Chapter 16: Recovery System

Database System Concepts, 6th Ed.

Outline

- Failure Classification
- Storage Structure
- Recovery and Atomicity
- Log-Based Recovery
- Recovery Algorithm
- Recovery with Early Lock Release
- ARIES Recovery Algorithm
- Remote Backup Systems

Failure Classification

- **Transaction failure** :
  - **Logical errors**: transaction cannot complete due to some internal error co
  - **System errors**: the database system must terminate an active transaction
    deadlock)
- **System crash**: a power failure or other hardware or software failure causes th
  - **Fail-stop assumption**: non-volatile storage contents are assumed to not b
    - Database systems have numerous integrity checks to prevent corruptio
- **Disk failure**: a head crash or similar disk failure destroys all or part of disk sto
  - Destruction is assumed to be detectable: disk drives use checksums to de

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 l
  - Modifying the database without ensuring that the transaction will commit
    state
  - Not modifying the database may result in lost updates if failure occurs just
- Recovery algorithms have two parts
  1. Actions taken during normal transaction processing to ensure enough info
  2. Actions taken after a failure to recover the database contents to a state th

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

Stable-Storage Implementation

- Maintain multiple copies of each block on separate disks
  - copies can be at remote sites to protect against disasters such as fire or flooding.
- Failure during data transfer can still result in inconsistent copies.

  Block transfer can result in
  - Successful completion
  - Partial failure: destination block has incorrect information
  - Total failure: destination block was never updated
- Protecting storage media from failure during data transfer (one solution):
  - Execute output operation as follows (assuming two copies of each block):

1. Write the information onto the first physical block.
2. When the first write successfully completes, write the same information onto the
3. The output is completed only after the second write successfully completes.

Stable-Storage Implementation (Cont.)

Protecting storage media from failure during data transfer (cont.):

- Copies of a block may differ due to failure during output operation. To recover from failu
  1. First find inconsistent blocks:
     1. *Expensive solution*: Compare the two copies of every disk block.
     2. *Better solution*:
        - Record in-progress disk writes on non-volatile storage (Non-volatile RAM o
        - Use this information during recovery to find blocks that may be inconsister
        - Used in hardware RAID systems
  2. If either copy of an inconsistent block is detected to have an error (bad checksum),

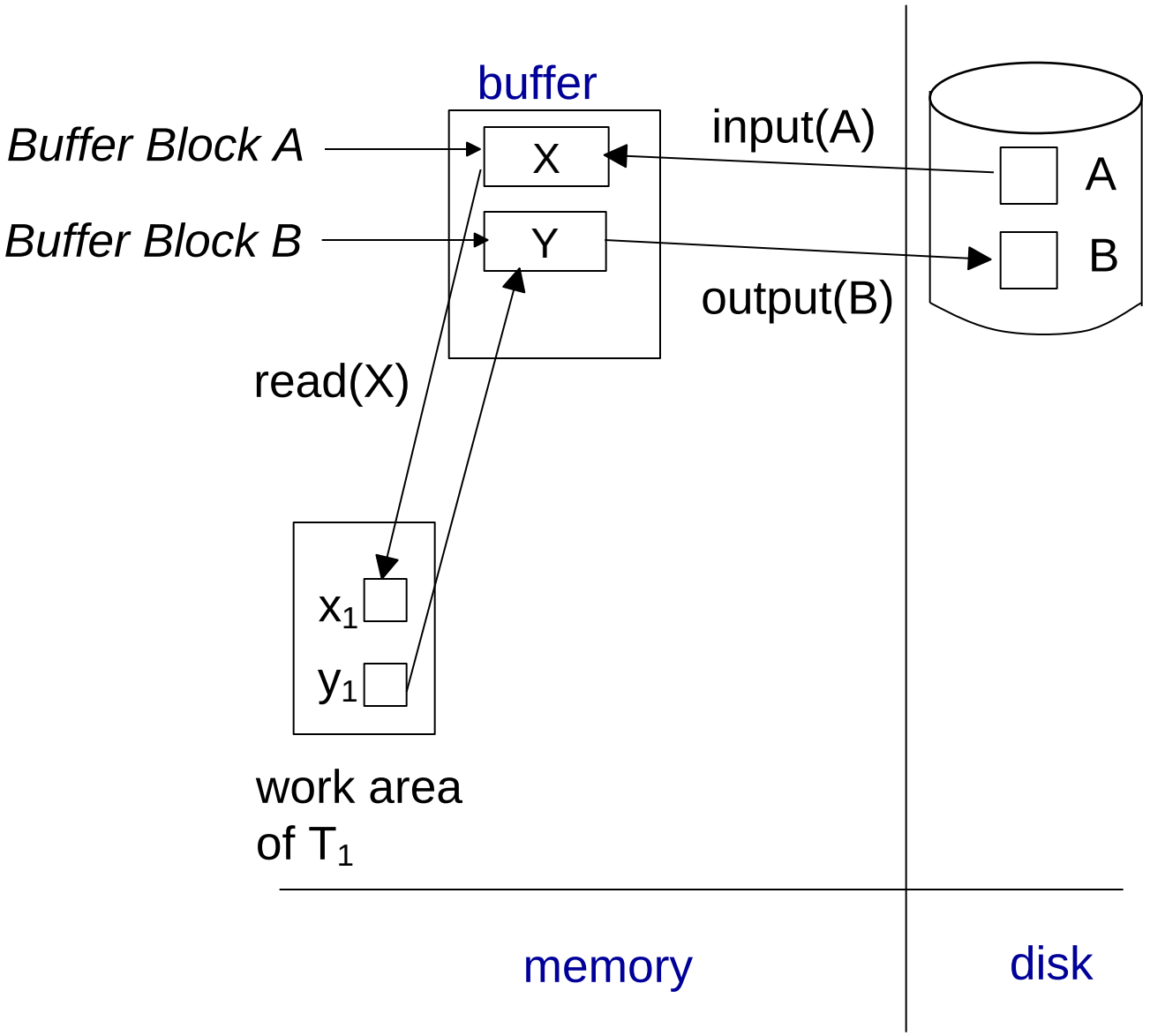
Data Access

- **Physical blocks** are those blocks residing on the disk.
- **System buffer blocks** are the blocks residing temporarily in main memory.
- Block movements between disk and main memory are initiated through the foll
  - **input**(*B*) transfers the physical block *B* to main memory.
  - **output**(*B*) transfers the buffer block *B* to the disk, and replaces the approp
- We assume, for simplicity, that each data item fits in, and is stored inside, a sin

Data Access (Cont.)

