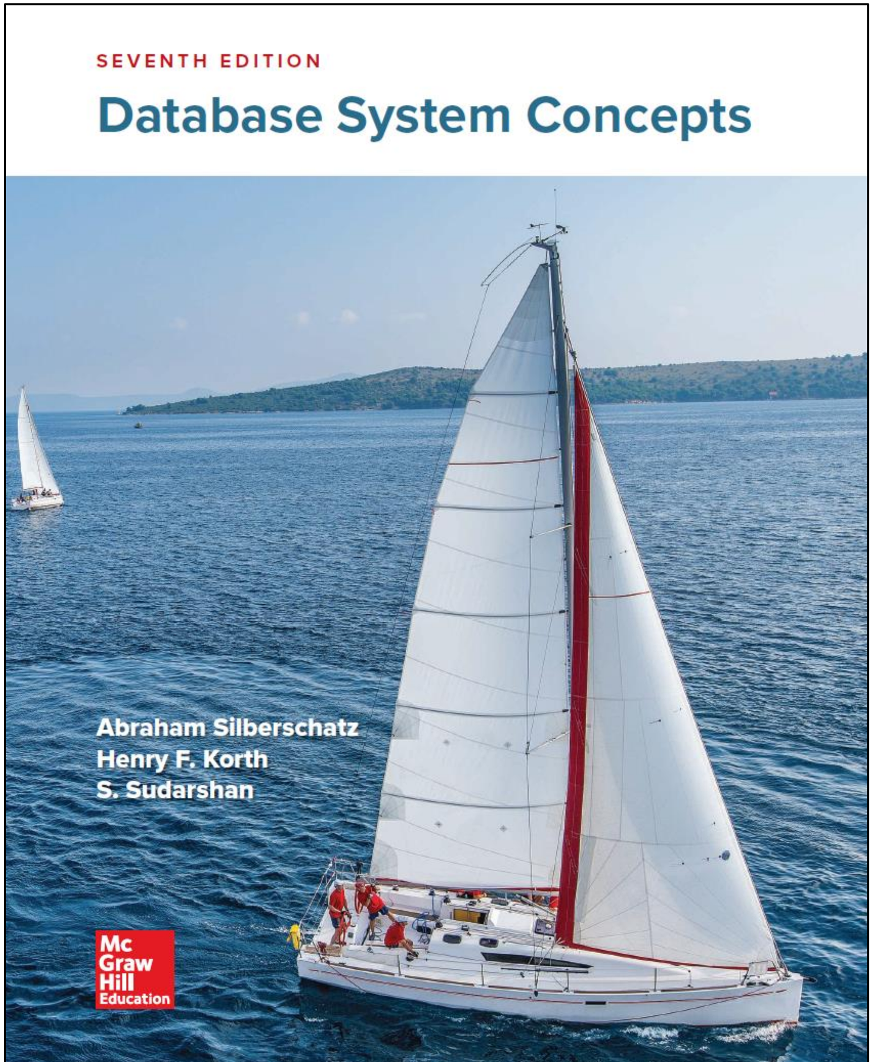
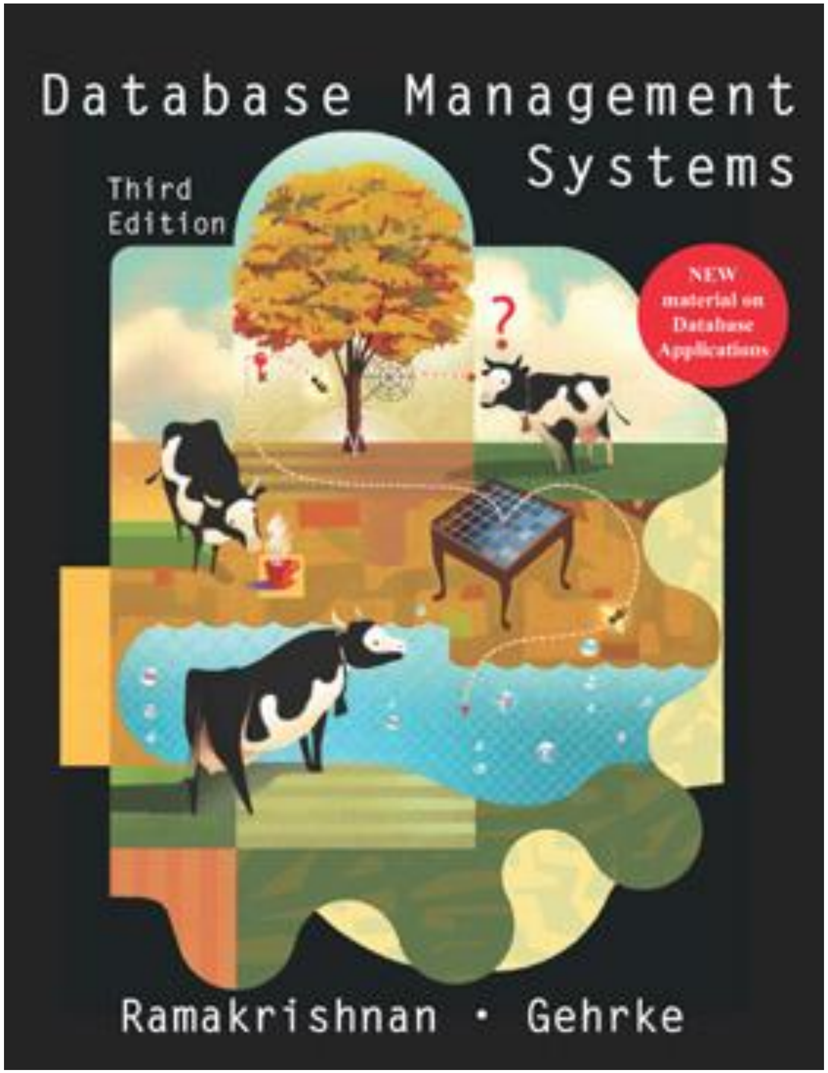
Data Administration in Information Systems

