COMP 421: Files & Databases

Lecture 21: Recovery

UNC
DEPARTMENT OF
COMPUTER SCIENCE

Announcements

Project 3 due 12/3

Final exam: 12/12, in this room, 12:00 – 3:00

Review Session in class on Wednesday

Additional Review Session on 10/10 (reading day)

Details TBA

Crash Recovery

Recovery algorithms are techniques to ensure database consistency, transaction atomicity, and durability despite failures.

Recovery algorithms have two parts:

→ Actions during normal txn processing to ensure that the DBMS can recover from a failure.

→ Actions after a failure to recover the database to a state that ensures atomicity, consistency, and durability.

Today

Crash Recovery Overview

STEAL + NO-FORCE

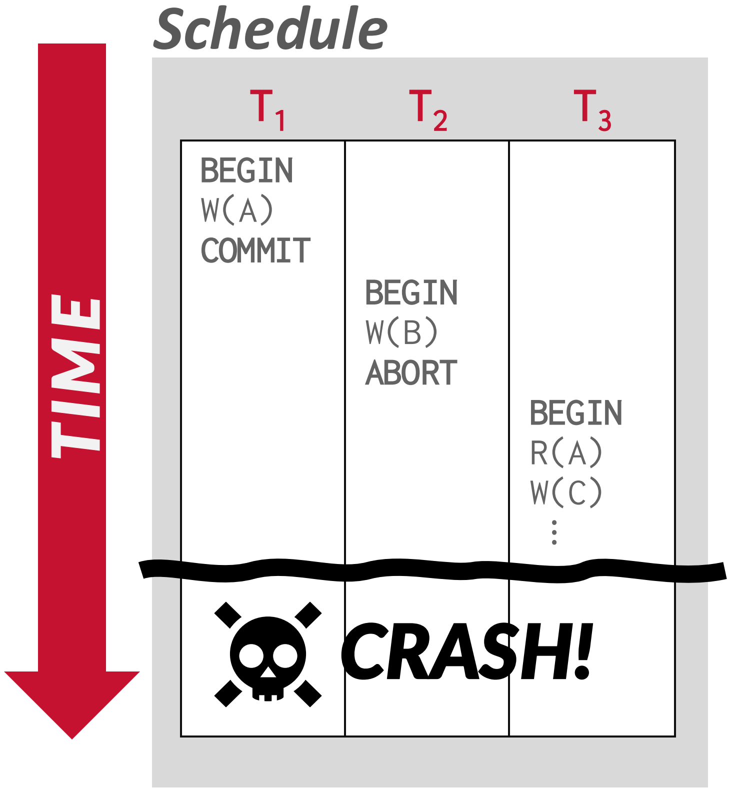
Atomicity: Txns may abort/fail.

Durability: Changes of committed txns should survive system failure.

Desired behavior after the DBMS restarts (i.e., the contents of volatile memory are lost):

→ T_1 should be durable.

→ T_2 + T_3 should be aborted.

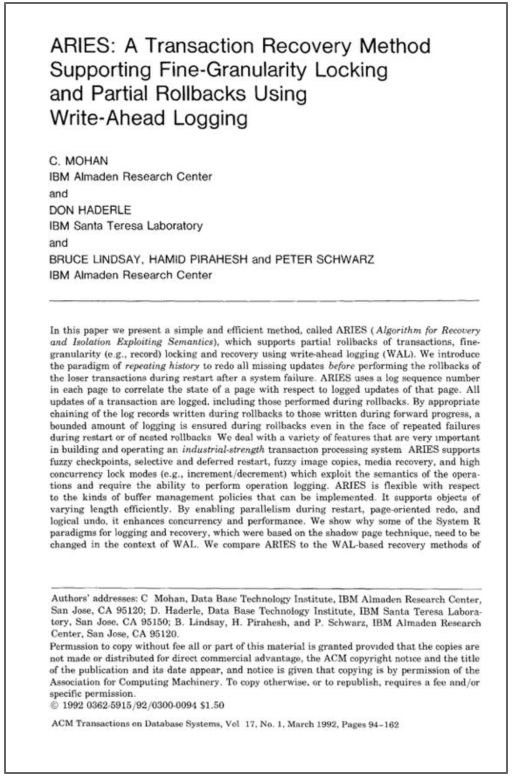


ARIES

Algorithms for Recovery and Isolation Exploiting Semantics

Developed at IBM Research in early 1990s for the DB2 DBMS.