- Each transaction T_i has its private work-area in which local copies of all data
  - T_i's local copy of a data item X is denoted by x_i.
  - B_X denotes block containing X
- Transferring data items between system buffer blocks and its private work-a
  - **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 buff
- Transactions
  - Must perform **read**(X) before accessing X for the first time (subsequent
  - The **write**(X) can be executed at any time before the transaction commi
- Note that **output**(B_X) need not immediately follow **write**(X). System can per

Example of Data Access



Recovery and Atomicity

- To ensure atomicity despite failures, we first output information describing t
- We study **log-based recovery mechanisms** in detail
  - We first present key concepts
  - And then (module 17) present the actual recovery algorithm
- Less used alternative: **shadow-paging** (brief details presented in the book)
- In this Module we assume serial execution of transactions. In Module 17, w

Log-Based Recovery

- A **log** is kept on stable storage.
  - The log is a sequence of **log records**, which maintains information abou
- When transaction T_i starts, it registers itself by writing a record
  $<T_i$ **start**>
  to the log
- *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
- When T_i finishes it last statement, the log record $<T_i$ **commi**t> is written.
- Two approaches using logs
  - Immediate database modification
  - Deferred database modification

Database Modification

- The **immediate-modification** scheme allows updates of an uncommitted trans
- Update log record must be written ***before*** a 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 disk storage can take place at any time before or a
- Order in which blocks are output can be different from the order in which they a
- The **deferred-modification** scheme performs updates to buffer/disk only at th
  - Simplifies some aspects of recovery
  - But has overhead of storing local copy
- We cover here only the immediate-modification scheme

Transaction Commit

- A transaction is said to have committed when its commit log record is output to
  - 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 transactio

Immediate Database Modification Example

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

- Note: B_X denotes block containing X.

B_C output before T_1 commits

B_A output after T_0 commits

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 valu
    - Each time a data item X is restored to its old value V a special log rec
    - When undo of a transaction is complete, a log record <T_i **abort**> is written out (to indicate that the undo was completed)
  - **redo**(T_i) sets the value of all data items updated by T_i to the new values, g
    - No logging is done in this case

Undo and Redo Operations (Cont.)

- The **undo** and **redo** operations are used in several different circumstances
  - The **undo is** used for transaction rollback during normal operation(in c
  - The **undo** and **redo** operations are used during recovery from failure.
- We need to deal with the case where during recovery from failure another

Transaction rollback (during normal operation)

- 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 <
  - 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 <

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**>
  - It may seem strange to redo transaction T_i if the record <T_i **abort**> record is i
    - such a redo redoes all the original actions including the steps that restore

Immediate Modification Recovery Example

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

(a)	(b)	(c)
$<T_0$ start$>$	$<T_0$ start$>$	$<T_0$ start$>$
$<T_0, A, 1000, 950>$	$<T_0, A, 1000, 950>$	$<T_0, A, 1000, 950>$
$<T_0, B, 2000, 2050>$	$<T_0, B, 2000, 2050>$	$<T_0, B, 2000, 2050>$
	$<T_0$ commit$>$	$<T_0$ commit$>$
	$<T_1$ start$>$	$<T_1$ start$>$
	$<T_1, C, 700, 600>$	$<T_1, C, 700, 600>$
		$<T_1$ commit$>$

Rec
(a) u
<

(b) redo (T_0) and undo (T_1): *A* and *B* are set to 950 and 2050 and C is restored t

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

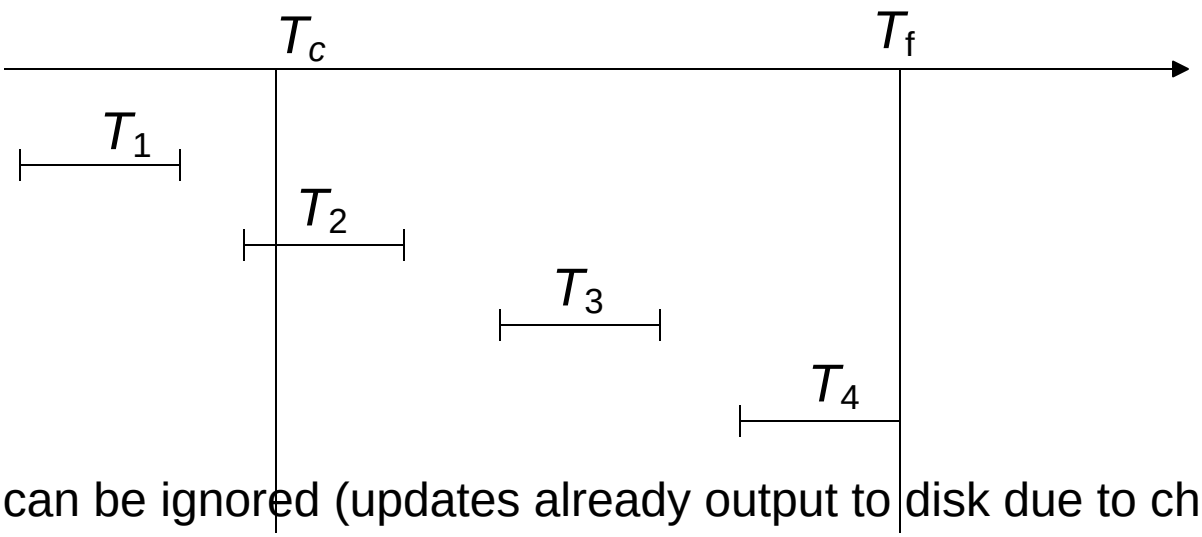
Checkpoints

- Redoing/undoing all transactions recorded in the log can be very slow
  - Processing the entire log is time-consuming if the system has run for a lon
  - We might unnecessarily redo transactions which have already output their
- Streamline recovery procedure by periodically performing **checkpointing**
  - All updates are stopped while doing checkpointing
  1. Output all log records currently residing in main memory onto stable storag
  2. Output all modified buffer blocks to the disk.
  3. Write a log record < **checkpoint** *L*> onto stable storage where *L* is a list o

Checkpoints (Cont.)

- During recovery we need to consider only the most recent transaction T_i that s
  - Scan backwards from end of log to find the most recent <**checkpoint** *L*> r
  - Only transactions that are in *L* or started after the checkpoint need to be r
  - Transactions that committed or aborted before the checkpoint already hav
- 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 tra
  - Parts of log prior to earliest <T_i **start**> record above are not needed for rec

Example of Checkpoints



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

Disk Crash

- What happens if the disk crashes and the data on it is gone?

