Lecture 9: ARIES from First Principles

CREATING THE NEXT®

Today's Agenda

Recap

Mains ideas of ARIES

- Mains ideas of ARIES:
    - WAL with STEAL/NO-FORCE
    - Fuzzy Checkpoints (snapshot of dirty page ids)
    - Redo everything since the earliest dirty page
    - Undo txns that never commit
    - Write CLRs when undoing, to survive failures during restarts

Mains ideas of ARIES

- **Buffer Manager**
  - PinPage, UnpinPage, ReadPage, WritePage, DirtyPageTable
- **Recovery Manager**
  - Restart, RecoverEarliestLSN, CreateLogRecord, RollbackTxn
- **Log Manager**
  - ReadNextLogRecord, AppendLogRecord, GetMasterRecord, SetMasterRecord
- **Txn Manager**
  - GetRecordInfo, SetRecordInfo, ActiveTxnTable
- **Disk Manager**
  - ReadBlock, WriteBlock

Today's Agenda

- Deriving ARIES from first principles
  - V1: Shadow Paging
  - V2: WAL–Deferred Updates
  - V3: WAL
  - V4: Commit-consistent checkpoints
  - V5: Fuzzy checkpoints
  - V6: CLRs
  - V7: Logical Undo
  - V8: Avoid selective redo

Definitions

Protocol vs Algorithm

- Protocol
  - Set of rules that govern how a system operates.
  - Rules establish the basic functioning of the different parts, how they interact with each other, and what constraints must be satisfied by the implementation.
- Algorithm
  - Set of instructions to transform inputs to desired outputs. It can be a simple script, or a complicated program. The order of the instructions is important.

Protocol vs Algorithm

- Protocol
  - Logging and recovery protocol dictates how the buffer manager interacts with the recovery manager to ensure the durability of changes made by committed txns.
- Algorithm
  - A sorting algorithm may return the records in a table in alphabetical order.

Policy vs Mechanism

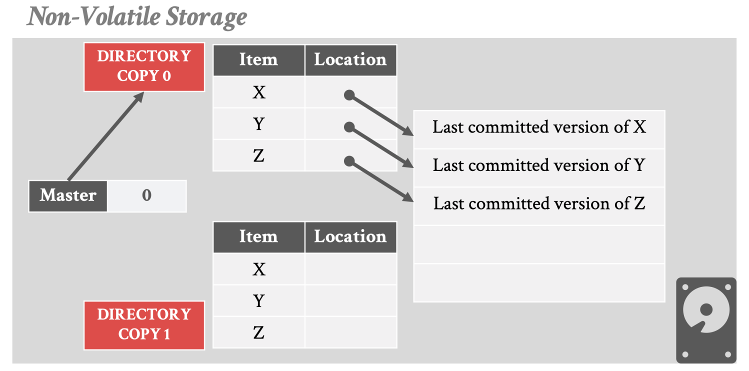
- Policy
  - Specifies the desired behavior of the system (**what**).
  - Example: Buffer manager may adopt the LRU policy for evicting pages from the buffer.
- Mechanism
  - Specifies how that behavior must be realized (**how**)
  - Example: We may implement the policy using: (1) uni-directional map + linked list, or (2) bi-directional map. Optimize the code for specific hardware technology.