Not all systems implement ARIES exactly as defined in this paper but they're close enough.



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

Authors' addresses: C Mohan, Data Base Technology Institute, IBM Almaden Research Center, San Jose, CA 95120; D. Haderle, Data Base Technology Institute, IBM Santa Teresa Laboratory, San Jose, CA 95150; B. Lindsay, H. Pirahesh, and P. Schwarz, IBM Almaden Research Center, San Jose, CA 95120.

ACM Transactions on Database Systems, Vol 17, No. 1, March 1992, Pages 94–162

ARIES – Main Ideas

Write-Ahead Logging:

→ Flush WAL record(s) changes to disk before a database object is written to disk.
→ Must use **STEAL** + **NO-FORCE** buffer pool policies.

Repeating History During Redo:

→ On DBMS restart, retrace actions and restore database to exact state before crash.

Logging Changes During Undo:

→ Record undo actions to log to ensure action is not repeated in the event of repeated failures.

Today's Agenda

Log Sequence Numbers

Normal Commit & Abort Operations

Fuzzy Checkpointing

Recovery Algorithm

WAL Records

We need to extend our log record format from last class to include additional info.

Every log record includes a globally unique **log sequence number** (LSN).

→ LSNs represent the physical order that txns make changes to the database.

Various components in the system keep track of **LSNs** that pertain to them…

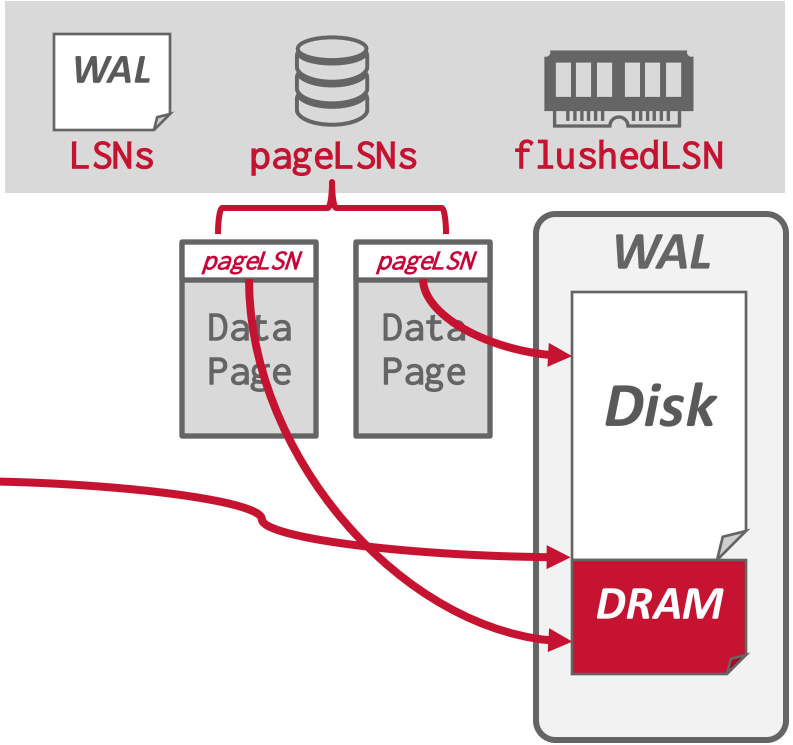
WAL Bookkeeping

Log Sequence Number (LSN).
→ Unique and monotonically increasing.

Each data page contains a **pageLSN**.
→ The LSN of the most recent log record that updated the page.

System keeps track of `flushedLSN`.
→ The max LSN flushed so far.