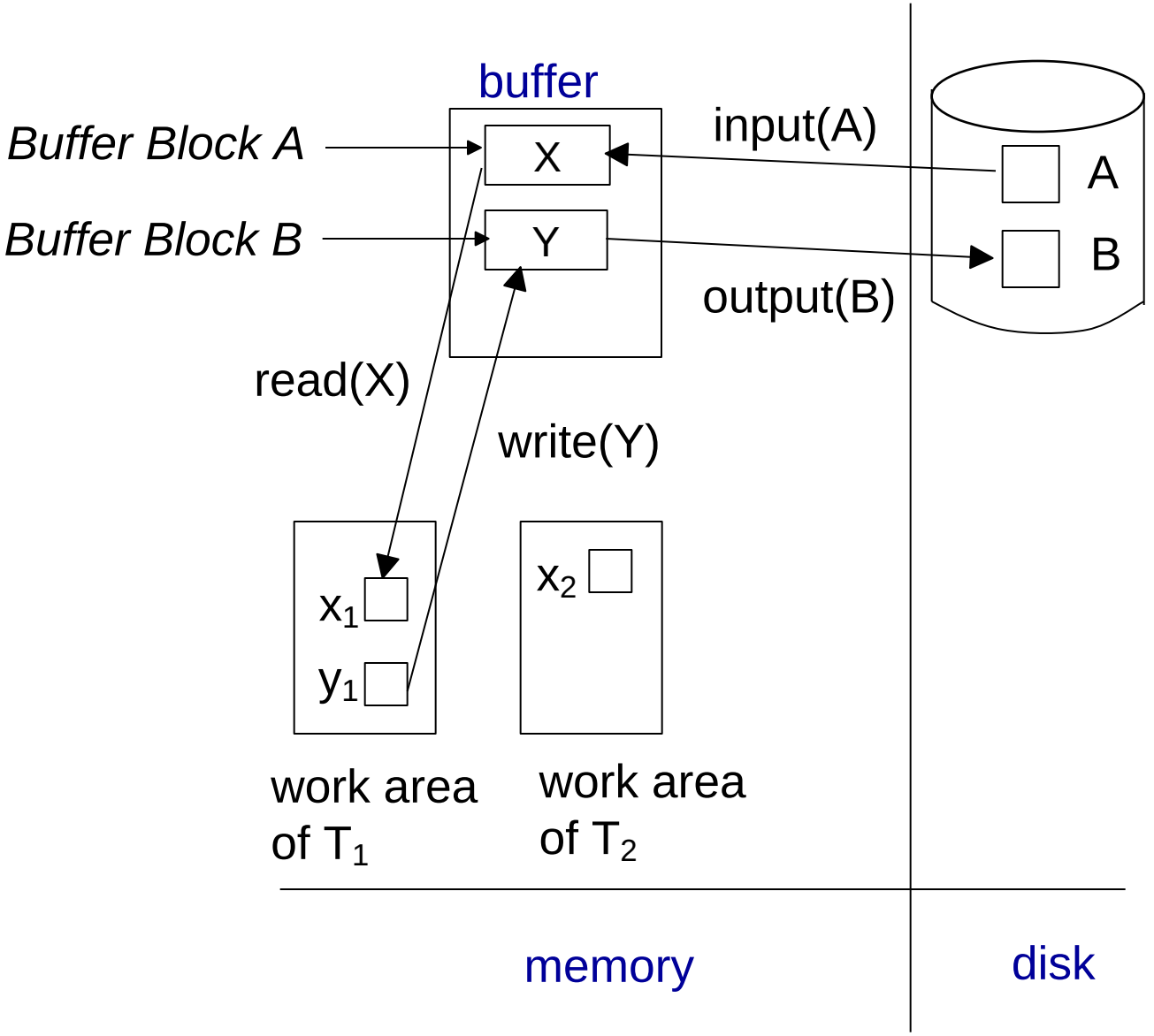
Recovery Schemes

- **So far:**
  - We covered key concepts
  - We assumed serial execution of transactions
- **Now**:
  - We discuss concurrency control issues
  - We present the components of the basic recovery algorithm
- **Later**:
  - We present extensions to allow more concurrency

Concurrency Control and Recovery

- With concurrent transactions, all transactions share a single disk buffer and a s
  - A buffer block can have data items updated by one or more transactions
- We assume that *if a transaction T_i has modified an item, no other transaction c aborted*
  - i.e. the updates of uncommitted transactions should not be visible to other
    - Otherwise how do we perform undo if T_1 updates A, then T_2 updates A
  - Can be ensured by obtaining exclusive locks on updated items and holding
- Log records of different transactions may be interspersed in the log.

Example of Data Access with Concurrent transactions



Recovery Algorithm

- **Logging** (during normal operation):
  - <T_i **start**> at transaction start
  - <T_i, X_j, V_1, V_2> for each update, and
  - <T_i **commit**> at transaction end
- **Transaction rollback (during normal operation)**
  - 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
    - 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

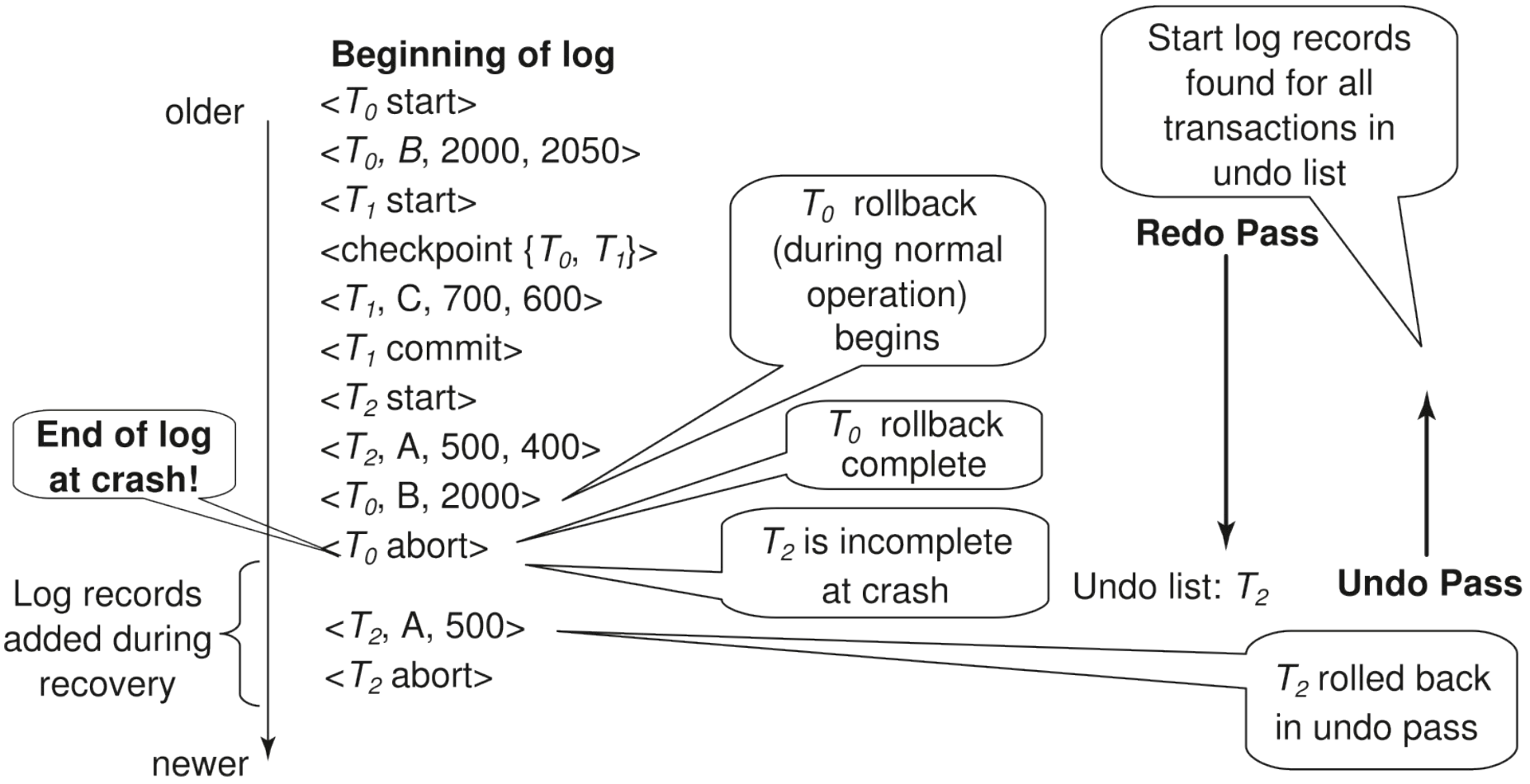
Recovery Algorithm (Cont.)