Database recovery

Database recovery



Database recovery



Failure Classification

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

Recovery Algorithms

- Suppose transaction T_i 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
  - Not modifying the database may result in lost updates if failure occurs just after transaction commits
- Recovery algorithms have two parts
  1. Actions taken during normal transaction processing to ensure enough information exists to recover from failures
  2. Actions taken after a failure to bring the database contents to a state that ensures atomicity, consistency and durability

Storage Structure

- **Volatile storage**
  - Does not survive system crashes
  - Examples: main memory, cache memory
- **Non-volatile storage**
  - Survives system crashes
  - Examples: disk, tape, flash memory, non-volatile RAM
  - But may still fail, losing data
- **Stable storage**
  - An ideal form of storage that survives all failures
  - Approximated by maintaining multiple copies on non-volatile media
    - RAID is not enough; copies should be at different remote sites to protect against disasters such as fire or flooding

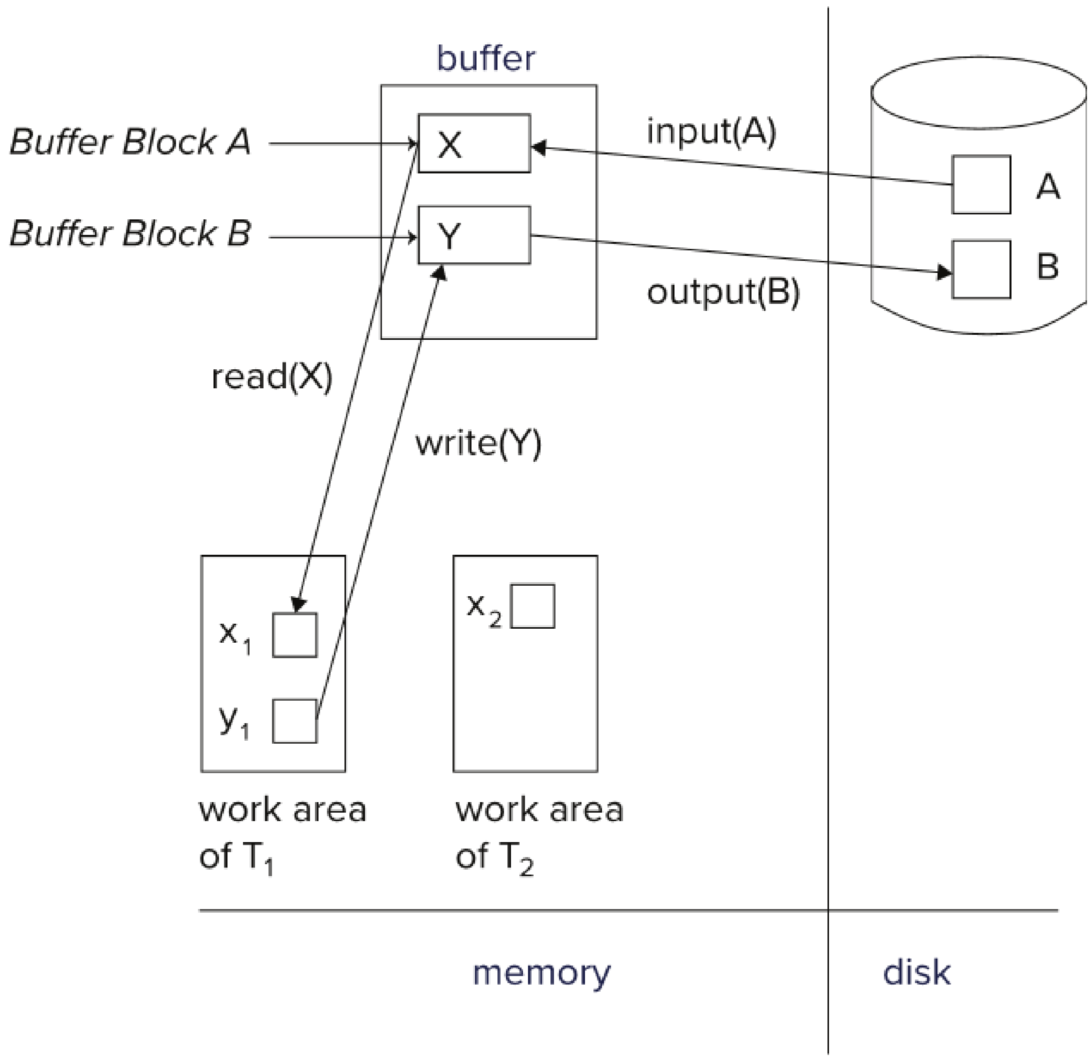
Data Access

- **Physical blocks** are those blocks residing on the disk.
- **Buffer blocks** are the blocks residing temporarily in main memory.
- Block movements between disk and main memory are initiated through the following two operations:
  - **input**(*B*) transfers the physical block *B* to main memory.
  - **output**(*B*) transfers the buffer block *B* to the disk, and replaces the appropriate physical block there.
- We assume, for simplicity, that each data item fits in, and is stored inside, a single block.

Data Access (Cont.)