Deriving ARIES

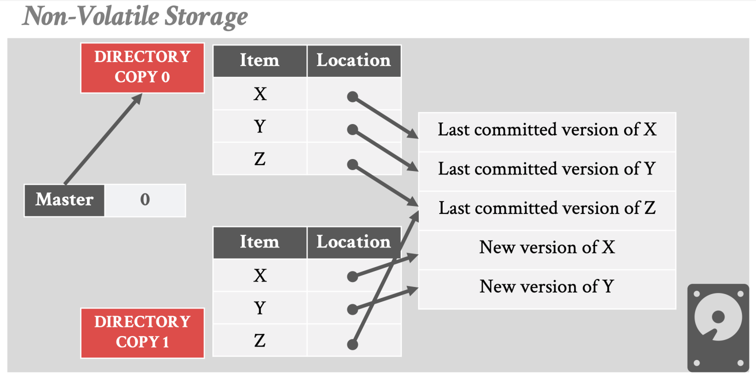
Constraints

- DRAM is volatile

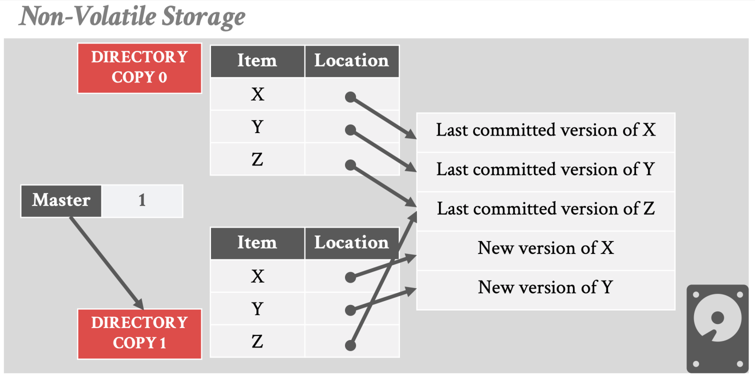
V1: SHADOW PAGING



V1: SHADOW PAGING



V1: SHADOW PAGING



V1: SHADOW PAGING

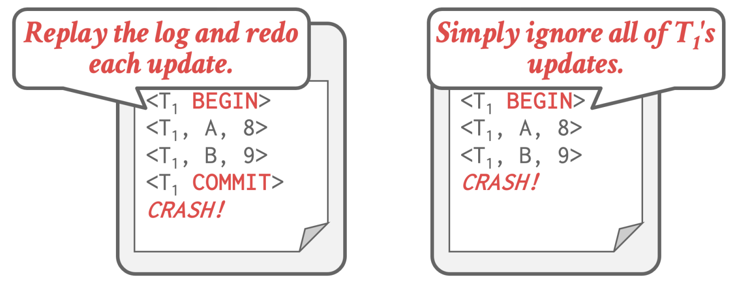
- Advantages
  - No need to write log records
  - Recovery is trivial (**NO UNDO** and **NO REDO**)
- Disadvantages
  - Commit overhead is high (**FORCE** and **NO STEAL**)
  - Flush every updated page to database on disk, page table, and master page
  - Data gets fragmented over time (versioning)
  - Need garbage collection to clean up older versions.
  - Need to copy page table

Constraints

- DRAM is volatile
- Avoid random writes to database on disk (NO FORCE)

WAL – Deferred Updates

- If we prevent the DBMS from writing dirty records to disk until the txn commits, then the DBMS does not need to store their original values.



Replay the log and redo each update.

<T_1 BEGIN>
<T_1, A, 8>
<T_1, B, 9>
<T_1 COMMIT>
CRASH!

Simply ignore all of T_1's updates.

<T_1 BEGIN>
<T_1, A, 8>
<T_1, B, 9>
CRASH!

V2: WAL–DEFERRED UPDATES

- **Phase 1 – Analysis**
  - Read the WAL to identify active txns at the time of the crash.
- **Phase 2 – Redo**
  - Start with the last entry in the log and scan backwards toward the beginning.
  - For each update log record with a given LSN, redo the action if:
  - pageLSN (on disk) < log record's LSN

V2: WAL–DEFERRED UPDATES

LSN Type	Where	Definition
flushedLSN	Memory	Last LSN in log on disk
pageLSN	$page_x$	Newest update to $page_x$
prevLSN	log record	LSN of prior log record by same txn

V2: WAL–DEFERRED UPDATES

- **PageLSN** (on disk – page)
  - Determine whether the log record's update needs to be re-applied to the page.
- **PrevLSN** (on disk – log record)
  - Log records of multiple transactions will be interleaved on disk
  - PrevLSN helps quickly locate the predecessor of a log record of a particular transaction
  - Facilitates parallel transaction-oriented undo

V2: WAL–DEFERRED UPDATES

- Advantages
  - No need to undo changes (**NO UNDO** + **REDO**)
  - Flush updated pages to log on disk with sequential writes
  - Commit overhead is reduced since random writes to database are removed from the transaction commit path
- Disadvantages
  - Buffer manager cannot replace a dirty slot last written by an uncommitted transaction. (**NO FORCE** & **NO STEAL**)
  - Cannot support transactions with change sets larger than the amount of memory available

Constraints

- DRAM is volatile
- Avoid random writes to database on disk (NO FORCE)
- Support transactions with change sets > DRAM (STEAL)

V3: WAL

- **Phase 1 – Analysis**
  - Read the WAL to identify dirty pages in the buffer pool and active txns at the time of the crash.
- **Phase 2 – Redo**
  - Repeat all actions starting from an appropriate point in the log.
- **Phase 3 – Undo**
  - Reverse the actions of txns that did not commit before the crash.

