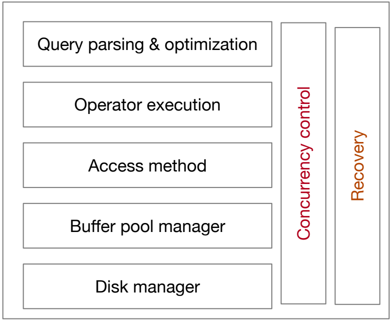
Crash Recovery

May 26, 2023

Announcement

- Closed book final exam: June 6.
- You may bring a cheat sheet of A4 size.

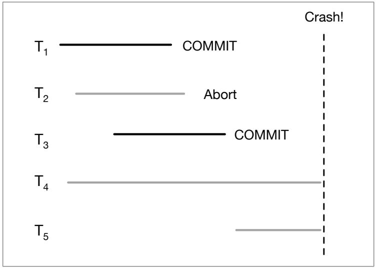
Overview



- Concurrency control: ensure isolation in concurrent database access.
- Recovery: ensure atomicity and durability via logging (this lecture).

Crash recovery

- Atomicity: TXNs may abort/rollback.
- Durability: What if DBMS stops running?



Desired state after system restarts:

- T_1 and T_3 should be durable.
- T_2, T_4 and T_5 should be aborted (effects not seen).

Failure classification

(C1) Transaction failure
  - Logical errors : TXNs cannot complete due to some internal condition.
  - System errors : DBMS terminates an active TX due to an error condition (e.g., deadlock).

(C2) System crash: a power failure or other hardware or software failures cause DBMS crash.

(C3) Disk failure: a head crash or similar disk failure destroys all or part of disk storage.

A recovery algorithm aims to handle (C1) and (C2) but not (C3).

Log-based crash recovery

- Logging: actions taken during normal transaction processing to ensure enough information exists to recover from failures.
- Recovery: actions taken after a failure to recover the database contents to a state that ensures atomicity, consistency and durability.

Logging

Log-based recovery

- A log is a sequence of records that keep information about update activities on the DB.
- Basic idea: Write what a TXN T plan to do in the log and leave enough information in the log so that we can figure out whether T has did it or not.
- Question #1: What information is written in the log?
- Question #2: When to write to the log?

Log records

- $\langle T_i \text{ start} \rangle$: T_i has started.
- $\langle T_i, X, V_{\text{old}}, V_{\text{new}} \rangle$: T_i executes $W(X)$ to update its values from V_{old} to V_{new}.
- $\langle T_i \text{ commit} \rangle$: T_i has committed.
- $\langle T_i \text{ abort} \rangle$: T_i has aborted.

T_1	T_2
R(A)	
A: = A -5	
W(A)	
	B:= 100
	W(B)
	COMMIT
COMMIT	

Transactions

<T_1 start>

<T_1, A, 100, 95>

<T_2 start>

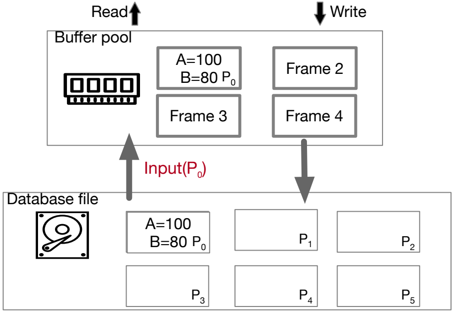
<T_2,B, 80, 100>

<T_2 commit>

<T_1 commit>

Log

Buffer pool polices



Read

Write

Buffer pool

A=100
B=80 P_0

Frame 2

Frame 3

Frame 4

Input(P_0)

Database file

A=100
B=80 P_0

P_1

P_2

P_3

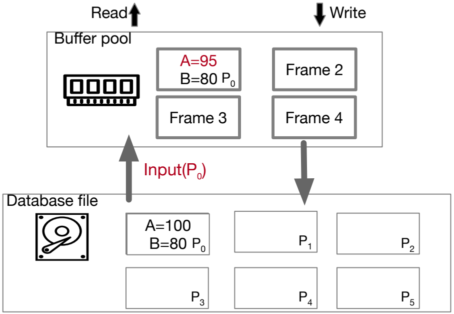
P_4

P_5

T_1	T_2
R(A)	
A: = A -5	

Transactions

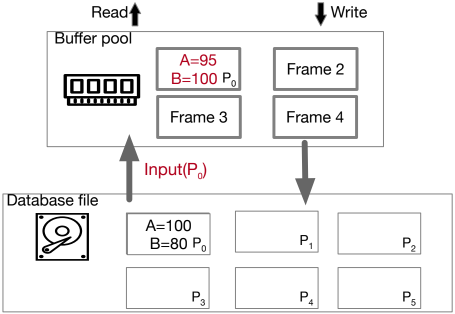
Buffer pool polices



T_1	T_2
R(A)	
A: = A -5	
W(A)	