- 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

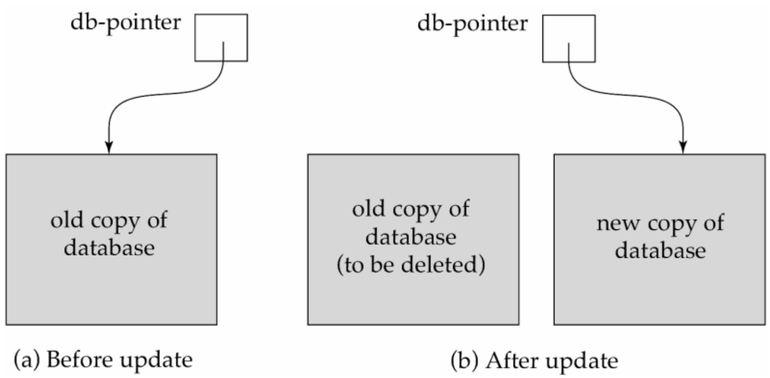
Example of Data Access



Recovery and Atomicity

- To ensure atomicity despite failures, we first output information describing the modifications to stable storage without 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-copy**
  - Briefly described in the book

shadow-copy



Log-Based Recovery

- A **log** is a sequence of **log records**.
  - The records keep information about update activities on the database.
  - The **log** is kept on stable storage.
- 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 is written:

  <T_i, X, V_1, V_2>

  - 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, the following log record is written:

  <T_i **commit**>

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

Database Modification

- Database modifications can be:
  - **immediate**: updates of an uncommitted transaction are made to the disk before the transaction commits
  - **deferred**: updates to disk only at the time of transaction commit
- Output of updated blocks to disk can take place at any time before or after transaction commit
- Update log record must be written *before* database item is written
  - For the moment, we assume that the log record is output directly to stable storage

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, C	*C* output *before* T_1 commits
<T_1 **commit**>			
		A	*A* output *after* T_0 commits

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**>

Undo and Redo Operations

- **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 (**compensation log record**)
    - 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

Recovering from Failure (Cont.)

- Suppose that transaction T_i was undone earlier, the <T_i **abort**> record was written to the log, and then a failure occurs
- On recovery from failure, transaction T_i is redone
  - Such a **redo** redoes all the original actions of transaction T_i including the steps that restored old values!
    - This is known as **repeating history**

Recovery Example

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

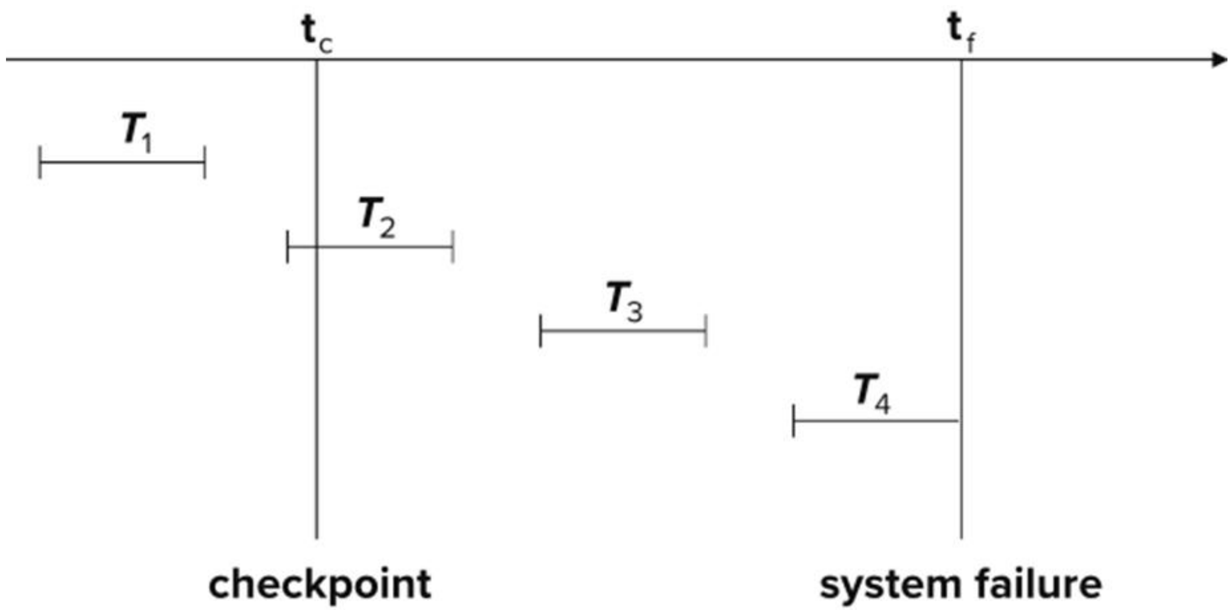
(a)	(b)	(c)
<T_0 start>	<T_0 start>	<T_0 start>
<T_0, A, 1000, 950>	<T_0, A, 1000, 950>	<T_0, A, 1000, 950>
<T_0, B, 2000, 2050>	<T_0, B, 2000, 2050>	<T_0, B, 2000, 2050>
	<T_0 commit>	<T_0 commit>
	<T_1 start>	<T_1 start>
	<T_1, C, 700, 600>	<T_1, C, 700, 600>
		<T_1 commit>

- Recovery actions in each case above are:
  - **a)** **undo**(T_0): B is restored to 2000 and A to 1000, and 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.

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 long time
  - 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.
  4. All updates are on hold while doing checkpointing

Example of Checkpoints



- Recovery after system failure:
  - Ignore T_1 (updates already output to disk due to checkpoint)
  - Redo T_2 and T_3
  - Undo T_4

Checkpoints (Cont.)

- 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.

