Advanced Databases
Recovery System

Nikolaus Augsten
nikolaus.augsten@sbg.ac.at
Department of Computer Sciences
University of Salzburg

database research group
http://dbresearch.uni-salzburg.at

WS 2018/19

Version 16. Januar 2019

Outline

Outline

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.
  - Fail-stop assumption: non-volatile storage contents are assumed to not be corrupted as result of a 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

- 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
  - 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 recover the database contents to a state that ensures atomicity, consistency and durability

Outline

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/1

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

Stable-Storage Implementation/2

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

- Copies of a block may differ due to failure during output operation. To recover from failure:
  1. First find inconsistent blocks:
     Expensive solution:
     - Compare the two copies of every disk block.

     Better solution:
     - Record in-progress disk writes on non-volatile storage (Non-volatile RAM or special area of disk).
     - Use this information during recovery to find blocks that may be inconsistent, and only compare copies of these.
     - Used in hardware RAID systems
  2. If either copy of an inconsistent block is detected to have an error (bad checksum), overwrite it by the other copy. If both have no error, but are different, overwrite the second block by the first block.

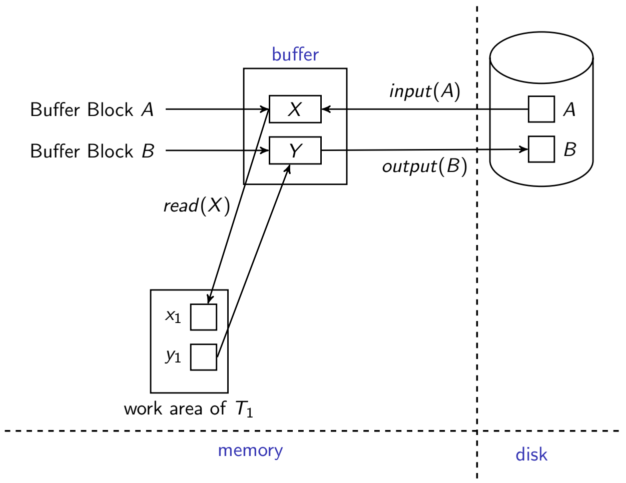
Data Access/1

- 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 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/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 denoted by x_i.
  - B_X denotes block containing X
- 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.
- Transactions
  - must perform **read**(X) before accessing X for the first time (subsequent reads can be from local copy)
  - can execute **write**(X) at any time before the transaction commits
- Note that output(B_X) need not immediately follow **write**(X). System can perform the output operation when it seems fit.

Data Access/2



Outline

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,
  - then present the actual recovery algorithm
- Less used alternative: shadow-copy and shadow-paging
- For now we assume serial execution of transactions and extend to the case of concurrent transactions later.

Log-Based Recovery

- A log is kept on stable storage.
  - The log is a sequence of log records, which maintains information about update activities on the database.
- 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_2 is the value to be written to X (the new value).
- When T_i finishes it last statement, the log record $< T_i$ **commit** $>$ is written.
- Two approaches using logs
  - Immediate database modification
  - Deferred database modification

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 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 after transaction commit
- Order in which blocks are output can be different from the order in which they are written.
- 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
- 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 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

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/1

- Undo of log record $< T_i, X, V_1, V_2 >$ writes the old value V_1 to X
- Redo of 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 (called redo-only) $< T_i, X, V >$ is written out
    - 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, going forward from the first log record for T_i
    - No logging is done in this case

Undo and Redo Operations/2

- The undo and redo operations are used in several different circumstances:
  - The undo is used for transaction rollback during normal operation (when a transaction must abort due to some logical error).
  - The undo and redo operations are used during recovery from failure.
- We need to deal with the case where during recovery from failure another failure occurs prior to the system having fully recovered.

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 $< T_i, X_j, V_1, V_2 >$
  - perform the undo by writing V_1 to X_j,
  - write a redo-only log record $< T_i, X_j, V_1 >$ (also called compensation log record)
- Once the record $< T_i$ **start** $>$ is found stop the scan and write the log record $< T_i$ **abort** $>$

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** $>$
- Repeating history:
  - Recovery redoes all the original actions including the steps that restored old values (redo-only log records).
  - It may seem strange to redo transaction T_i if the record $< T_i$ **abort** $>$ record is in the log. To see why this works, note that if $< T_i$ **abort** $>$ is in the log, so are the redo-only records written by the undo operation. Thus, the end result will be to undo T_i's modifications in this case. This slight redundancy simplifies the recovery algorithm and enables faster overall recovery time.