Transactions

Buffer pool polices



Read

Write

Buffer pool

A=95
B=100 P_0

Frame 2

Frame 3

Frame 4

Input(P_0)

Database file

A=100
B=80 P_0

P_1

P_2

P_3

P_4

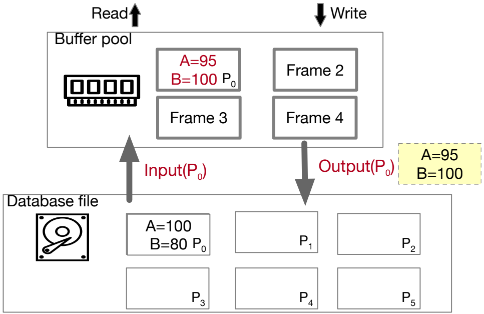
P_5

T_1	T_2
R(A)	
A: = A -5	
W(A)	
	B:= 100
	W(B)
	COMMIT

Transactions

- No-Force: A TXN can commit even if its updates have not been flushed to disk.

Buffer pool polices

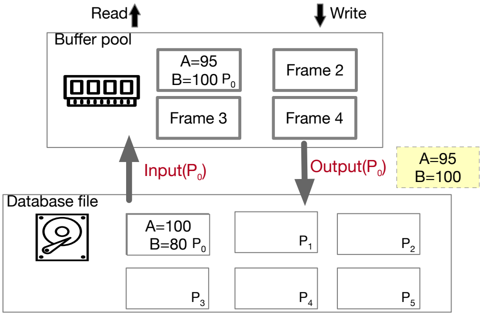


T_1	T_2
R(A)	
A: = A -5	
W(A)	
	B:= 100
	W(B)
	COMMIT
COMMIT	

Transactions

- No-Force: A TXN can commit even if its updates have not been flushed to disk.
- Steal: A buffer pool page with uncommitted updates can be flushed to disk anytime.

Buffer pool polices



T_1	T_2
R(A)	
A: = A -5	
W(A)	
	B:= 100
	W(B)
	COMMIT ✗
COMMIT	

Transactions

- No-Force: A TXN can commit even if its updates have not been flushed to disk.
- Steal: A buffer pool page with uncommitted updates can be flushed to disk anytime.

Question: What would happen if the DBMS crashes while outputing P_0?

Force vs. No-Force

Whether require a TXN to flush all its updates to disk before it is allowed to commit?

Force: Yes.

- Provides durability without REDO logging.
- Poor performance: many random writes at commit time.

No-Force: No.

- Complicates durability: what happens if DBMS crashes before the updates of a TXN are flushed to disk?
- Need to REDO updates by committed TXNs to ensure durability.
- Good performance: reduce random writes at commit time.

Steal vs. No-Steal

Whether allow buffer-pool pages with uncommitted data to overwrite committed data on disk?

No-Steal: No

- Useful for ensuring atomicity without UNDO logging.
- Poor runtime performance: consider a TXN that update all records in a table.

Steal: Yes

- Complicates atomicity: (i) what if a TXN that flushed updates to disk aborts? (ii) what if system crashes before a TXN is finished?
- Need to UNDO uncommitted TXNs to ensure atomicity.
- Good runtime performance.

Buffer pool polices (recap)

	No-Steal	Steal
No-Force		Fastest
Force	Slowest	

Performance implications

	No-Steal	Steal
No-Force		UNDO REDO
Force	NO UNDO NO REDO	

Logging/recovery implications

Preferred buffer pool policy: NO-Force + Steal.

Question. How to ensure correctness?

Write-Ahead Logging (WAL)

Buffer pool policy: No-force + Steal.

WAL rule: Log records that correspond to the changes made to a DB object must have been written to disk before the DB object is allowed to be flushed to disk.

1. Log records are output to disk in the order in which they are created.
2. A TXN T_i enters the commit state only after the log record $\langle T_i \text{ commit} \rangle$ has been output to disk.
3. Before a buffer pool page is output to disk, all log records pertaining to the data in page must have been output to disk.