V3: WAL

LSN Type	Where	Definition
flushedLSN	Memory	Last LSN in log on disk
pageLSN	$page_x$	Newest update to $page_x$
prevLSN	log record	LSN of prior log record by same txn
recLSN	DPT	Oldest update to $page_x$ since it was last flushed
lastLSN	ATT	Latest action of txn T_i

V3: WAL

- **RecLSN** (in memory – Dirty Page Table)
  - Determine whether page state has not made it to disk.
  - If there is a suspicion, then page has to accessed.
  - Serves to limit the number of pages whose PageLSN has to be examined
  - If a file sync operation is found in the log, all the pages in the file are removed from the dirty page table
- **LastLSN** (in memory – Active Transaction Table)
  - Determine log records which have to rolled back for the yet-to-be-completely-undone uncommitted transactions

V3: WAL

- Advantages
  - Maximum flexibility for buffer manager
- Disadvantages
  - Log will keep growing over time thereby slowing down recovery and taking up more storage space.

Constraints

- DRAM is volatile
- Avoid random writes to database on disk (NO FORCE)
- Support transactions with change sets > DRAM (STEAL)
- Recovery time must be bounded.

V4: COMMIT-CONSISTENT CHECKPOINTS

LSN Type	**Where**	**Definition**
flushedLSN	Memory	Last LSN in log on disk
pageLSN	$page_x$	Newest update to $page_x$
prevLSN	log record	LSN of prior log record by same txn
recLSN	DPT	Oldest update to $page_x$ since it was last flushed
lastLSN	ATT	Latest action of txn T_i
MasterRecord	Disk	LSN of latest checkpoint

V4: COMMIT-CONSISTENT CHECKPOINTS

- **Phase 1 – Analysis**
  - Read the WAL starting from the **latest checkpoint**.
- **Phase 2 – Redo**
  - Repeat all actions starting from an appropriate point in the log.
- **Phase 3 – Undo**
  - Reverse the actions of txns that did not commit before the crash.

V4: COMMIT-CONSISTENT CHECKPOINTS

- Advantages
  - Recovery time is bounded due to checkpoints.
- Disadvantages
  - With commit consistent checkpointing, DBMS must stop processing transactions while taking checkpoint
  - Users will suffer long delays due to checkpointing

Constraints

- DRAM is volatile
- Avoid random writes to database on disk (NO FORCE)
- Support transactions with change sets > DRAM (STEAL)
- Recovery time must be bounded.
- Users must not suffer long delays due to checkpointing.

V5: FUZZY CHECKPOINTS

- Instead of flushing **all** dirty pages, only flush those dirty pages that have not been flushed since before the **previous checkpoint**.
- This guarantees that, at any time, all updates of committed transactions that occurred before the **penultimate** (*i.e.*, second to last) checkpoint have been applied to database on disk - during the last checkpoint, if not earlier.

V5: FUZZY CHECKPOINTS

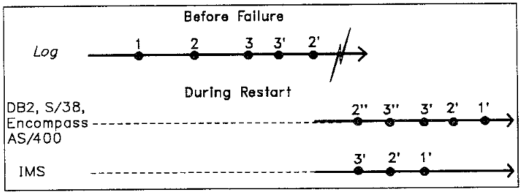
- Advantages
  - With fuzzy checkpointing, DBMS can concurrently process transactions while taking checkpoints.
- Problem
  - Repeated failures during recovery can lead to unbounded amount of logging during recovery

Constraints

- DRAM is volatile
- Avoid random writes to database on disk (NO FORCE)
- Support transactions with change sets > DRAM (STEAL)
- Recovery time must be bounded.
- Users must not suffer long delays due to checkpointing.
- Cope with failures during recovery.

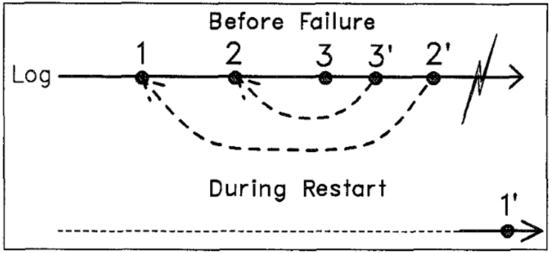
V6: COMPENSATION LOG RECORDS

- Problems: (1) compensating compensations and (2) duplicate compensations



I′ is the CLR for I and I″ is the CLR for I′

V6: COMPENSATION LOG RECORDS



I′ is the Compensation Log Record for I
I′ points to the predecessor, if any, of I

V6: COMPENSATION LOG RECORDS

LSN Type	**Where**	**Definition**
flushedLSN	Memory	Last LSN in log on disk
pageLSN	$page_x$	Newest update to $page_x$
prevLSN	log record	LSN of prior log record by same txn
recLSN	DPT	Oldest update to $page_x$ since it was last flushed
lastLSN	ATT	Latest action of txn T_i
MasterRecord	Disk	LSN of latest checkpoint
undoNextLSN	log record	LSN of prior to-be-undone record

Constraints

- DRAM is volatile
- Avoid random writes to database on disk (NO FORCE)
- Support transactions with change sets > DRAM (STEAL)
- Recovery time must be bounded.
- Users must not suffer long delays due to checkpointing.
- Cope with repeated failures during recovery.
- Increase concurrency of undo.