- **Recovery from failure**: Two phases
  - **Redo phase**: replay updates of **all** transactions, whether they committed,
  - **Undo phase**: undo all incomplete transactions
- **Redo 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
     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

Recovery Algorithm (Cont.)

- **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-lis
        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



Log Record Buffering

- **Log record buffering**: log records are buffered in main memory, instead of of
  - Log records are output to stable storage when a block of log records in the
- Log force is performed to commit a transaction by forcing all its log records (in
- Several log records can thus be output using a single output operation, reducin

Log Record Buffering (Cont.)

- The rules below must be followed if log records are buffered:
  - Log records are output to stable storage in the order in which they are crea
  - Transaction T_i enters the commit state only when the log record <T_i **commit**> has been output to stable storage.
  - Before a block of data in main memory is output to the database, all log re
    - This rule is called the **write-ahead logging** or **WAL** rule
      - Strictly speaking WAL only requires undo information to be output

Database Buffering

- 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
  - If the block chosen for removal has been updated, it must be output to disk
- The recovery algorithm supports the **no-force policy**: i.e., updated blocks nee
  - **force policy**: requires updated blocks to be written at commit
    - More expensive commit
- The recovery algorithm supports the **steal policy**:i.e., blocks containing updat

Database Buffering (Cont.)

- If a block with uncommitted updates is output to disk, log records with undo inf
  - (Write ahead logging)
- No updates should be in progress on a block when it is output to disk. Can be
  - Before writing a data item, transaction acquires exclusive lock on block co
  - Lock can be released once the write is completed.
    - Such locks held for short duration are called **latches**.
- **To output a block to disk**
  1. First acquire an exclusive 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 latch on the block

Buffer Management (Cont.)

- Database buffer can be implemented either
  - in an area of real main-memory reserved for the database, or
  - in virtual memory
- Implementing buffer in reserved main-memory has drawbacks:
  - Memory is partitioned before-hand between database buffer and applicatio
  - Needs may change, and although operating system knows best how mem

Buffer Management (Cont.)

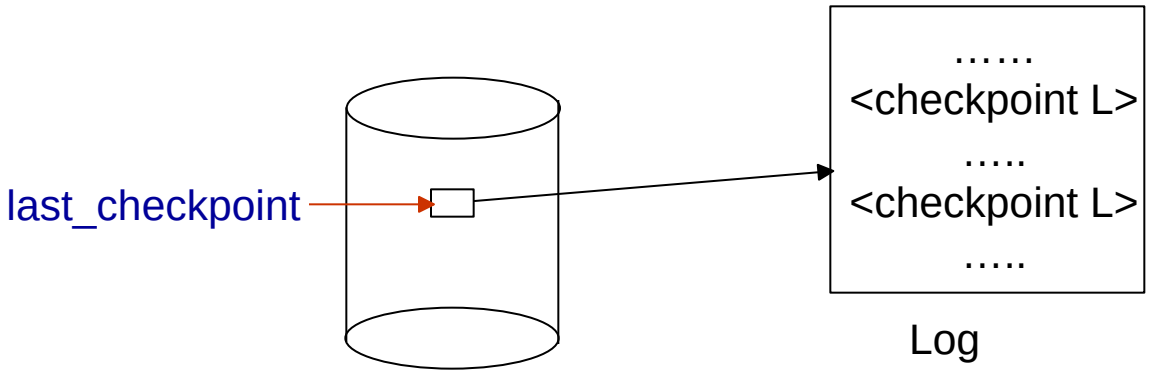
- Database buffers are generally implemented in virtual memory in spite of some
  - When operating system needs to evict a page that has been modified, the
  - When database decides to write buffer page to disk, buffer page may be i
    - Known as **dual paging** problem.
  - Ideally when OS needs to evict a page from the buffer, it should pass con
    1. Output the page to database instead of to swap space (making sure t
    2. Release the page from the buffer, for the OS to use
  - Dual paging can thus be avoided, but common operating systems do not

Fuzzy Checkpointing

- 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 list *M* of modified buffer blocks
  4. Now permit transactions to proceed with their actions
  5. Output to disk all modified buffer blocks in list *M*
     - blocks should not be updated while being output
     - Follow WAL: all log records pertaining to a block must be outpu
  6. Store a pointer to the **checkpoint** record in a fixed position **last_c**

Fuzzy Checkpointing (Cont.)

- When recovering using a fuzzy checkpoint, start scan from the **checkpo**
  - Log records before **last_checkpoint** have their updates reflected in
  - Incomplete checkpoints, where system had crashed while performin



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 st
  - Periodically **dump** the entire content of the database to stable storage
  - No transaction may be active during the dump procedure; a procedure
    - Output all log records currently residing in main memory onto stable
    - Output all buffer blocks onto the disk.
    - Copy the contents of the database to stable storage.
    - Output a record <**dump**> to log on stable storage.

Recovering from Failure of Non-Volatile Storage

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

Recovery with Early Lock Release and Logical Undo

Recovery with Early Lock Release

- Support for high-concurrency locking techniques, such as those used for B^+-tre
  - Supports “logical undo”
- Recovery based on “repeating history”, whereby recovery executes exactly the

Logical Undo Logging

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

Physical Redo

- Redo information is logged **physically** (that is, new value for each write) even
  - Logical redo is very complicated since database state on disk may not be '
  - Physical redo logging does not conflict with early lock release

Operation Logging

- Operation logging is done as follows:
  1. When operation starts, log <T_i, O_j, **operation-begin**>. Here O_j is a unique
  2. While operation is executing, normal log records with physical redo and ph
  3. When operation completes, <T_i, O_j, **operation-end**, U> is logged, where

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

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

Physical redo of steps in insert

Operation Logging (Cont.)

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

Transaction Rollback with Logical Undo

Rollback of transaction T_i is done as follows:

- Scan the log backwards

1. If a log record $<T_i, X, V_1, V_2>$ is found, perform the undo and log a al $<T_i, X$
2. If a $<T_i, O_j,$ **operation-end**, $U>$ record is found
   - Rollback the operation logically using the undo information U.
     - Updates performed during roll back are logged just like during norr
     - At the end of the operation rollback, instead of logging an **operati** record $<T_i, O_j,$ **operation-abort**$>$.
   - Skip all preceding log records for T_i until the record $<T_i, O_j$ **operation-begin**$>$ is found

Transaction Rollback with Logical Undo (Cont.)