Example of WAL

<T_1 start>

<T_1, A, 100, 95>

<T_2 start>

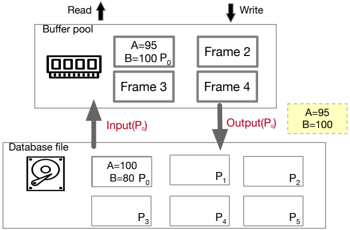
<T_2,B, 80, 100>

<T_2 commit>

Log

T_1	T_2
R(A)	
A: = A -5	
W(A)	
	B:= 100
	W(B)
	COMMIT ✗
COMMIT	

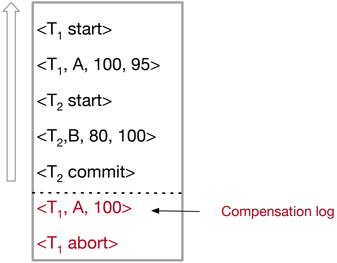
Transactions



1. When is T_2 considered committed?
   - After the log record $\langle T_2$ commit$\rangle$ has been flushed to disk.
2. What appends if buffer pool page in Frame 1 is flushed to disk?
   - The log records up to $\langle T_2, B, 80, 100\rangle$ has been flushed to disk.
3. Assuming T_2 has committed, what if the DBMS crashes while flushing Frame 1 to disk?
   - Need to UNDO T_1 and REDO T_2.

UNDO for atomicity

UNDO(T): restore values of all data items updated by T to their old values.



<T_1 start>

<T_1, A, 100, 95>

<T_2 start>

<T_2,B, 80, 100>

<T_2 commit>

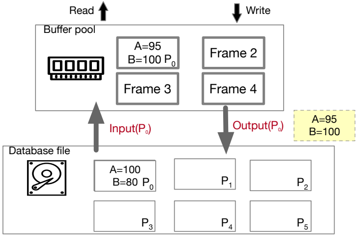
<T_1, A, 100>

<T_1 abort>

Undo T_1

T_1	T_2
R(A)	
A: = A -5	
W(A)	
	B:= 100
	W(B)
	COMMIT ✗
COMMIT	

Transactions



- Process the log backwards. Why?
- Write a compensation log record to the log whenever an old values is resorted.
- After UNDO(T) is done, write a record $\langle T \text{ abort}\rangle$ to the log.
- UNDO helps to deliver atomicity.

REDO for durability

REDO(T): set the values of all data items updated by T to the new values.

<T_1 start>

<T_1, A, 100, 95>

<T_2 start>

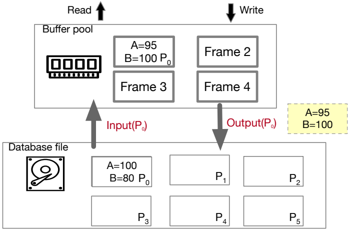
<T_2,B, 80, 100>

<T_2 commit>

Redo T_2

T_1	T_2
R(A)	
A: = A -5	
W(A)	
	B:= 100
	W(B)
	COMMIT ✗
COMMIT	

Transactions



- Process forward from the first log record of T.
- No logging is done in this case.
- Helps to provides durability.

WAL Recap

- WAL rule: Log records that correspond to the changes made to a DB object must be written to disk before the DB object is allowed to be flushed to disk.
- Performance: Enable "Steal + No-Force" buffer pool policy.
  - Steal: dirt pages can be flushed to disk anytime.
  - No-Force: can commit even if its modifications have not been flushed to disk.
- Correctness: Require UNDO & REDO logging
  - Undo incomplete TXNs to ensure atomicity
  - Redo committed ones to guarantee durability.

Checkpointing

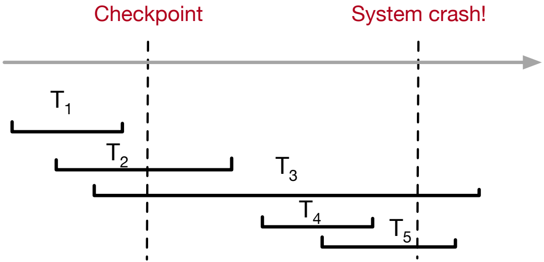
To UNDO/REDO all TXNs recorded in the log can be expensive.

DBMS periodically takes a checkpoint where it flush all buffers to disk.

1. Stop accepting new TXNs.
2. Flush all log records currently residing the memory.
3. Flush all dirt pages to disk.
4. Write a log record $\langle$checkpoint L$\rangle$ to disk.
5. L is a list of TXN still active at the time of checkpoint.

Recovery starts from the last checkpoint.

Checkpoint example



- T_1 can be ignored since update already flushed to disk due to checkpoint.
- Need to redo T_2 and T_4.
- Need to undo T_3 and T_5.

