



# Chapter 16: Recovery System

Database System Concepts, 6<sup>th</sup> Ed.  
 ©Silberschatz, Korth and Sudarshan  
 See [www.db-book.com](http://www.db-book.com) for conditions on re-use

---

---

---

---

---

---

---

---

---

---



## Outline

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

---

---

---

---

---

---

---

---

---

---



## 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

---

---

---

---

---

---

---

---

---

---



## 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

---

---

---

---

---

---

---

---

---

---



## 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  $\langle T_i \text{ start} \rangle$  to the log
- Before  $T_i$  executes **write**( $X$ ), a log record  $\langle T_i, X, V_1, V_2 \rangle$  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 its last statement, the log record  $\langle T_i \text{ commit} \rangle$  is written.
- Two approaches using logs
  - Immediate database modification
  - Deferred database modification

---

---

---

---

---

---

---

---

---

---



## 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

---

---

---

---

---

---

---

---

---

---



## Undo and Redo Operations

- **Undo** of a log record  $\langle T_i, X, V_1, V_2 \rangle$  writes the **old** value  $V_1$  to  $X$
- **Redo** of a log record  $\langle T_i, X, V_1, V_2 \rangle$  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**)  $\langle T_i, X, V \rangle$  is written out
    - ▶ When undo of a transaction is complete, a log record  $\langle T_i, \text{abort} \rangle$  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 (Cont.)

- The **undo** and **redo** operations are used in several different circumstances:
  - The **undo** is used for transaction rollback during normal operation (in case a transaction cannot complete its execution 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.

---

---

---

---

---

---

---

---

---

---

---

---



## Undo and Redo on Recovering from Failure

- When recovering after failure:
  - Transaction  $T_i$  needs to be undone if the log
    - ▶ contains the record  $\langle T_i, \text{start} \rangle$ ,
    - ▶ but does not contain either the record  $\langle T_i, \text{commit} \rangle$  or  $\langle T_i, \text{abort} \rangle$ .
  - Transaction  $T_i$  needs to be redone if the log
    - ▶ contains the records  $\langle T_i, \text{start} \rangle$
    - ▶ and contains the record  $\langle T_i, \text{commit} \rangle$  or  $\langle T_i, \text{abort} \rangle$
  - It may seem strange to redo transaction  $T_i$  if the record  $\langle T_i, \text{abort} \rangle$  record is in the log. To see why this works, note that if  $\langle T_i, \text{abort} \rangle$  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.
    - ▶ such a redo redoes all the original actions including the steps that restored old value. Known as **repeating history**

---

---

---

---

---

---

---

---

---

---

---

---



## 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**
- 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 (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<sub>i</sub> start>** is found for every transaction  $T_i$  in *L*.
  - Parts of log prior to earliest **<T<sub>i</sub> start>** record above are not needed for recovery, and can be erased whenever desired.

---

---

---

---

---

---

---

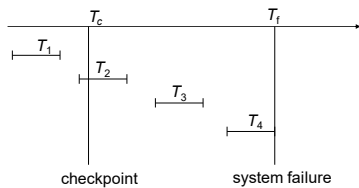
---

---

---



## 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 Algorithm

- **Logging** (during normal operation):
  - $\langle T_i \text{ start} \rangle$  at transaction start
  - $\langle T_i, X_j, V_1, V_2 \rangle$  for each update, and
  - $\langle T_i \text{ commit} \rangle$  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  $\langle T_i, X_j, V_1, V_2 \rangle$ 
    - ▶ perform the undo by writing  $V_1$  to  $X_j$
    - ▶ write a log record  $\langle T_i, X_j, V_1 \rangle$ 
      - such log records are called **compensation log records**
  - Once the record  $\langle T_i \text{ start} \rangle$  is found stop the scan and write the log record  $\langle T_i \text{ abort} \rangle$

---

---

---

---

---

---

---

---

---

---



## 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  $\langle \text{checkpoint } L \rangle$  record, and set undo-list to  $L$ .
  2. Scan forward from above  $\langle \text{checkpoint } L \rangle$  record
    1. Whenever a record  $\langle T_i, X_j, V_1, V_2 \rangle$  is found, redo it by writing  $V_2$  to  $X_j$
    2. Whenever a log record  $\langle T_i \text{ start} \rangle$  is found, add  $T_i$  to undo-list
    3. Whenever a log record  $\langle T_i \text{ commit} \rangle$  or  $\langle T_i \text{ abort} \rangle$  is found, remove  $T_i$  from undo-list

---

---

---

---

---

---

---

---

---

---



## Recovery Algorithm (Cont.)

- **Undo phase:**
  1. Scan log backwards from end
    1. Whenever a log record  $\langle T_i, X_j, V_1, V_2 \rangle$  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  $\langle T_i, X_j, V_1 \rangle$
    2. Whenever a log record  $\langle T_i \text{ start} \rangle$  is found where  $T_i$  is in undo-list,
      1. Write a log record  $\langle T_i \text{ abort} \rangle$
      2. Remove  $T_i$  from undo-list
    3. Stop when undo-list is empty
      - i.e.,  $\langle T_i \text{ start} \rangle$  has been found for every transaction in undo-list
- After undo phase completes, normal transaction processing can commence

---

---

---

---

---

---

---

---

---

---



## 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  $\langle T_i, \text{commit} \rangle$  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

- 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 (Cont.)