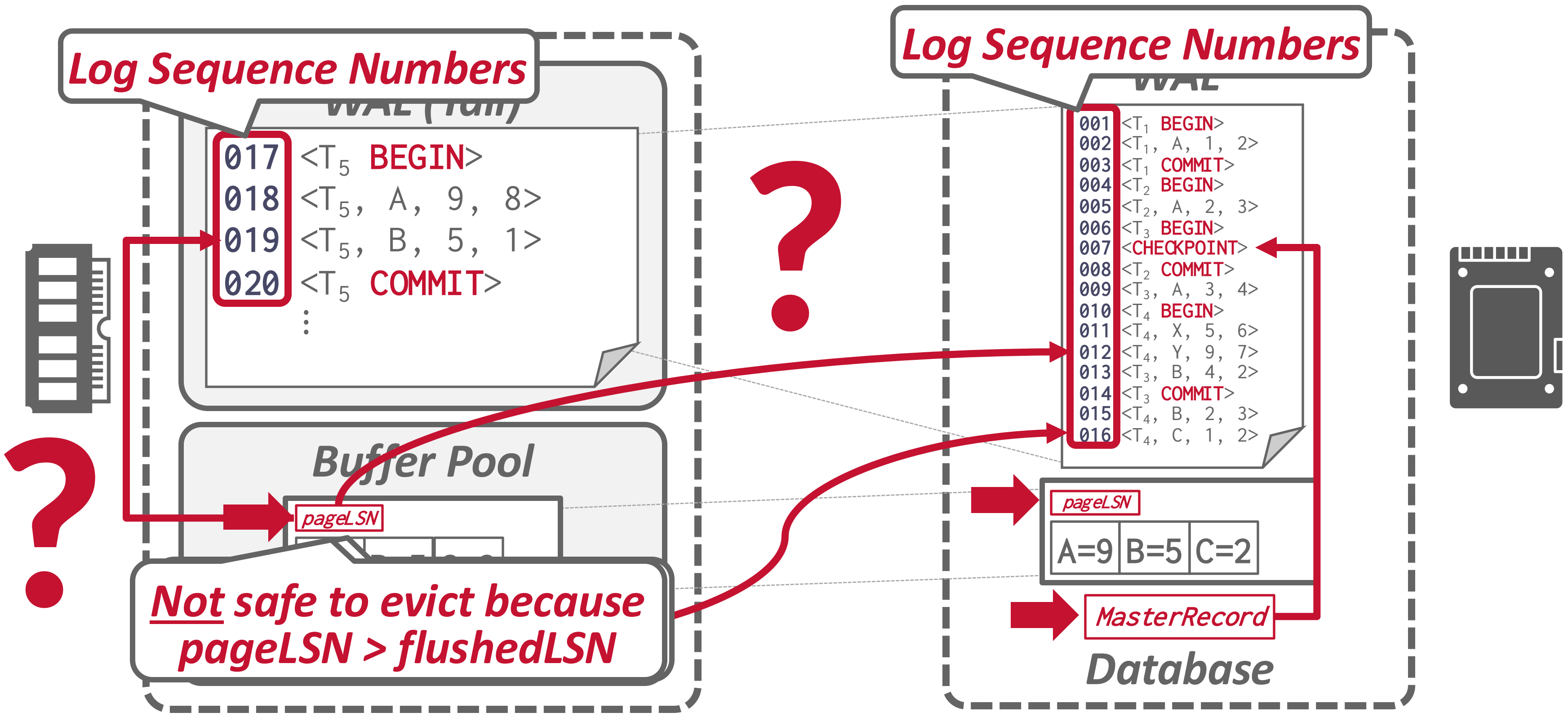
WAL: Before a page$_x$ is written,
pageLSN$_x$ ≤ flushedLSN



WAL Bookkeeping

Name	Location	Definition
flushedLSN	Memory	Last LSN in log on disk
pageLSN	$page_x$	Newest update to $page_x$
MasterRecord	Disk	LSN of latest checkpoint

Writing Log Records



Writing Log Records

All log records have an LSN.

Update the pageLSN every time a txn modifies a record in the page.

Update the flushedLSN in memory every time the DBMS writes the WAL buffer to disk.

Normal Execution

Each txn invokes a sequence of reads and writes, followed by commit or rollback.

Assumptions in this lecture:

→ All log records fit within a single page.
→ Disk writes are atomic.
→ Single-versioned tuples with Strong Strict 2PL.
→ **STEAL** + **NO-FORCE** buffer management with WAL.