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, no crash):
  - 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> (**compensation log record**)
  - Once the record <T_i **start**> is found stop the scan and write the log record <T_i **abort**>

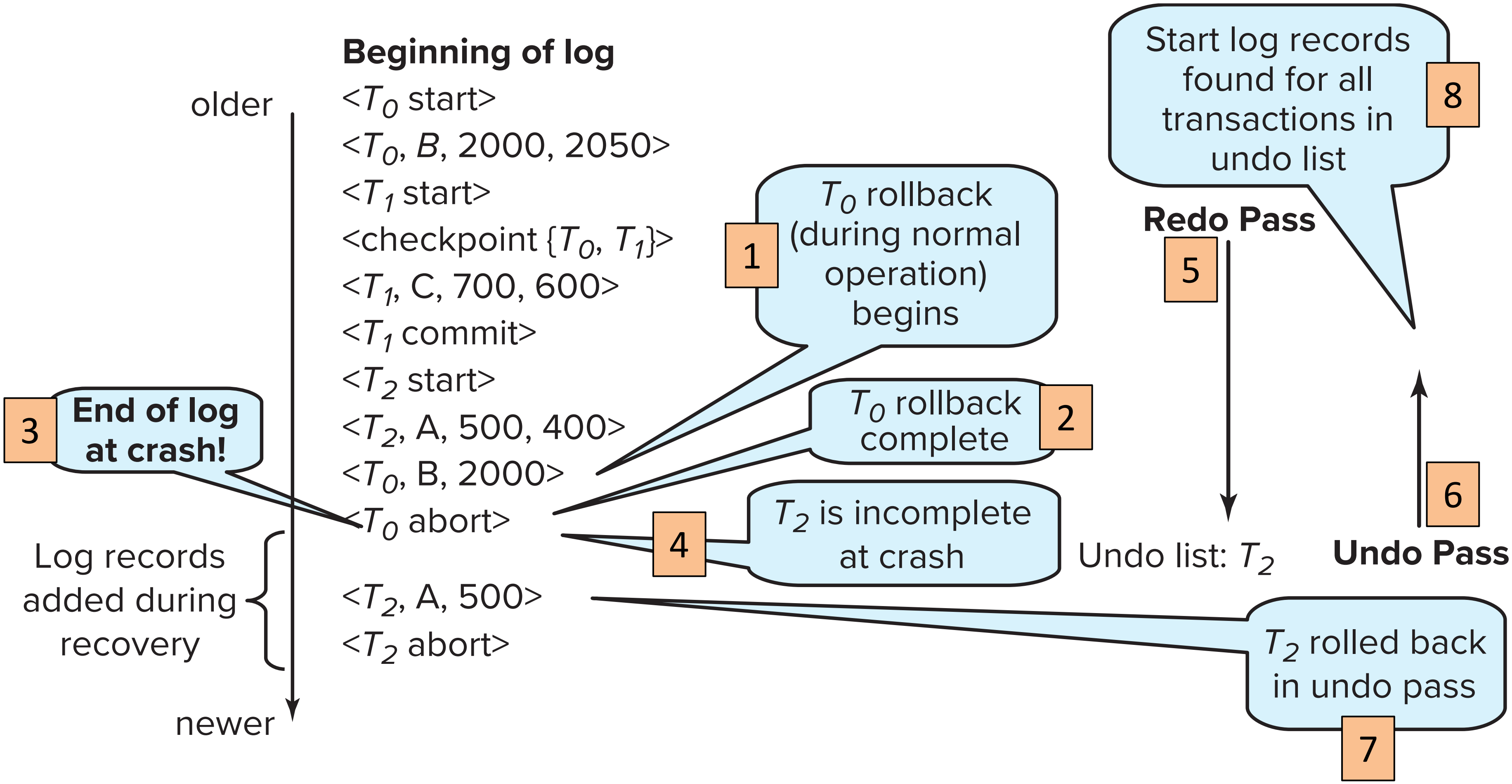
Recovery Algorithm (Cont.)

- **Recovery from failure**: Two phases
  - **Redo phase**: replay updates of *all* transactions, whether they committed, aborted, or are incomplete
  - **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 $<T_i, X_j, V_1, V_2>$ or $<T_i, X_j, V_2>$ 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

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-list perform the following rollback actions:
        1. perform undo by writing V_1 to X_j
        2. write a **compensation 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
        1. 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 being output directly to stable storage.
  - Log records are output to stable storage when a block of log records in the buffer is full, or a **log force** operation is executed.
- Log force is performed to commit a transaction by forcing all its log records (including the commit record) to stable storage.
- Several log records can thus be output using a single output operation, reducing the I/O cost.