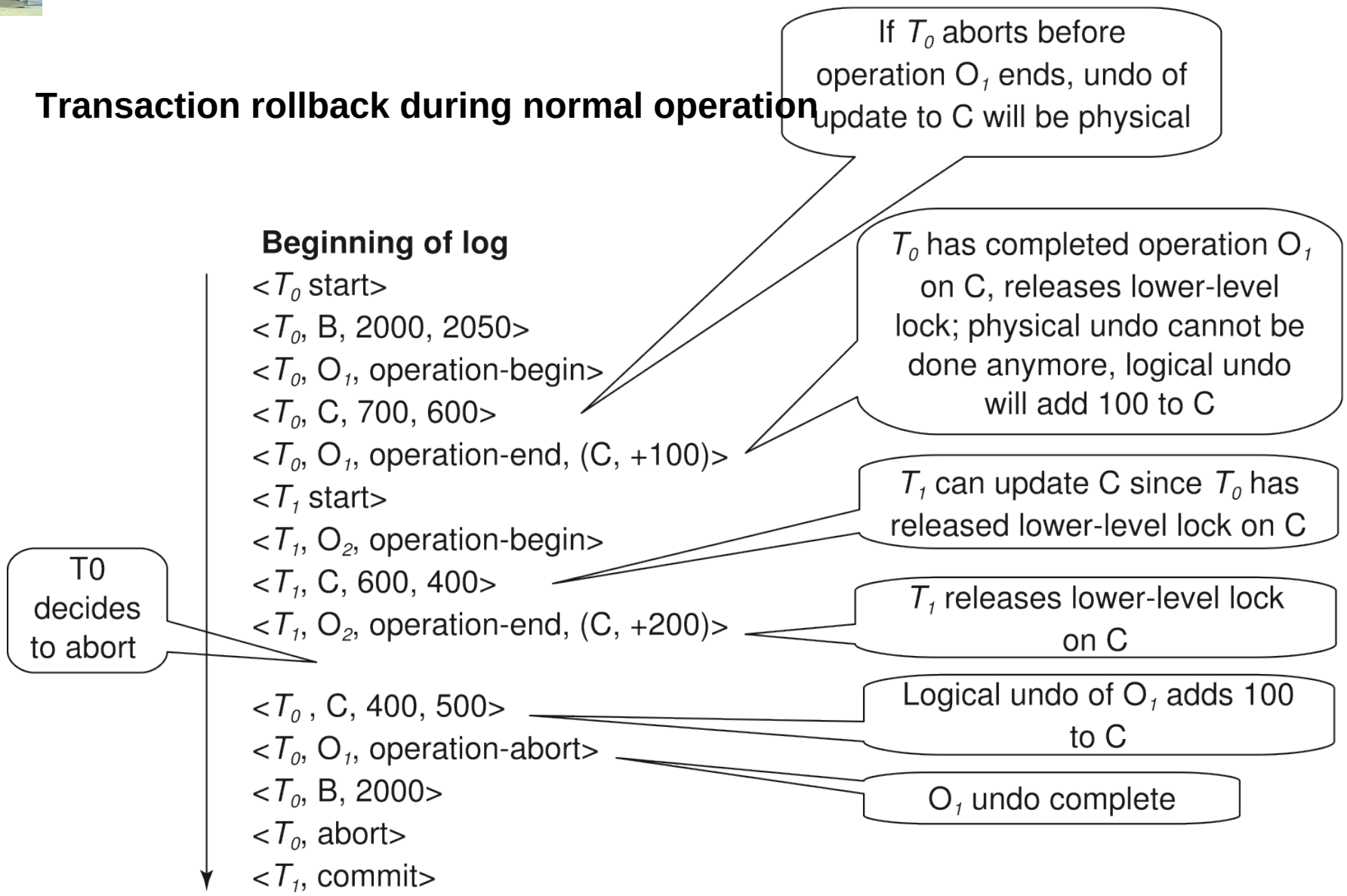
- Transaction rollback, scanning the log backwards (cont.):
  3. If a redo-only record is found ignore it
  4. If a <T_i, O_j, **operation-abort**> record is found:
     - skip all preceding log records for T_i until the record <T_i, O_j, **operation-begi**n> is found.
  5. Stop the scan when the record <T_i, start> is found
  6. Add a <T_i, **abort**> record to the log

Some points to note:

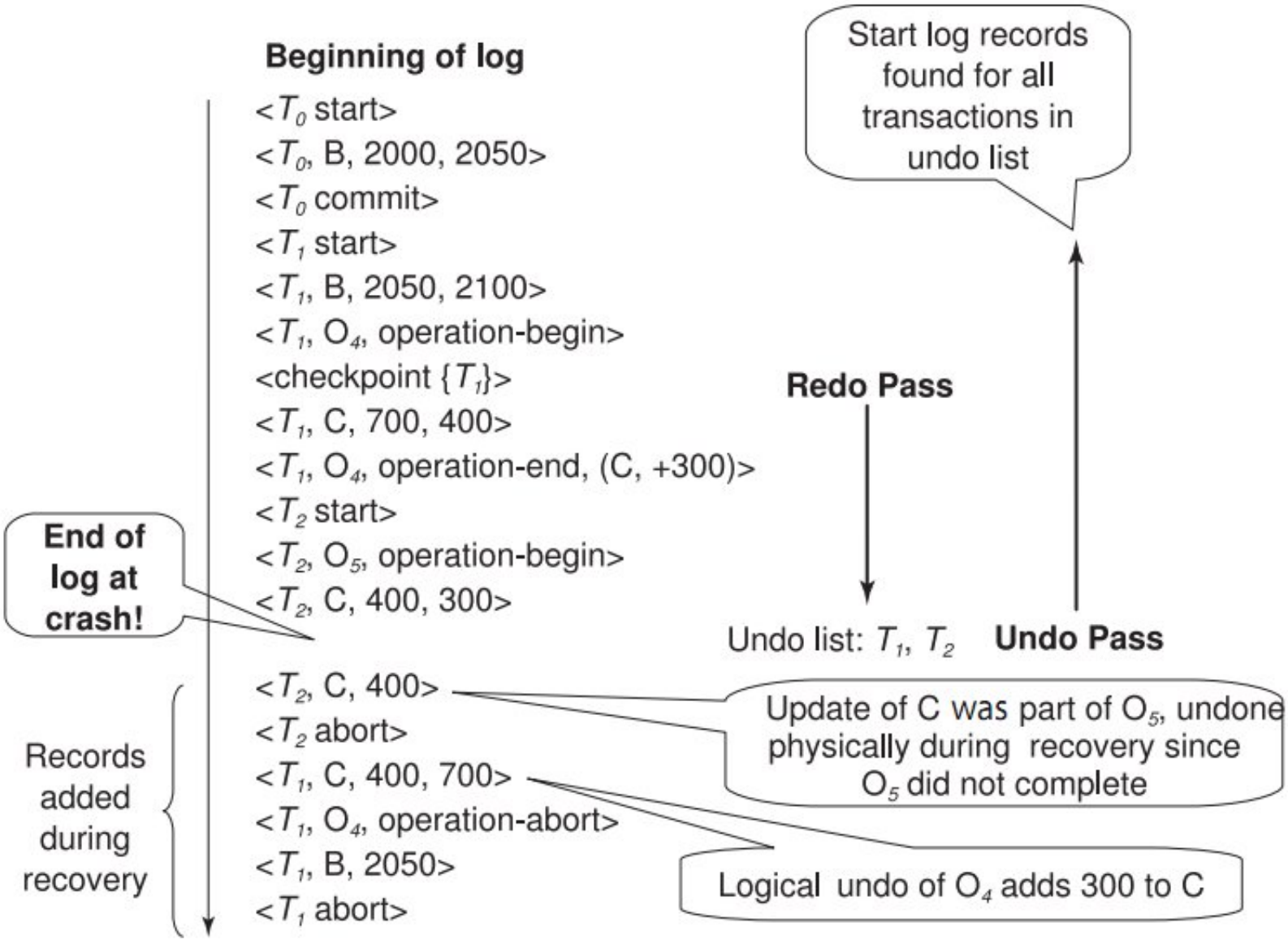
- Cases 3 and 4 above can occur only if the database crashes while a transacti
- Skipping of log records as in case 4 is important to prevent multiple rollback of