Transaction Commit

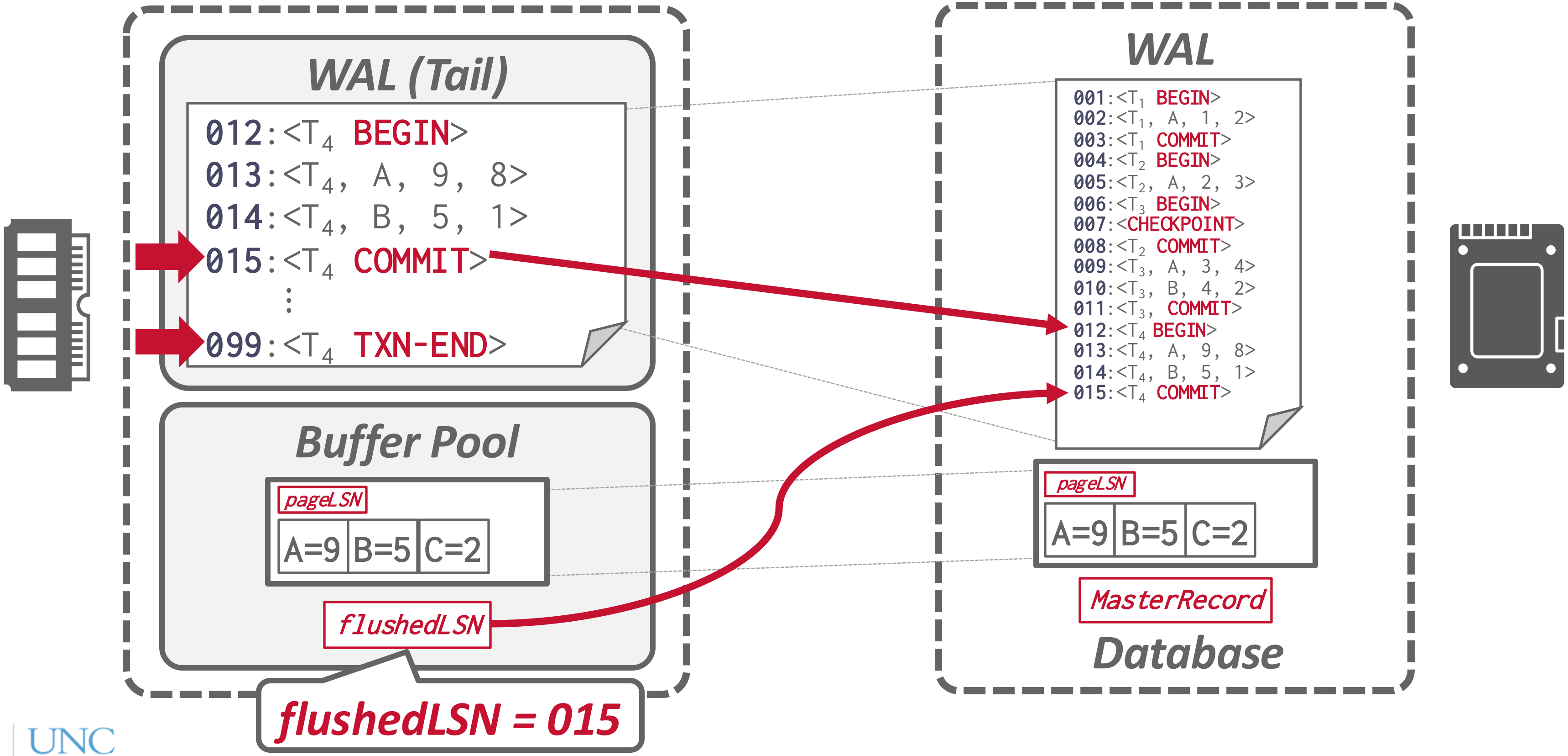
When a txn commits, the DBMS writes a COMMIT record to log and guarantees that all log records up to txn's COMMIT record are flushed to disk.

→ Log flushes are sequential, synchronous writes to disk.
→ Many log records per log page.

When commit succeeds, write a TXN-END record to log.

→ Indicates that no new log record for a txn will appear in the log ever again.
→ This does not need to be flushed immediately.