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** $>$

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.

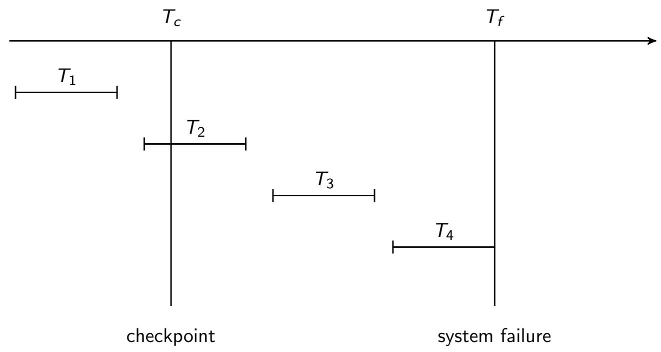
Checkpoints/1

- 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
- All updates are stopped while doing 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.

Checkpoints/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.

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

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 single log
  - 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 can modify the same item until T_i has committed or aborted
  - i.e. the updates of uncommitted transactions should not be visible to other transactions
    - Otherwise how do we perform undo if T_1 updates A, then T_2 updates A and commits, and finally T_1 has to abort?
  - Can be ensured by obtaining exclusive locks on updated items and holding the locks till end of transaction (strict two-phase locking)
- Log records of different transactions may be interspersed in the log.

Outline

Recovery Algorithm/1

- 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 $< T_i, X_j, V_1, V_2 >$
    - 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** $>$

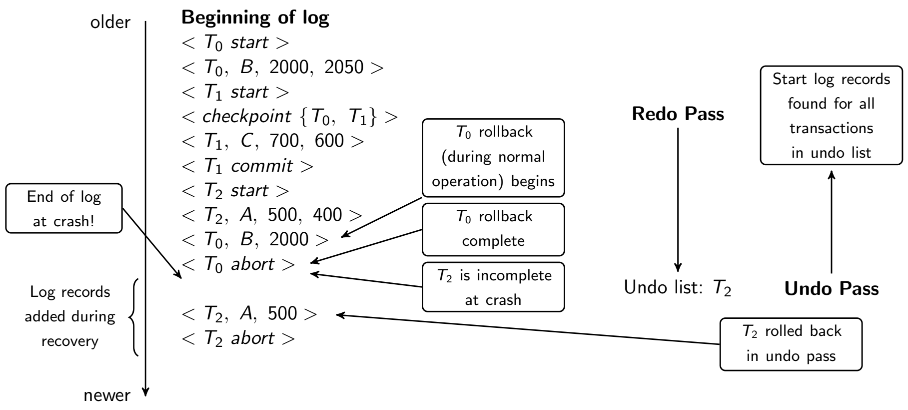
Recovery Algorithm/2

- 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 Ti to undo-list
     3. Whenever a log record $< T_i$ **commit** $>$ or $< T_i$ **abort** $>$ is found, remove T_i from undo-list
- After redo: database is in the same state as at time of crash

Recovery Algorithm/2

- Undo phase: 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
     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/1

- Log record buffering: log records are buffered in main memory, instead of 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.

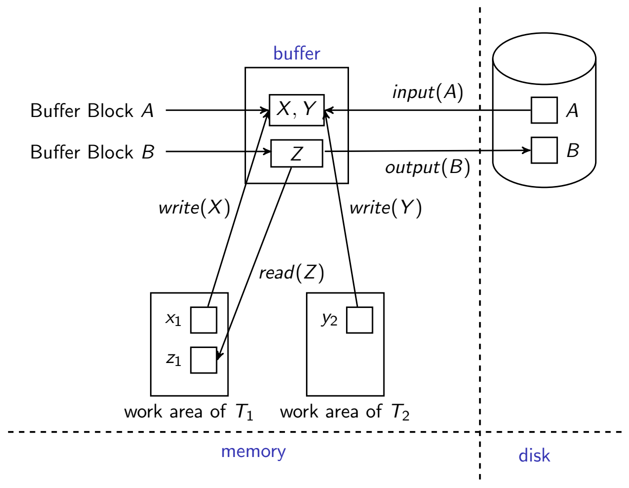
Log Record Buffering/2

- 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 or WAL rule
    - Strictly speaking WAL only requires undo information to be output