- 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

Database System Concepts - 6th Edition

16.19

©Silberschatz, Korth and Sudarshan

---

---

---

---

---

---

---

---

---

---



## Fuzzy Checkpointing

- 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

Database System Concepts - 6th Edition

16.20

©Silberschatz, Korth and Sudarshan

---

---

---

---

---

---

---

---

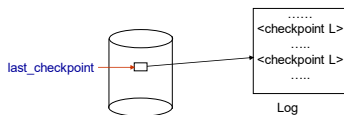
---

---



## Fuzzy Checkpointing (Cont.)

- 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



Database System Concepts - 6th Edition

16.21

©Silberschatz, Korth and Sudarshan

---

---

---

---

---

---

---

---

---

---



## ARIES Recovery Algorithm

---

---

---

---

---

---

---

---

---

---



## 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
- 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. Dirty page table to avoid unnecessary redos during recovery
  3. 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 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
  - Flushed LSN
    - ▶ Last log entry flushed to disk
  - 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 effects are reflected on the page
  - To update a page:
    - ▶ X-latch the page, and write the log record
    - ▶ Update the page
    - ▶ Record the LSN of the log record in PageLSN
    - ▶ Unlock page
  - To flush page to disk, must first S-latch page
    - ▶ Thus page state on disk is operation consistent
      - Required to support physiological redo
  - PageLSN is used during recovery to prevent repeated redo
    - ▶ Thus ensuring idempotence

---

---

---

---

---

---

---

---

---

---

---

---

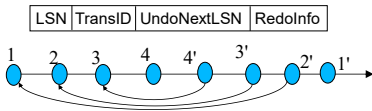


## ARIES Data Structures: Log Record

- Each log record contains LSN of previous log record of the same transaction

LSN	TransID	PrevLSN	RedoInfo	UndoInfo
-----	---------	---------	----------	----------

- 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




---

---

---

---

---

---

---

---

---

---

---

---



## 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 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

---

---

---

---

---

---

---

---

---

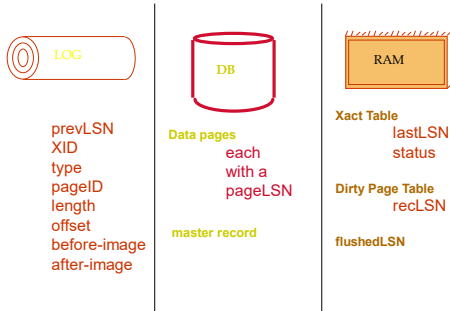
---

---

---



## The Big Picture: What's Stored Where



Database System Concepts - 6th Edition

16.28

©Silberschatz, Korth and Sudarshan

---

---

---

---

---

---

---

---

---

---

---

---



## 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

Database System Concepts - 6th Edition

16.29

©Silberschatz, Korth and Sudarshan

---

---

---

---

---

---

---

---

---

---

---

---



## 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

Database System Concepts - 6th Edition

16.30

©Silberschatz, Korth and Sudarshan

---

---

---

---

---

---

---

---

---

---

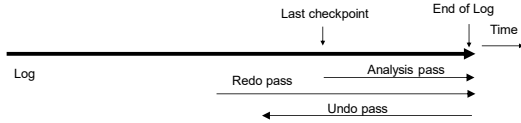
---

---



## 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 Xact table = list of transactions in checkpoint log record
  - Reads LSN of last log record for each transaction in undo-list from checkpoint log record
- Scans forward from checkpoint
- .. Cont. on next page ...

---

---

---

---

---

---

---

---



## ARIES Recovery: Analysis (Cont.)

### Analysis pass (cont.)

- Scans forward from checkpoint
  - If any log record found for transaction not in Xact table, adds transaction to table
  - 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 Xact table
  - Keeps track of last log record for each transaction in Xact table
    - 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 Xact table 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!

---

---

---

---

---

---

---

---

---

---

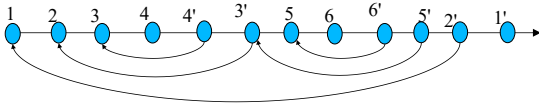
---

---



## 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 physically or physiologically).
    - ▶ CLR for record  $n$  noted as  $n'$  in figure below
  - Set UndoNextLSN of the CLR to the PrevLSN value of the update log record
    - ▶ Arrows indicate UndoNextLSN value




---

---

---

---

---

---

---

---

---

---

---

---



## ARIES: 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 undone 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




---

---

---

---

---

---

---

---



## End of Chapter 16

---

---

---

---

---

---

---

---