Transaction Commit



Transaction Abort

One mechanism will handle both rollback during normal operation ***and*** UNDO after failure

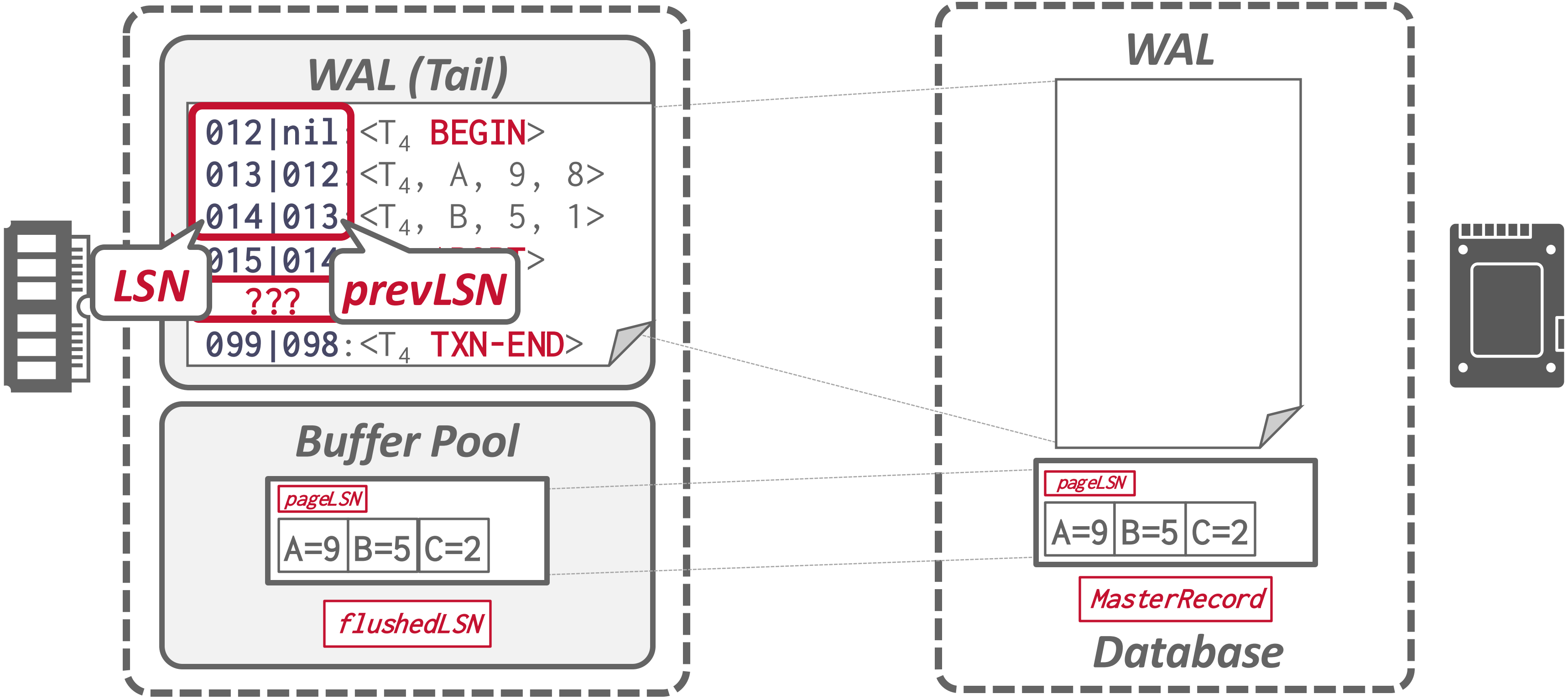
Easier to first understand normal rollbacks

We need to add another field to our log records:

→ `prevLSN`: The previous **LSN** for the txn.

→ This maintains a linked-list for each txn that makes it easy to walk through its records.

Transaction Abort



Compensation Log Records

A **CLR** describes the actions taken to undo the actions of a previous update record.

It has all the fields of an update log record plus the `undoNextLSN` pointer (i.e., the next-to-be-undone LSN).

CLRs are added to log records but the DBMS does not wait for them to be flushed before notifying the application that the txn aborted.

Transaction Abort – CLR Example

LSN	prevLSN	TxnId	Type	Object	Before	After	UndoNextLSN
001	nil	T_1	BEGIN	-	-	-	-
002	001	T_1	UPDATE	A	30	40	-
⋮							
011	002	T_1	ABORT	-	-	-	-

Transaction Abort – CLR Example

TIME

LSN	prevLSN	TxnId	Type	Object	Before	After	UndoNextLSN
001	nil	T_1	BEGIN	-	-	-	-
002	001	T_1	UPDATE	A	30	40	-
⋮							
011	002	T_1	ABORT	-	-	-	-
⋮							
026	011	T_1	CLR-002	A	40	30	001

The LSN of the next log record to be undone.