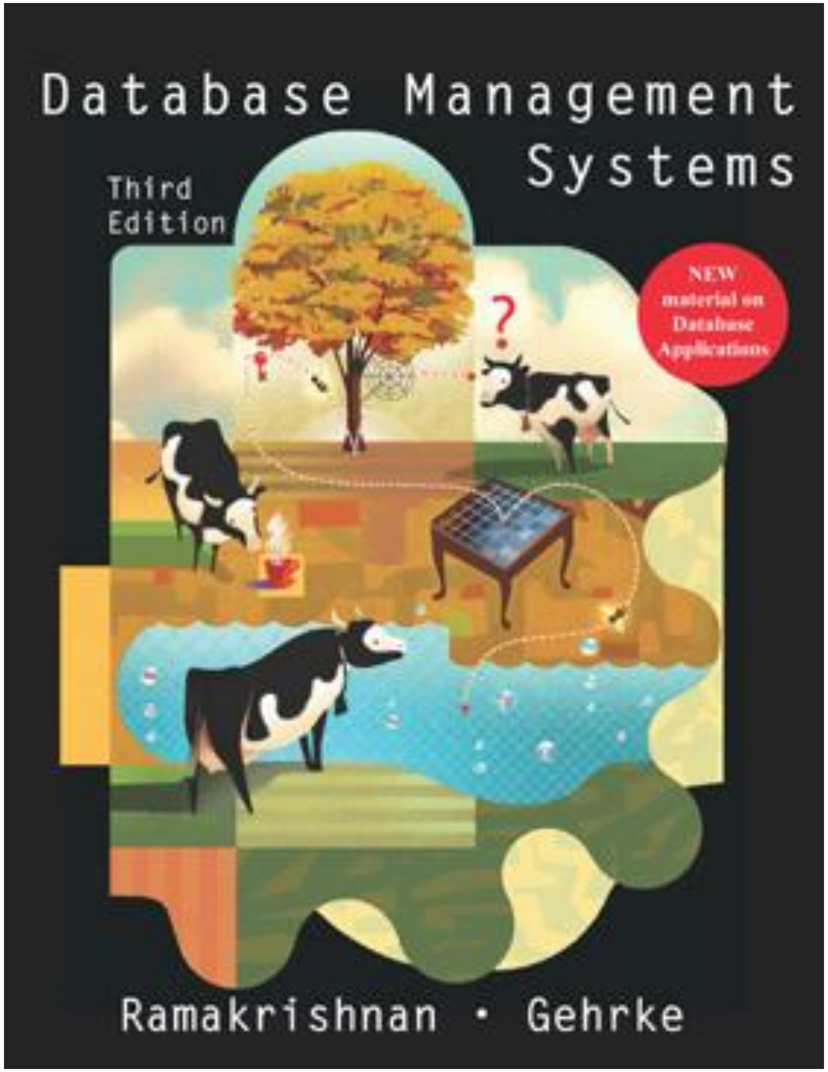
Log Record Buffering (Cont.)

- The rules below must be followed if log records are buffered:
  - Log records are output to stable storage in the order in which they are created.
  - Transaction T_i enters the commit state only when the log record <T_i **commit**> has been output to stable storage.
  - Before a block of data in main memory is output to the database, all log records pertaining to data in that block must have been output to stable storage.
    - This rule is called the **write-ahead logging (WAL)** rule

Log Record Buffering (Cont.)

- After the log records have been written to disk, blocks of data in main memory are output to the database.
- 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**.
- **To output a block to disk**
  1. First acquire an exclusive latch on the block
     - 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

ARIES (Algorithm for Recovery and Isolation Exploiting Semantics)



xvi DATABASE MANAGEMENT SYSTEMS

ARIES

- ARIES is a state-of-the-art recovery algorithm
  - The recovery algorithm we studied earlier is modeled after ARIES, but greatly simplified
- In ARIES,
  - Blocks are called **pages**
  - Every log record has a **log sequence number (LSN)**
  - Every **page** in the database contains the **LSN** of the most recent log record that changed that page
    - This is called the **pageLSN**
    - Updating a **page** creates a new log record and sets the **pageLSN** of that page to the **LSN** of that log record.
  - Each log record contains a pointer to the previous log record of the same transaction
    - This is called the **prevLSN**
    - The first log record of a transaction has **prevLSN** = NULL

ARIES (Cont.)

- Besides the log, ARIES uses the two additional data structures

- **Dirty page table**
  - Contains one entry for each *dirty page* in the buffer, i.e. a page with changes that are not yet reflected on disk.
  - Each entry contains a **recLSN**, which is the LSN of the first log record that caused the page to become dirty.

- **Transaction table**
  - Contains one entry for each *active transaction*.
  - Each entry contains a **lastLSN**, which is the LSN of the most recent log record for the transaction.

Data Structures in ARIES

pageID	recLSN
P500	
P600	
P505	

DIRTY PAGE TABLE

transID	lastLSN
T1000	
T2000	

TRANSACTION TABLE

prevLSN	transID	type	pageID	length	offset	before-image	after-image
	T1000	update	P500	3	21	ABC	DEF
	T2000	update	P600	3	41	HIJ	KLM
	T2000	update	P500	3	20	GDE	QRS
	T1000	update	P505	3	21	TUV	WXY

LOG

prevLSN	transID	type	pageID	length	offset	before-image	after-image

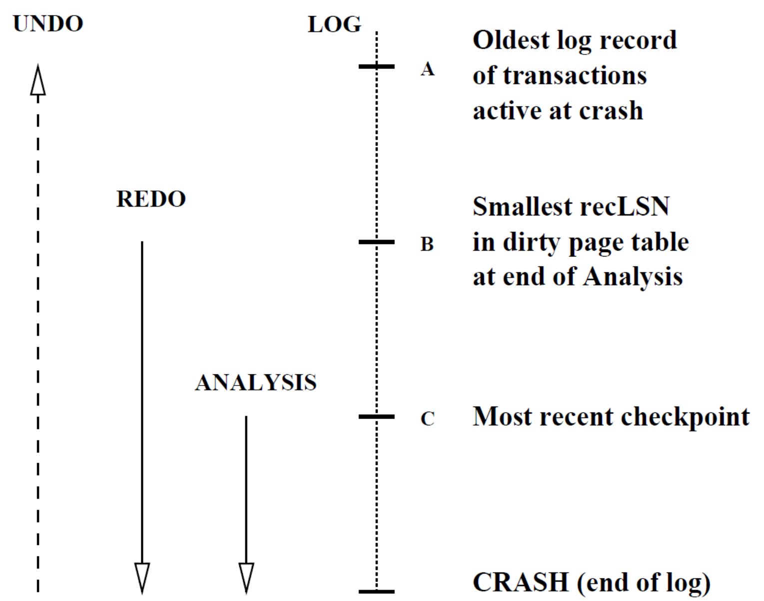
Fields common to all log records (prevLSN, transID, type) — Additional fields for update log records (pageID, length, offset, before-image, after-image)

Checkpoints in ARIES

- Checkpointing in ARIES has multiple steps:
  - A **begin_checkpoint** record is written to indicate when the checkpoint starts.
  - An **end_checkpoint** record is constructed, including the current contents of the **transaction table** and of the **dirty page table**.
    - While the **end_checkpoint** record is being constructed, the system continues executing transactions and possibly writing other log records.
  - The system writes the **end_checkpoint** record to stable storage.
    - The **transaction table** and the **dirty page table** are accurate at the time of the **begin_checkpoint** record.
  - The system writes the **LSN** of the **begin_checkpoint** record to a special position on disk. This checkpoint is now complete.