V7: LOGICAL UNDO

- Record logical operations to be undone instead of physical offsets
    - Undo action need not be exact physical inverse of original action (*i.e.*, page offsets need not be recorded)
    - Example: Insert key X in B+tree
    - X can be initially inserted in Page 10 by T_1
    - X may be moved to Page 20 by another txn T_2 before T_1 commits
    - Later, if T_1 is aborted, logical undo (Delete key X in B+tree) will automatically remove it from Page 20

V7: LOGICAL UNDO

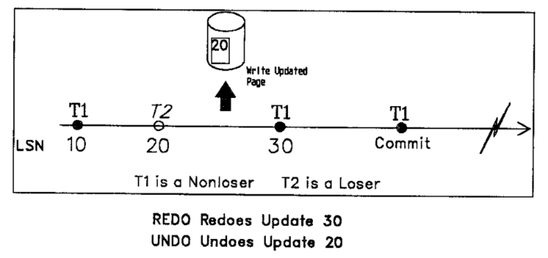
- Logical undo enables:
    - Highly-parallel **transaction-oriented logical undo**
    - Works with fast **page-oriented physical redo**
    - Hence, this protocol performs **physiological logging**
- Record logical ops for index and space management (*i.e.*, garbage collection)
    - Avoid rebuilding indexes from scratch during recovery
    - Reclaim storage space of deleted records
    - Example: Put in slot 5 (instead of Put at offset 30)

Constraints

- DRAM is volatile
- Avoid random writes to database on disk (NO FORCE)
- Support transactions with change sets > DRAM (STEAL)
- Recovery time must be bounded.
- Users must not suffer long delays due to checkpointing.
- Cope with repeated failures during recovery.
- Increase concurrency of undo (logical undo).
- Support record-level locking

V8: AVOID SELECTIVE REDO

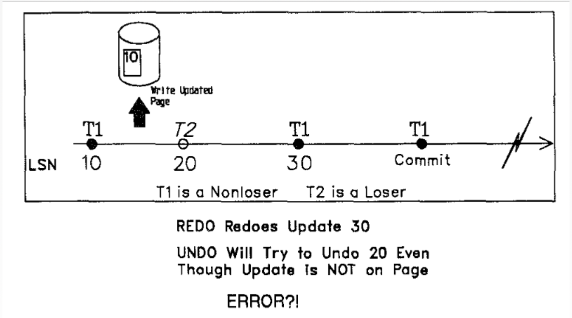
- Problem-free scenario



REDO Redoes Update 30
UNDO Undoes Update 20

V8: AVOID SELECTIVE REDO

- Problematic scenario: **UNDOing non-existent changes**



REDO Redoes Update 30

UNDO Will Try to Undo 20 Even Though Update Is NOT on Page

ERROR?!

V8: AVOID SELECTIVE REDO

- Problematic scenario:
  - Does not work with logical undo
  - Example: Consider a B+tree index with non-unique keys
  - T_1 inserted key X in Page 10 and committed
  - T_2 inserted key X in Page 10 and is not committed
  - T_3 inserted key Y in Page 10 and committed
  - Only T_1's changes make it to disk
  - While redoing T_3, we push the LSN forward
  - We must undo T_2 (since pageLSN > T_2's log record's LSN)
  - Executing Delete key X will incorrectly remove T_1's changes

V8: AVOID SELECTIVE REDO

- Solution:
  - Replay history of both committed and uncommitted transactions
  - Rather than selectively redo-ing committed transactions.
  - Then state of database guaranteed to be equivalent to that at the time of failure

Summary

- DRAM is volatile
- Avoid random writes to database on disk (NO FORCE)
- Support transactions with change sets > DRAM (STEAL)
- Recovery time must be bounded.
- Users must not suffer long delays due to checkpointing.
- Cope with repeated failures during recovery.
- Increase concurrency of undo (logical undo)
- Support record-level locking (avoid selective redo)

Conclusion

Parting Thoughts

- Protocols evolve over time to better handle user, workload, and hardware constraints.
- Deconstructing protocols will help you better appreciate the internals of complex software systems and learn the art of designing protocols.

Next Class

- Case Studies