Transaction Abort – CLR Example

TIME

LSN	prevLSN	TxnId	Type	Object	Before	After	UndoNextLSN
001	nil	T_1	BEGIN	-	-	-	-
002	001	T_1	UPDATE	A	30	40	-
⋮							
011	002	T_1	ABORT	-	-	-	-
⋮							
026	011	T_1	CLR-002	A	40	30	001
027	026	T_1	TXN-END	-	-	-	nil

ABORT Algorithm

First write an ABORT record to log for the txn.

Then analyze the txn's updates in reverse order.
For each update record:
→ Write a CLR entry to the log.
→ Restore old value.

Lastly, write a TXN-END record and release locks.

Notice: CLRs never need to be undone.

Today's Agenda

~~Log Sequence Numbers~~

~~Normal Commit & Abort Operations~~

Fuzzy Checkpointing

Recovery Algorithm

Non-Fuzzy Checkpoints

The DBMS halts everything when it takes a checkpoint to ensure a consistent snapshot:

→ Halt the start of any new txns.
→ Flushes dirty pages on disk.

This is bad for runtime performance but makes recovery easy.

Non-Fuzzy Checkpoints

Pause modifying txns while the DBMS takes the checkpoint.

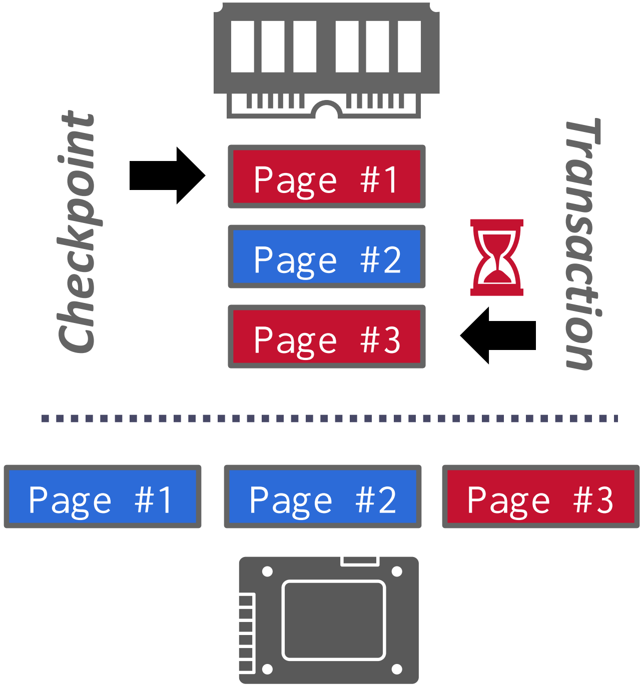
→ Prevent queries from acquiring write latch on table/index pages.

→ Don't have to wait until all txns finish before taking the checkpoint.

To allow updates during checkpoints, must record internal state as of the beginning of the checkpoint.

→ **Active Transaction Table (ATT)**

→ **Dirty Page Table (DPT)**



Active Transaction Table (ATT)

One entry per currently active txn.

→ `txnId`: Unique txn identifier.
→ `status`: The current "mode" of the txn.
→ `lastLSN`: Most recent LSN created by txn.

Remove entry after the `TXN-END` record.

Txn Status Codes:

→ Running (R)
→ Committing (C)
→ Candidate for Undo (U)

Dirty Page Table (DPT)

Keep track of which pages in the buffer pool contain changes that have not been flushed to disk.

One entry per dirty page in the buffer pool:

→ **recLSN**: The LSN of the log record that first caused the page to be dirty.

Slightly Better Checkpoints

At the first checkpoint, assuming P_{11} was flushed, T_2 is still running and there is only one dirty page (P_{22}).

At the second checkpoint, assuming P_{22} was flushed, T_2 and T_3 are active and the dirty pages are (P_{11}, P_{33}).

