record is written at primary
  - Problem: updates may not arrive at backup before it takes over ➔ lost updates
- **Two-very-safe scheme:**
  - Allow commit when transaction's commit log record is written at both primary and backup
  - Problem: low availability since transactions cannot commit if either site fails
- **Two-safe scheme:** (절충안)
  - 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 the two-very-safe scheme
  - Avoids problem of lost transactions in one-safe
    - Back-up이 active해지면 다시 Two-very-safe scheme으로 돌아가므로

Time to Commit Example

(a) One-safe; T1 can commit even if L1 is not written in the log of the backup



(b) Two-very-safe; T1 cannot commit if L1 is not written in the log of the backup