Transaction Rollback with Logical Undo

- **Transaction rollback during normal operation**



Failure Recovery with Logical Undo



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

Recovery Algorithm with Logical Undo

Basically same as earlier algorithm, except for changes described earlier fo

1. (**Redo phase**): Scan log forward from last < **checkpoint** *L*> record till end of l
    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*

    This brings database to state as of crash, with committed as well as uncommit

    Now *undo-list* contains transactions that are **incomplete**, that is, have neither

Recovery with Logical Undo (Cont.)

Recovery from system crash (cont.)

2. (**Undo phase**): Scan log backwards, performing undo on log records of transa
   - Log records of transactions being rolled back are processed as described
     - 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 **abo**
   - 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 **commi**

ARIES Recovery Algorithm

ARIES

- ARIES is a state of the art recovery method
  - Incorporates numerous optimizations to reduce overheads during normal
  - The recovery algorithm we studied earlier is modeled after ARIES, but gre
- Unlike the recovery algorithm described earlier, ARIES
  1. Uses **log sequence number (LSN)** to identify log records
     - Stores LSNs in pages to identify what updates have already been app
  2. Physiological redo
  3. Dirty page table to avoid unnecessary redos during recovery
  4. Fuzzy checkpointing that only records information about dirty pages, and
     - More coming up on each of the above …

ARIES Optimizations

- **Physiological redo**
  - Affected page is physically identified, action within page can be logical
    - Used to reduce logging overheads
      - e.g. when a record is deleted and all other records have to be mo
        - Physiological redo can log just the record deletion
        - Physical redo would require logging of old and new values for r
    - Requires page to be output to disk atomically
      - Easy to achieve with hardware RAID, also supported by some disk
      - Incomplete page output can be detected by checksum techniques
        - But extra actions are required for recovery
        - Treated as a media failure

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
  - Page LSN
  - Log records of several different types
  - Dirty page table

ARIES Data Structures: Page LSN

- Each page contains a **PageLSN** which is the LSN of the last log record whose
  - 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

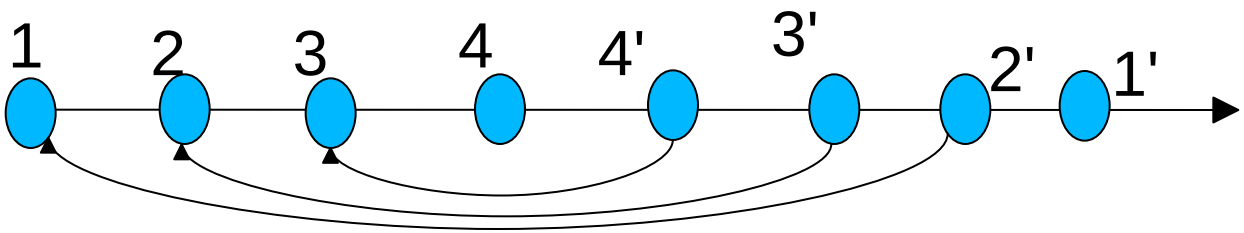
ARIES Data Structures: Log Record

- 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 a
  - Serves the role of operation-abort log records used in earlier recovery algorithr
  - 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

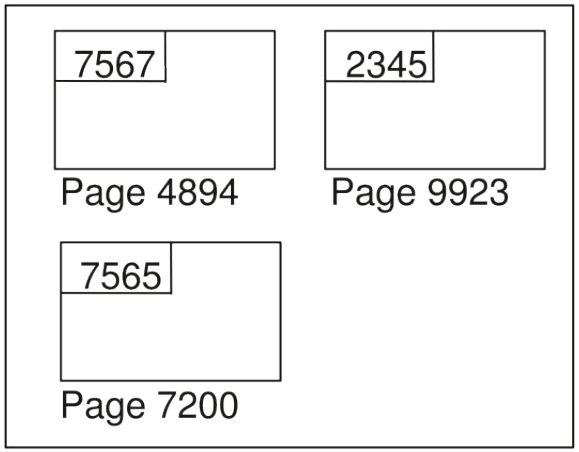
LSN	TransID	UndoNextLSN	RedoInfo



ARIES Data Structures: DirtyPage Table

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

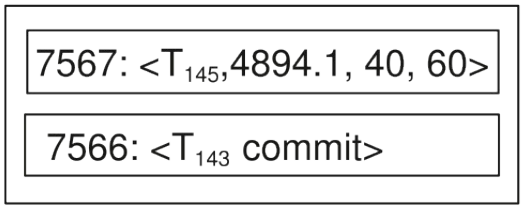
ARIES Data Structures



Database Buffer

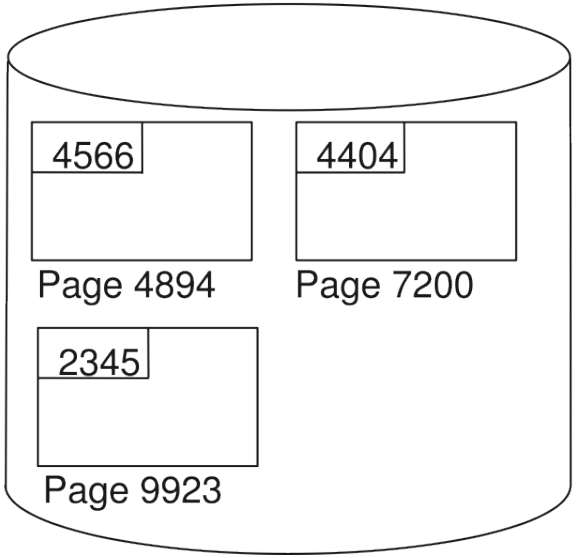
PageID	PageLSN	RecLSN
4894	7567	7564
7200	7565	7565