How to use this metadata to avoid stalling txns during checkpoint?

```
001:<T1 BEGIN>
002:<T2 BEGIN>
003:<T1, A→P11, 100, 120>
004:<T1 COMMIT>
005:<T2, C→P22, 100, 120>
006:<T1 TXN-END >
007:<CHECKPOINT
     ATT={T2},
     DPT={P22}>
008:<T3 BEGIN>
009:<T2, A→P11, 120, 130>
010:<T2 COMMIT>
011:<T3, B→P33, 200, 400>
012:<CHECKPOINT
     ATT={T2,T3},
     DPT={P11,P33}>
013:<T3, B→P33, 400, 600>
```

Fuzzy Checkpoints

A **fuzzy checkpoint** is where the DBMS allows active txns to continue to run while the system writes the log records for checkpoint.

→ No attempt to force dirty pages to disk.

New log records to track checkpoint boundaries:

→ `CHECKPOINT-BEGIN`: Indicates start of checkpoint

→ `CHECKPOINT-END`: Contains ATT + DPT.

Fuzzy Checkpoints

Assume the DBMS flushes P_{11} before the first checkpoint starts.

Any txn that begins after the checkpoint starts is excluded from the ATT in the CHECKPOINT-END record.

The LSN of the CHECKPOINT-BEGIN record is written to the MasterRecord when it completes.

001:<T_1 BEGIN>
002:<T_2 BEGIN>
003:<T_1, $A \rightarrow P_{11}$, 100, 120>
004:<T_1 COMMIT>
005:<T_2, $C \rightarrow P_{22}$, 100, 120>
006:<T_1 TXN-END>
007:<CHECKPOINT-BEGIN>
008:<T_3 BEGIN>
009:<T_2, $A \rightarrow P_{11}$, 120, 130>
010:<CHECKPOINT-END
ATT={T_2},
DPT={P_{22}}>
011:<T_2 COMMIT>
012:<T_3, $B \rightarrow P_{33}$, 200, 400>
013:<CHECKPOINT-BEGIN>
014:<T_3, $B \rightarrow P_{33}$, 10, 12>
015:<CHECKPOINT-END
ATT={T_2,T_3},
DPT={P_{11},P_{33}}>

ARIES – Recovery Phases

Phase #1 – Analysis

→ Examine the WAL in forward direction starting at MasterRecord 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 (even txns that will abort).

Phase #3 – Undo

→ Reverse the actions of txns that did not commit before the crash.

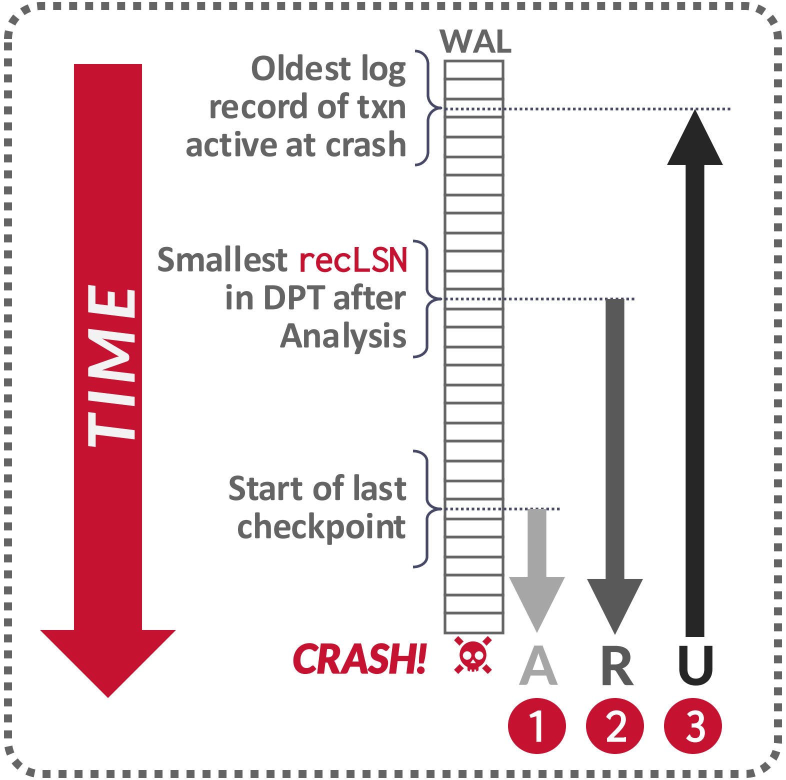
ARIES – Overview

Start from last BEGIN-CHECKPOINT found via MasterRecord.