Database Buffering/1

- 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
- The recovery algorithm supports the no-force policy: i.e., updated blocks need not be written to disk when transaction commits
  - force policy: requires updated blocks to be written at commit
    - More expensive commit
- The recovery algorithm supports the steal policy: i.e., blocks containing updates of uncommitted transactions can be written to disk, even before the transaction commits

Database Buffering/2



- Both T_1 and T_2 write a data item (X resp. Y) on block A

Database Buffering/3

- If a block with uncommitted updates is output to disk, log records with undo information for the updates are output to the log on stable storage first
  - (Write ahead logging)
- No updates should be in progress on a block when it is output to disk. Can be ensured as follows.
  - Before writing a data item, transaction acquires exclusive lock on block containing the data item
  - Lock can be released once the write is completed.
    - Such locks held for short duration are called latches.
- 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/1

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

Buffer Management/2

- Database buffers are generally implemented in virtual memory in spite of some drawbacks:
  - When operating system needs to evict a page that has been modified, the page is written to swap space on disk.
  - When database decides to write buffer page to disk, buffer page may be in swap space, and may have to be read from swap space on disk and output to the database on disk, resulting in extra I/O!
    - Known as dual paging problem.
  - Ideally when OS needs to evict a page from the buffer, it should pass control to database, which in turn should
    1. Output the page to database instead of to swap space (making sure to output log records first), if it is modified
    2. Release the page from the buffer, for the OS to use

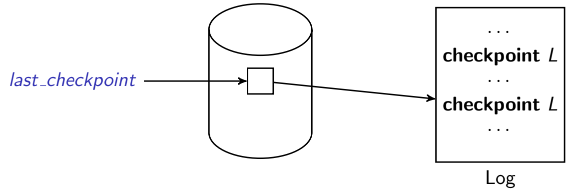
    Dual paging can thus be avoided, but common operating systems do not support such functionality.

Fuzzy Checkpointing/1

- To avoid long interruption of normal processing during checkpointing, allow updates to happen 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 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

- 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



Disk Crash

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

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.

Failure with Loss of Nonvolatile 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

Outline

Recovery with Early Lock Release

- Support for high-concurrency locking techniques, such as those used for B^+-*tree* concurrency control, which release locks early
  - Supports "logical undo"
- Recovery based on "repeating history", whereby recovery executes exactly the same actions as normal processing

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).
- For such operations, undo log records should contain the undo operation to be executed
  - Such logging is called logical undo logging, in contrast to physical undo logging
    - Operations are called logical operations
  - Other examples:
    - delete of tuple, to undo insert of tuple (allows early lock release on space allocation information)
    - subtract amount deposited, to undo deposit (allows early lock release on bank balance)

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

- Operation logging is done as follows:
  1. When operation starts, log $< T_i,\ O_j,\ \textit{operation-begin} >$. Here 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, $< T_i,\ O_j,\ \textit{operation-end},\ U >$ is logged, where U contains information needed to perform a logical undo.

  Example: insert of $(key,\ record\text{-}id)$ pair $(K5,\ RID7)$ into index $I9$

$$
\begin{array}{l}
< T_1,\ O_1,\ \textit{operation-begin} > \\
\ldots \\
\left.\begin{array}{l} < T_1,\ X,\ 10,\ K5 \\ < T_1,\ Y,\ 45,\ RID7 \end{array}\right\}\ \text{Physical redo of steps in insert} \\
< T_1,\ O_1,\ \textit{operation-end},\ (\textit{delete}\ I9,\ K5,\ RID7) >
\end{array}
$$

Operation Logging/2

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

Transaction Rollback with Logical Undo/1

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

Transaction Rollback with Logical Undo/2

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

Some points to note:

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

Transaction Rollback with Logical Undo

- Transaction rollback during normal operation

older ↓ newer

Beginning of log

$< T_0\ start >$
$< T_0,\ B,\ 2000,\ 2050 >$
$< T_0,\ O_1,\ operation\text{-}begin >$ ← If T_0 aborts before operation O_1 ends, undo of update to C will be physical
$< T_0,\ C,\ 700,\ 600 >$
$< T_0,\ O_1,\ operation\text{-}end,\ (C,+100) >$ ← T_0 has complete operation O_1 on C, release lower-level lock; physical undo cannot be done anymore, logical undo will add 100 to C
$< T_1\ start >$
$< T_1,\ O_2,\ operation\text{-}begin >$
$< T_1,\ C,\ 600,\ 400 >$ ← T_1 can update C since T_0 has released lower-level lock on C
$< T_1,\ O_2,\ operation\text{-}end,\ (C,+200) >$ ← T_1 release lower-level lock on C

T_0 decides to abort →

$< T_0,\ C,\ 400,\ 500 >$ ← Logical undo of O_1 adds 100 to C
$< T_0,\ operation\text{-}abort >$ ← O_1 undo complete
$< T_0,\ B,\ 2000 >$
$< T_0\ abort >$
$< T_1\ commit >$

Failure Recovery with Logical Undo

older ↓ newer

Beginning of log

$< T_0\ start >$
$< T_0,\ B,\ 2000,\ 2050 >$
$< T_0\ commit >$
$< T_1\ start >$
$< T_1,\ B,\ 2050,\ 2100 >$
$< T_1,\ O_4,\ operation\text{-}begin >$
$< checkpoint\ \{T_1\} >$
$< T_1,\ C,\ 700,\ 400 >$
$< T_1,\ O_4,\ operation\text{-}end,\ (C,+300) >$
$< T_2\ start >$
$< T_2,\ O_5,\ operation\text{-}begin >$
$< T_2,\ C,\ 400,\ 300 >$

End of log at crash! →

Records added during recovery:

$< T_2,\ C,\ 400 >$ ← Update of C was part of O_5, undone physically during recovery since O_5 did not complete
$< T_2\ abort >$
$< T_1,\ C,\ 400,\ 700 >$ ← Logical undo of O_4 adds 300 to C
$< T_1,\ O_4,\ operation\text{-}abort >$
$< T_1,\ B,\ 2050 >$
$< T_1\ abort >$

Redo Pass ↓ Undo list: T_1, T_2

Undo Pass ↑ Start log records found for all transactions in undo list

Transaction Rollback: Another Example

- Example with a complete and an incomplete operation

$< T_1\ start >$
$< T_1,\ O_1,\ operation\text{-}begin >$
...
$< T_1,\ X,\ 10,\ K5 >$
$< T_1,\ Y,\ 45,\ RID7 >$
$< T_1,\ O_1,\ operation\text{-}end,\ (delete\ I9,\ K5,\ RID7) >$
$< T_1,\ O_2,\ operation\text{-}begin >$
$< T_1,\ Z,\ 45,\ 70 >$
← T_1 Rollback begins here
$< T_1,\ Z,\ 45 >$ ← redo-only log record during physical undo (of incomplete O_2)
$< T_1,\ Y,\ \ldots,\ \ldots >$ ← Normal redo records for logical undo of O_1
...
$< T_1,\ O_1,\ operation\text{-}abort >$ ← What if crash occurred immediately after this?
$< T_1\ abort >$

Recovery Algorithm with Logical Undo/1

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

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 Algorithm with Logical Undo/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.

Outline

ARIES

- ARIES is a state of the art recovery method
  - Incorporates numerous optimizations to reduce overheads during normal processing and to speed up recovery
  - The recovery algorithm we studied earlier is modeled after ARIES, but greatly simplified by removing optimizations
- 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 applied to a database page
  2. Physiological redo
  3. Dirty page table to avoid unnecessary redos during recovery
  4. Fuzzy checkpointing that only records information about dirty pages, and does not require dirty pages to be written out at checkpoint time
     - 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 moved to fill hole
      - Physiological redo can log just the record deletion
      - Physical redo would require logging of old and new values for much of the page
  - Requires page to be output to disk atomically
    - Easy to achieve with hardware RAID, also supported by some disk systems
    - Incomplete page output can be detected by checksum techniques,
      - But extra actions are required for recovery
      - Treated as a media failure