Dirty Page Table

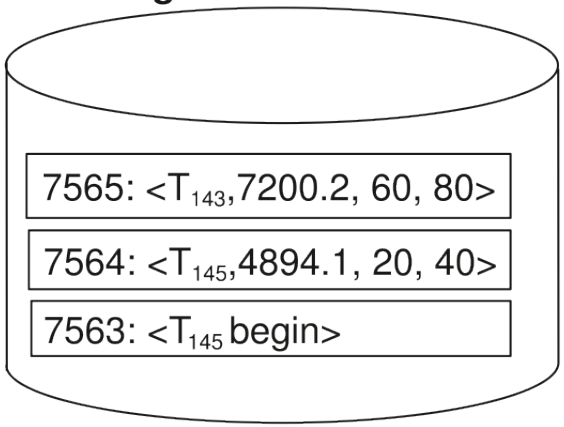


Log Buffer (PrevLSN and UndoNextLSN fields not shown)

Stable data



Stable log



ARIES Data Structures: Checkpoint Log

- **Checkpoint log record**
  - Contains:
    - DirtyPageTable and list of active transactions
    - For each active transaction, LastLSN, the LSN of the last log record wr
      transaction
  - Fixed position on disk notes LSN of last completed
    checkpoint log record
- Dirty pages are not written out at checkpoint time
  - Instead, they are flushed out continuously, in the background
- Checkpoint is thus very low overhead
  - can be done frequently

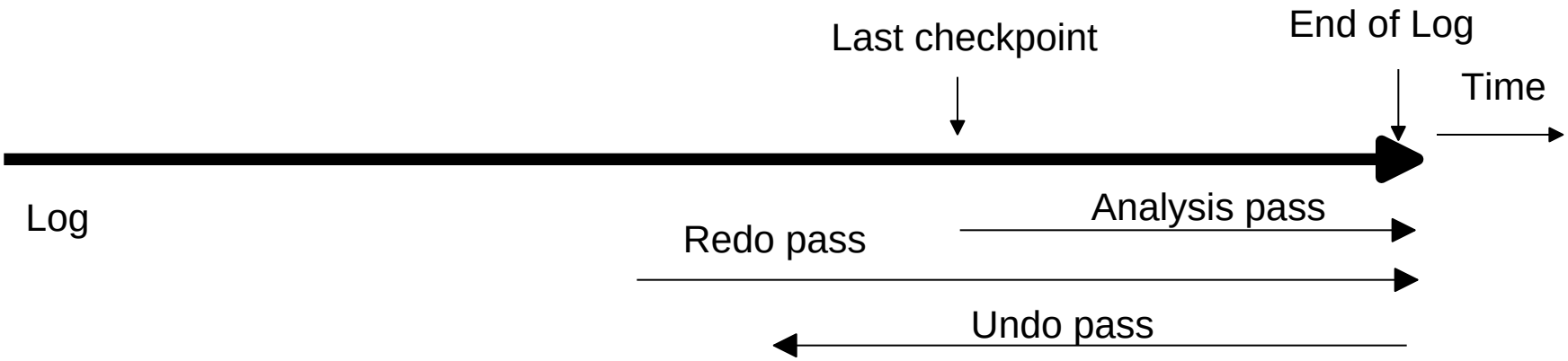
ARIES Recovery Algorithm

ARIES recovery involves three passes

- Analysis pass: Determines
  - Which transactions to undo
  - Which pages were dirty (disk version not up to date) at time of crash
  - RedoLSN: LSN from which redo should start
- Redo pass:
  - Repeats history, redoing all actions from RedoLSN
    - RecLSN and PageLSNs are used to avoid redoing actions already refle
- 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

Aries Recovery: 3 Passes

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



ARIES Recovery: Analysis

Analysis pass

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

ARIES Recovery: Analysis (Cont.)

Analysis pass (cont.)

- Scans forward from checkpoint
  - If any log record found for transaction not in undo-list, adds transaction to
  - Whenever an update log record is found
    - If page is not in DirtyPageTable, it is added with RecLSN set to LSN of record
  - If transaction end log record found, delete transaction from undo-list
  - Keeps track of last log record for each transaction in undo-list
    - May be needed for later undo
- At end of analysis pass:
  - RedoLSN determines where to start redo pass
  - RecLSN for each page in DirtyPageTable used to minimize redo work
  - All transactions in undo-list need to be rolled back

ARIES Redo Pass

Redo Pass: Repeats history by replaying every action not already reflected in the

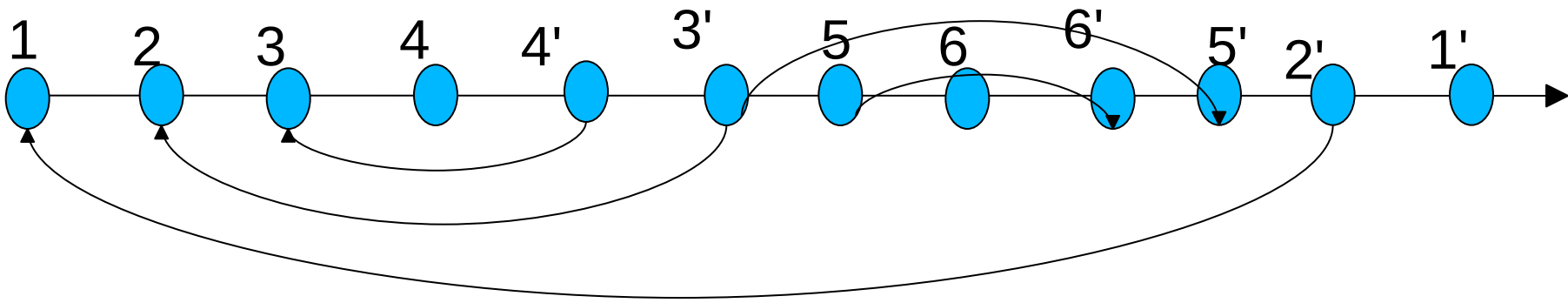
- Scans forward from RedoLSN. Whenever an update log record is found:

1. If the page is not in DirtyPageTable or the LSN of the log record is less tha
2. Otherwise fetch the page from disk. If the PageLSN of the page fetched f