Recovery in ARIES

- The recovery process in ARIES has three phases:
  1. **Analysis**: Identifies dirty pages in the buffer pool (i.e., changes that have not been written to disk) and active transactions at the time of the crash.
  2. **Redo**: Repeats all actions, starting from an appropriate point in the log, and restores the database state to what it was at the time of the crash.
  3. **Undo**: Undoes the actions of transactions that did not commit, so that the database reflects only the actions of committed transactions.



Analysis phase

- Analysis begins at the most recent checkpoint
  - Initializes the **dirty page table** and **transaction table** from that checkpoint.

- Analysis proceeds forward until the end of the log
  - New dirty pages are added the **dirty page table** with the **recLSN** of the first log record where those pages have become dirty.
  - Transactions are added to (or updated in) the **transaction table** with the **lastLSN** of the last log record where those transactions appeared.
  - Completed transactions are removed from the **transaction table**, and are marked for **redo**.
  - The remaining transactions were active at the time of the crash, and are marked for **undo**.

Redo phase

- Redo begins at the smallest **recLSN** found in the **dirty page table**
  - This **recLSN** is the oldest update that mat not have been written to disk.

- Redo proceeds forward until the end of the log
  - All updates to pages in the **dirty page table** are reapplied, except for updates with **LSN** < **recLSN** or **LSN** ≤ **pageLSN**.
  - The **pageLSN** is set to the **LSN** of the log record being redone.
  - No additional log records are written during the redo phase.
  - The redo phase also reapplies the updates of **compensation log records** created during the undo phase.

Undo phase

- Undo begins at the end of the log
  - The **transaction table** identifies the transactions that were active at the time of the crash; the goal is to undo those transactions.

- Undo goes backward until the beginning of active transactions
  - Starts at the **lastLSN** of each transaction; goes backward using **prevLSN**
  - Each update is undone by reverting the page to its old contents
  - When undoing, a **compensation log record (CLR)** is written to the log
  - The **CLR** has a pointer to the next action to be undone (**undonextLSN**)
    - In the event of a crash, this allows skipping actions that have already been undone (because the redo phase redoes the **CLRs**).
    - The last **CLR** of a transaction has **undonextLSN** = NULL, which indicates that the transaction has been completely undone.
  - **CLRs** are not undone in the undo phase, but are redone in the redo phase.

ARIES: Examples

- The following examples are from the book:
  - Database Management Systems, 3rd edition: R. Ramakrishnan, J. Gehrke 2003 McGraw-Hill
- Note the following differences:
  - There is a special <T_i **end**> event that marks the end of a transaction (when it has been committed or completely rolled back)
  - The <T_i **abort**> event does not indicate when a transaction has been completely undone (this is indicated by <T_i **end**>)
  - <T_i **abort**> indicates when a transaction error occurred

ARIES: Example 1

- Analysis

transID	lastLSN
T1	10
T3	60

pageID	recLSN
P5	10
P3	20
P1	50

LSN	LOG
10	update: T1 writes P5
20	update: T2 writes P3
30	T2 commit
40	T2 end
50	update: T3 writes P1
60	update: T3 writes P3
	CRASH, RESTART

- Redo
  - LSN 10; LSN 20; LSN 50; LSN 60

- Undo
  - LSN 60; LSN 50; LSN 10
  - LSN 70: CLR Undo T3 LSN 60, undonextLSN = 50
  - LSN 80: CLR Undo T3 LSN 50, undonextLSN = NULL, T3 end
  - LSN 90: CLR Undo T1 LSN 10, undonextLSN = NULL, T1 end

ARIES: Example 2

- Analysis

transID	lastLSN
T1	70
T3	60

pageID	recLSN
P5	20
P3	30

- Redo
  - LSN 20; LSN 30; LSN 60
- Undo
  - LSN 60; LSN 20
  - LSN 80: CLR Undo T3 LSN 60, undonextLSN = NULL, T3 end
  - LSN 90: CLR Undo T1 LSN 20, undonextLSN = NULL, T1 end

LSN	LOG
00	begin_checkpoint
10	end_checkpoint
20	update: T1 writes P5
30	update: T2 writes P3
40	T2 commit
50	T2 end
60	update: T3 writes P3
70	T1 abort
	CRASH, RESTART