Recovery Algorithm

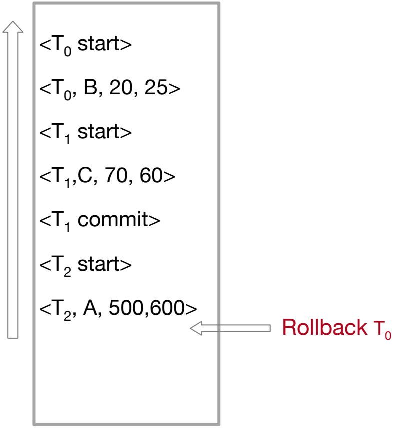
Recovery algorithm

- A recovery algorithm takes care of both normal rollback and recovery from system crash.
- Logging during normal operation
  - Write $\langle T_i, \text{sart}\rangle$ record when TXN T_i starts
  - Write $\langle T_i, X, V_{\text{old}}, V_{\text{new}}\rangle$ record for each update
  - Write $\langle T_i, \mathbf{commit}\rangle$ when TXN T_i ends

  The logging process follows the WAL protocol.
- Conduct checkpointing periodically to reduce recovery costs.

Transaction rollback

- Let T be the TXN to be rolled back.
- Scan the log backwards from the end, and for each log record of the form $\langle T, X, V_1, V_2 \rangle$
  - Update X with the old value V_1 (UNDO).
  - Write a compensation log record $< T, X, V_1 >$.
- Once the record $\langle T, start \rangle$ found, stop the scan and append a log record $\langle T, abort \rangle$.



<T_0 start>

<T_0, B, 20, 25>

<T_1 start>

<T_1,C, 70, 60>

<T_1 commit>

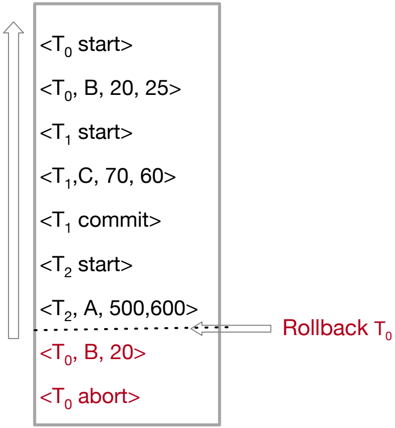
<T_2 start>

<T_2, A, 500,600>

Rollback T_0

Transaction rollback

- Let T be the TXN to be rolled back.
- Scan the log backwards from the end, and for each log record of the form $\langle T, X, V_1, V_2 \rangle$
  - Update X with the old value V_1 (UNDO).
  - Write a compensation log record $< T, X, V_1 >$.
- Once the record $\langle T, start \rangle$ found, stop the scan and append a log record $\langle T, abort \rangle$.



<T_0 start>

<T_0, B, 20, 25>

<T_1 start>

<T_1,C, 70, 60>

<T_1 commit>

<T_2 start>

<T_2, A, 500,600>

Rollback T_0

<T_0, B, 20>

<T_0 abort>

Recovery from a system crash

Analysis & redo phase:

- Analyze and identify which TXNs committed since checkpoint and which failed.
- Replay all actions of all TXNs. This is also known as repeating history.
- This phase makes sure committed TXNs are durable.

Undo phase:

- Undo all incomplete TXNs to ensure atomicity.
- Write compensation logs during the undo phase.

Analysis & redo phase

1. Find the last $\langle \text{Checkpoint } L \rangle$ record and set the undo-list to L.
2. Scan forward from the log $\langle \text{Checkpoint, } L \rangle$ record and repeat history as follows.
   - Whenever a record $\langle T_i, X_j, V_1, V_2 \rangle$ or $\langle T_i, X_j, V_2 \rangle$ is found, redo it by writing V_2 to X_j (repeat history).
   - Whenever a $\langle T_i \text{ start} \rangle$ record is found, add T_i to the undo-list.
   - Whenever a $\langle T_i \text{ commit} \rangle$ or $\langle T_i \text{ abort} \rangle$ record is found, remove T_i the undo-list.