ARIES Data Structures

- 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

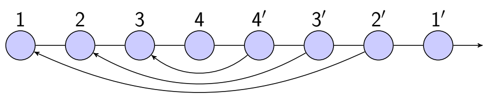
ARIES Data Structures: Log Record

- 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

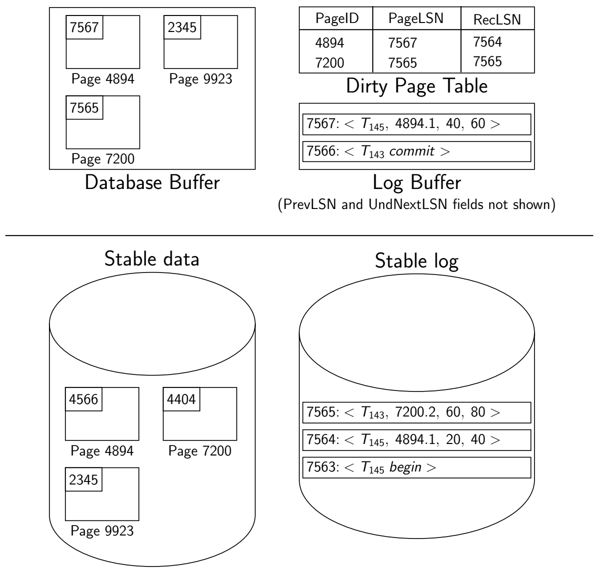
LSN	TransID	UndoNextLSN	RedoInfo



ARIES Data Structures: DirtyPage Table

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

ARIES Data Structures



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 written by the transaction
  - Fixed position on disk notes LSN of last completed checkpoint log record
- Dirty pages are not written out at checkpoint time
  - Instead, they are flushed out continuously, in the background
- Checkpoint is thus very low overhead
  - can be done frequently

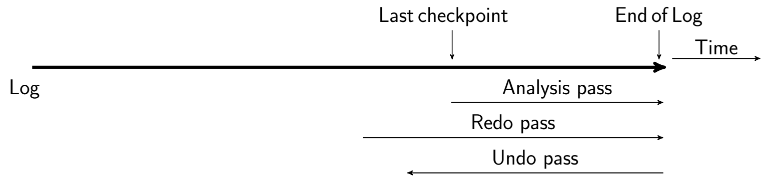
ARIES Recovery Algorithm

ARIES recovery involves three passes

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

Aries Recovery: 3 Passes

- 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/1

Analysis pass

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

ARIES Recovery: Analysis/2

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

ARIES Redo Pass

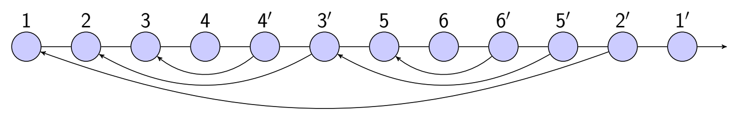
Redo Pass: Repeats history by replaying every action not already reflected in the page on disk, as follows:

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

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

ARIES Undo Actions

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

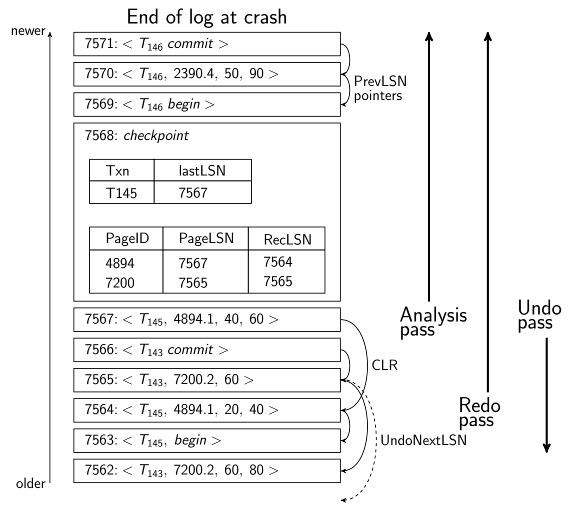


ARIES: Undo Pass

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

Recovery Actions in ARIES



Other ARIES Features/1

- 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

- 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