ARIES: Example 3

- Analysis

transID	lastLSN
T1	80
T3	90

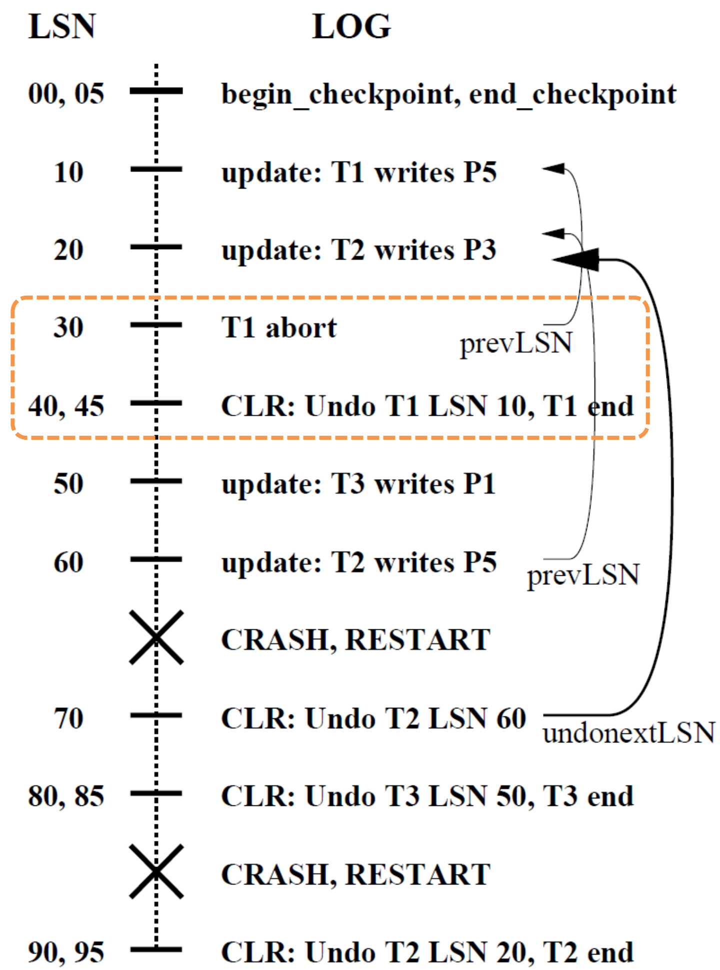
pageID	recLSN
P1	20
P2	30
P3	40
P5	80

- Redo
  - LSN 20; LSN 30; LSN 40; LSN 60; LSN 80
- Undo
  - LSN 80; LSN 60; LSN 40; LSN 20
  - LSN 100: CLR Undo T1 LSN 80, undonextLSN = 20
  - LSN 110: CLR Undo T3 LSN 60, undonextLSN = 40
  - LSN 120: CLR Undo T3 LSN 40, undonextLSN = NULL, T3 end
  - LSN 130: CLR Undo T1 LSN 20, undonextLSN = NULL, T1 end

LSN	LOG
00	begin_checkpoint
10	end_checkpoint
20	update: T1 writes P1
30	update: T2 writes P2
40	update: T3 writes P3
50	T2 commit
60	update: T3 writes P2
70	T2 end
80	update: T1 writes P5
90	T3 abort
	CRASH, RESTART

ARIES: Example 4

- LSN 30: T1 aborts
- LSN 40: CLR Undo T1 LSN 10, T1 end
- (No crash yet)



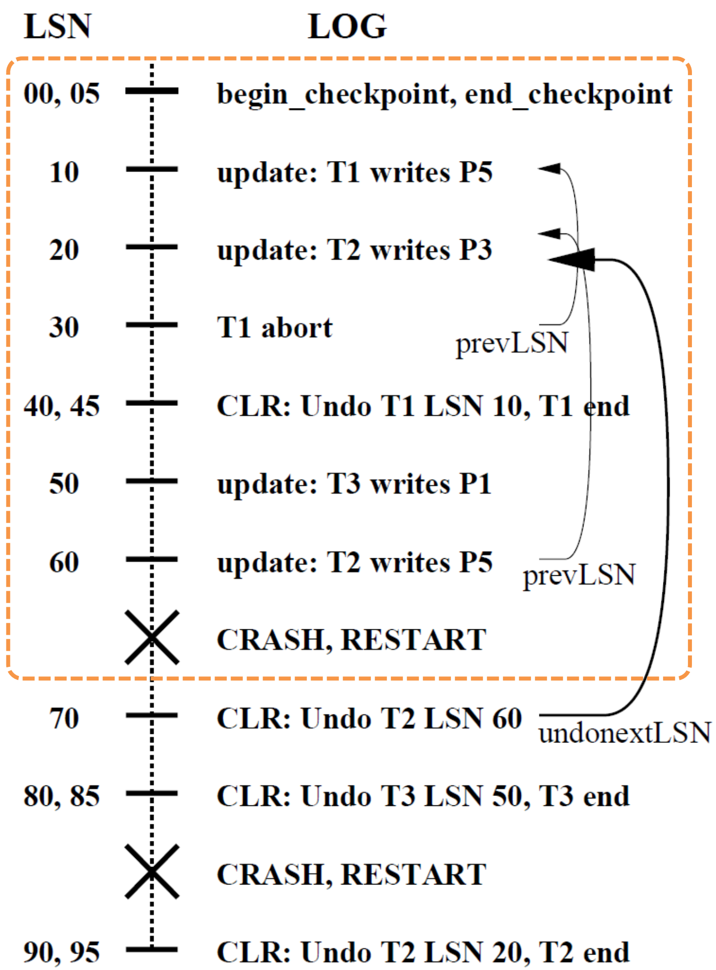
ARIES: Example 4 (Cont.)