NOTE: if either test is negative the effects of the log record have already appe

ARIES Undo Actions

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

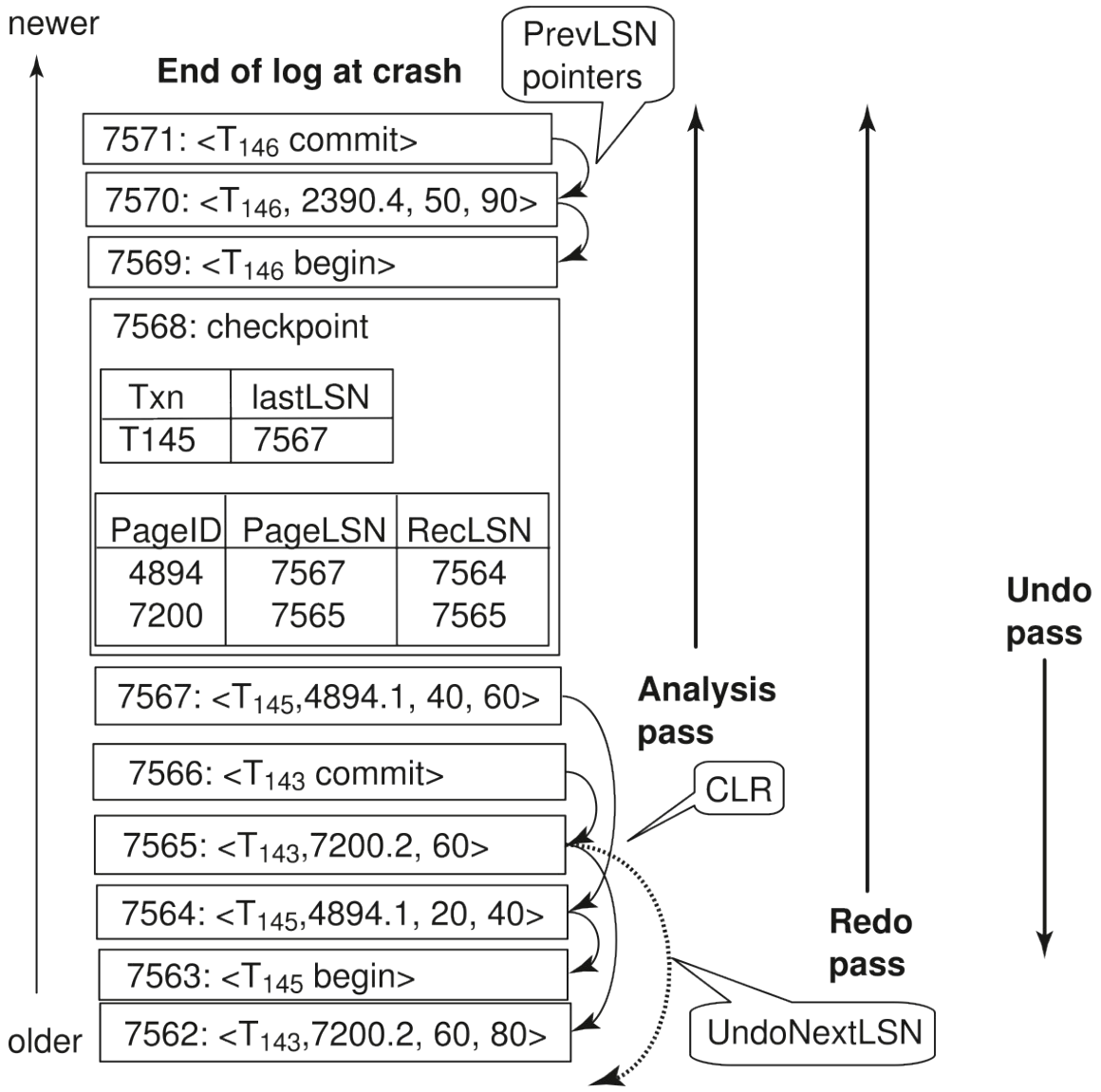


ARIES: Undo Pass

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 reco pass.
    - At each step pick largest of these LSNs to undo, skip back to it and un
    - After undoing a log record
      - For ordinary log records, set next LSN to be undone for transaction
      - For compensation log records (CLRs) set next LSN to be undo to U
        - All intervening records are skipped since they would have been
- Undos performed as described earlier

Recovery Actions in ARIES



Other ARIES Features

- Recovery Independence
  - Pages can be recovered independently of others
    - E.g. if some disk pages fail they can be recovered from a backup while used
- Savepoints:
  - Transactions can record savepoints and roll back to a savepoint
    - Useful for complex transactions
    - Also used to rollback just enough to release locks on deadlock

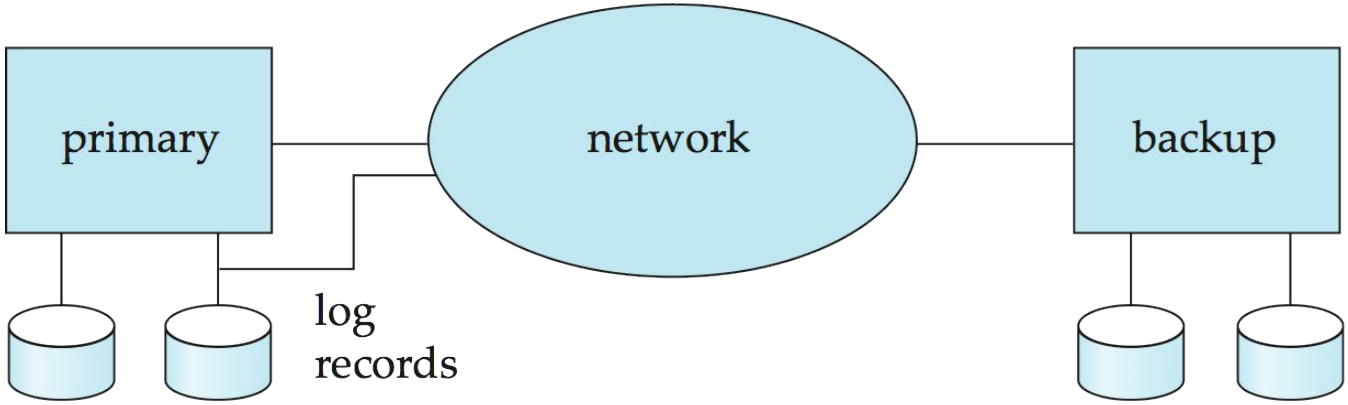
Other ARIES Features (Cont.)

- Fine-grained locking:
  - Index concurrency algorithms that permit tuple level locking on indices can
    - These require logical undo, rather than physical undo, as in earlier reco algorithm
- Recovery optimizations: For example:
  - Dirty page table can be used to prefetch pages during redo
  - Out of order redo is possible:
    - redo can be postponed on a page being fetched from disk, and performed when page is fetched.
    - Meanwhile other log records can continue to be processed

Remote Backup Systems

Remote Backup Systems

- Remote backup systems provide high availability by allowing transaction proce



Remote Backup Systems (Cont.)

- **Detection of failure**: Backup site must detect when primary site has failed
  - to distinguish primary site failure from link failure maintain several commun
  - Heart-beat messages
- **Transfer of control**:
  - To take over control backup site first perform recovery using its copy of the primary.
    - Thus, completed transactions are redone and incomplete transactions
  - 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

Remote Backup Systems (Cont.)

- **Time to recover**: To reduce delay in takeover, backup site periodically proces
- **Hot-Spare** configuration permits very fast takeover:
  - Backup continually processes redo log record as they arrive, applying the
  - When failure of the primary is detected the backup rolls back incomplete tr
- Alternative to remote backup: distributed database with replicated data
  - Remote backup is faster and cheaper, but less tolerant to failure
    - more on this in Chapter 19

Remote Backup Systems (Cont.)

- Ensure durability of updates by delaying transaction commit until update is log
- **One-safe:** commit as soon as transaction's commit log record is written at prin
  - Problem: updates may not arrive at backup before it takes over.
- **Two-very-safe:** commit when transaction's commit log record is written at prin
  - Reduces availability since transactions cannot commit if either site fails.
- **Two-safe:** proceed as in two-very-safe if both primary and backup are active.
  - Better availability than two-very-safe; avoids problem of lost transactions i

End of Chapter 16