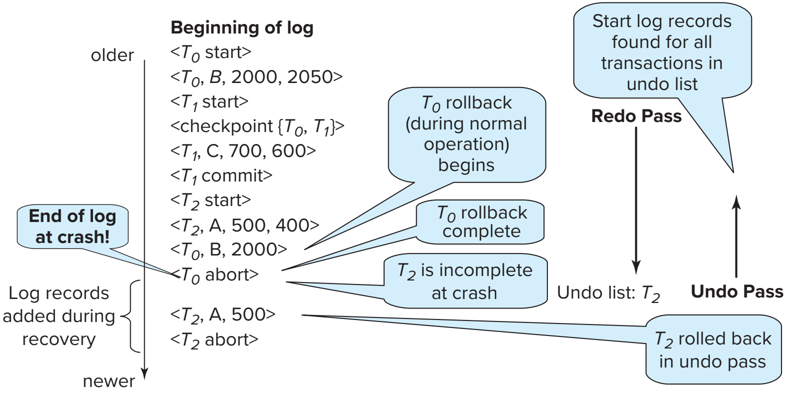
- The undo-list tracks incomplete TXNs and will be handled by the undo phase.
- Compensation log are also replayed. This simplifies the recovery logic.

Undo phase

Scan log backwards from the end to undo incomplete TXNs.

1. Whenever a record $\langle T_i, X_j, V_1, V_2 \rangle$ is found, where T_i is in the undo-list,
    - write V_1 to X_j (undo) and write a compensation log $\langle T_i, X_j, V_1 \rangle$.
2. Whenever a log record $\langle T_i\ \text{start} \rangle$ is found, where T_i is in the undo-list,
    - write a log record $\langle T_i\ \text{abort} \rangle$ and remove T_i from the undo-list.
3. The undo phase stops when the undo-list becomes empty.

A recovery example



Beginning of log

older

<T_0 start>

<T_0, B, 2000, 2050>

<T_1 start>

<checkpoint {T_0, T_1}>

<T_1, C, 700, 600>

<T_1 commit>

<T_2 start>

<T_2, A, 500, 400>

<T_0, B, 2000>

<T_0 abort>

<T_2, A, 500>

<T_2 abort>

newer

End of log at crash!

Log records added during recovery

T_0 rollback (during normal operation) begins

T_0 rollback complete

T_2 is incomplete at crash

T_2 rolled back in undo pass

Start log records found for all transactions in undo list

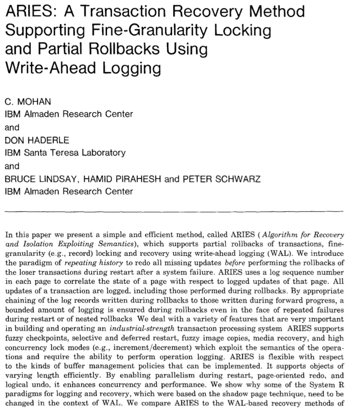
Redo Pass

Undo list: T_2

Undo Pass

The ARIES algorithm

- Algorithm for Recovery and Isolation Exploiting Semantics
- Developed at IBM Research in the early 1990s for DB2.
- The gold standard for recovery
    - Write-ahead logging
    - Repeating history during Redo
    - Logging changes during Undo
    - Many well-tuned optimizations.



ARIES: A Transaction Recovery Method Supporting Fine-Granularity Locking and Partial Rollbacks Using Write-Ahead Logging

C. MOHAN
IBM Almaden Research Center
and
DON HADERLE
IBM Santa Teresa Laboratory
and
BRUCE LINDSAY, HAMID PIRAHESH and PETER SCHWARZ
IBM Almaden Research Center

In this paper we present a simple and efficient method, called ARIES (*Algorithm for Recovery and Isolation Exploiting Semantics*), which supports partial rollbacks of transactions, fine-granularity (e.g., record) locking and recovery using write-ahead logging (WAL). We introduce the paradigm of *repeating history* to redo all missing updates *before* performing the rollbacks of the loser transactions during restart after a system failure. ARIES uses a log sequence number in each page to correlate the state of a page with respect to logged updates of that page. All updates of a transaction are logged, including those performed during rollbacks. By appropriate chaining of the log records written during rollbacks to those written during forward progress, a bounded amount of logging is ensured during rollbacks even in the face of repeated failures during restart or of nested rollbacks We deal with a variety of features that are very important in building and operating an *industrial-strength* transaction processing system ARIES supports fuzzy checkpoints, selective and deferred restart, fuzzy image copies, media recovery, and high concurrency lock modes (e.g., increment/decrement) which exploit the semantics of the operations and require the ability to perform operation logging. ARIES is flexible with respect to the kinds of buffer management policies that can be implemented. It supports objects of varying length efficiently. By enabling parallelism during restart, page-oriented redo, and logical undo, it enhances concurrency and performance. We show why some of the System R paradigms for logging and recovery, which were based on the shadow page technique, need to be changed in the context of WAL. We compare ARIES to the WAL-based recovery methods of

---

C. Mohan et al. ARIES: a transaction recovery method supporting fine-granularity locking and partial rollbacks using write-ahead logging. TODS 1992.