- (First crash) Analysis

transID	lastLSN
T2	60
T3	50

pageID	recLSN
P5	10
P3	20
P1	50

- Redo
  - LSN 10; LSN 20; LSN 40 (CLR); LSN 50; LSN 60
- Undo
  - LSN 60; LSN 50; *Crash!!* ❌
  - LSN 70: CLR Undo T2 LSN 60, undonextLSN = 20
  - LSN 80: CLR Undo T3 LSN 50, undonextLSN = NULL, T3 end



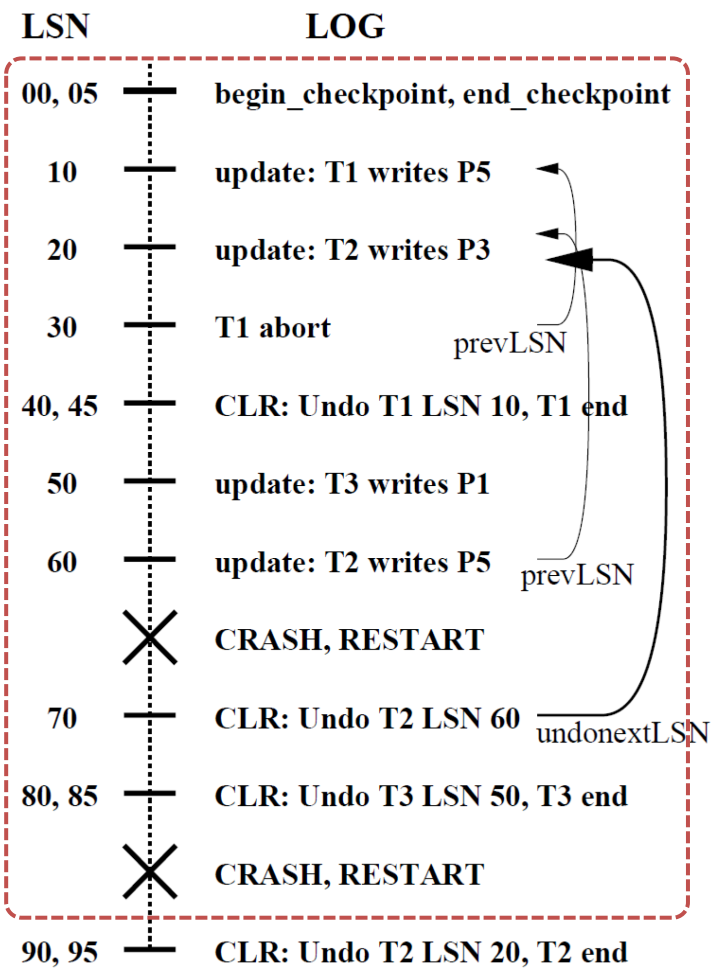
ARIES: Example 4 (Cont.)

- (Second crash) Analysis

transID	lastLSN
T2	70

pageID	recLSN
P5	10
P3	20
P1	50

- Redo
  - LSN 10; LSN 20; LSN 40 (CLR); LSN 50; LSN 60; LSN 70 (CLR); LSN 80 (CLR);
- Undo
  - (LSN 70); LSN 20
  - LSN 90: CLR Undo T2 LSN 20, undonextLSN = NULL, T2 end

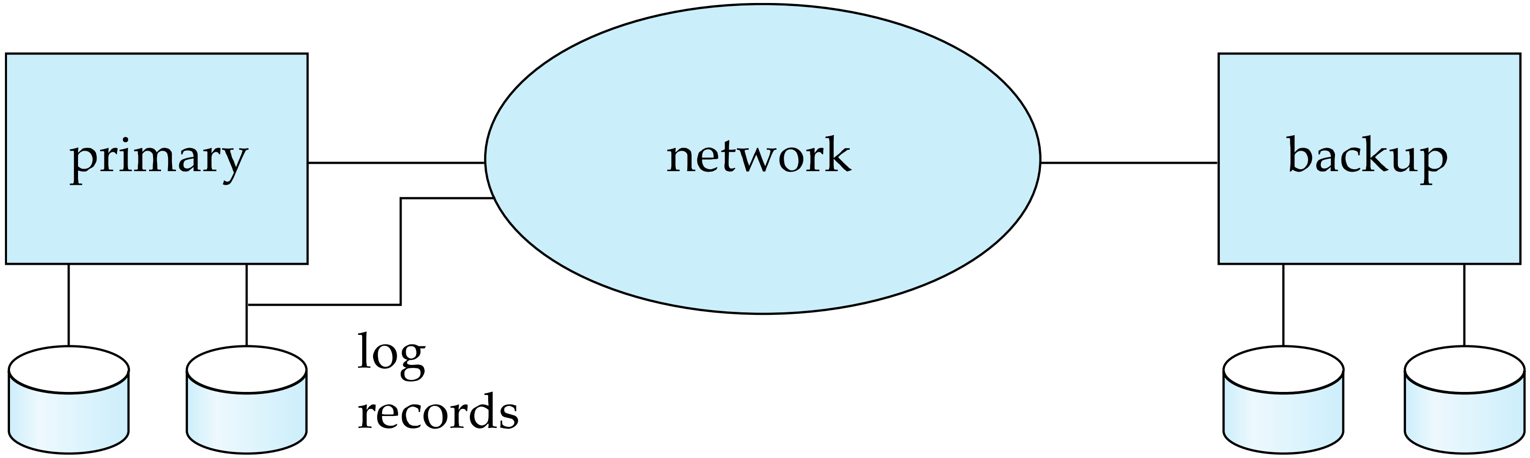


Failure with Loss of Nonvolatile Storage

- 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; a procedure similar to checkpointing must take place
    - Output all log records currently residing in main memory onto stable storage.
    - Output all buffer blocks onto the disk.
    - Copy the contents of the database to stable storage.
    - Output a record <**dump**> to log on stable storage.
- To recover from disk failure
  - restore database from most recent dump.
  - Consult the log and redo all transactions that committed after the dump

Remote Backup Systems

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



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

Remote Backup Systems (Cont.)

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

Remote Backup Systems (Cont.)

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