Analysis: Figure out which txns committed or failed since checkpoint.

Redo: Repeat all actions.

Undo: Reverse effects of failed txns.



Analysis Phase

Scan log forward from last successful checkpoint.

If the DBMS finds a TXN-END record, remove its corresponding txn from **ATT**.

All other records:

→ If txn not in **ATT**, add it with status UNDO.

→ On commit, change txn status to COMMIT.

For update log records:

→ If page P not in **DPT**, add P to **DPT**, set its recLSN=LSN.

Analysis Phase

At end of the Analysis Phase:

→ **ATT** identifies which txns were active at time of crash.

→ **DPT** identifies which dirty pages might not have made it to disk.

Analysis Phase Example

WAL

010:<CHECK

⋮

020:<T_{96}, A→P_{33}, 10, 15>

Modify A in page P_{33}

⋮

030:<CHECKPOINT-END
ATT={T_{96},T_{97}},
DPT={P_{20}}>

⋮

040:<T_{96} COMMIT>

⋮

050:<T_{96} TXN-END>

⋮

CRASH!

LSN	ATT	DPT
010		
020		
030		
040		
050		

ATT: *(TxnId, Status)*

DPT: *(PageId, RecLSN)*

Redo Phase

The goal is to repeat history to reconstruct the database state at the moment of the crash:
→ Reapply all updates (even aborted txns!) and redo CLRs.

There are techniques that allow the DBMS to avoid unnecessary reads/writes, but we will ignore that in this lecture…

Redo Phase

Scan forward from the log record containing smallest recLSN in **DPT**.

For each update log record or CLR with a given LSN, redo the action unless:

→ The affected page is not in **DPT**, or

→ The affected page is in **DPT**, but that log record's LSN is less than the page's recLSN. (The update was propagated to disk.)

→ Log record's LSN ≤ pageLSN;
DBMS must fetch page from the disk to read page value.

Redo Phase

To redo an action:

→ Reapply logged update.

→ Set pageLSN to log record's LSN.

→ No additional logging, no forced flushes!

At the end of Redo Phase, write TXN-END log records for all txns with status C and remove them from the **ATT**.

Undo Phase

Undo all txns that were active at the time of crash and therefore will never commit.

→ These are all the txns with U status in the **ATT** after the Analysis Phase.

Process them in reverse LSN order using the `lastLSN` to speed up traversal.

→ At each step, pick the largest `lastLSN` across all transactions in the **ATT**.

→ Traverse `lastLSN`s in the same order, but in reverse, for how the updates happened originally.

Write a **CLR** for every modification.

Full Example

prevLSNs

001:<CHECKPOINT-BEGIN>

002:<CHECKPOINT-END>

003:<T_1, A→P_5, 1, 2>

004:<T_2, B→P_3, 2, 3>

005:<T_1 ABORT>

006:<CLR Undo T_1 LSN=003>

007:<T_1 TXN-END>

008:<T_3, C→P_1, 4, 5>

009:<T_2, D→P_5, 6, 7>

CRASH!

Full Example

ATT

TxnId	Status	lastLSN
T_2	U	009

DPT

PageId	recLSN
P_1	008
P_3	004
P_5	003

flushedLSN

001:<CHECKPOINT-BEGIN>

002:<CHECKPOINT-END>

003:<T_1, A→P_5, 1, 2>

004:<T_2, B→P_3, 2, 3>

005:<T_1 ABORT>

006:<CLR Undo T_1 LSN=003>

007:<T_1 TXN-END>

008:<T_3, C→P_1, 4, 5>

009:<T_2, D→P_5, 6, 7>

CRASH!

101:<CLR Undo T_2 LSN=009 UndoNext=004>

102:<CLR Undo T_3 LSN=008>

103:<T_3 TXN-END>

CRASH!

Full Example

ATT

TxnId	Status	lastLSN
T_2	U	101

DPT

PageId	recLSN
P_1	008
P_3	004
P_5	003

flushedLSN

001:<CHECKPOINT-BEGIN>
002:<CHECKPOINT-END>
003:<T_1, A→P_5, 1, 2>
004:<T_2, B→P_3, 2, 3>
005:<T_1 ABORT>
006:<CLR Undo T_1 LSN=003>
007:<T_1 TXN-END>
008:<T_3, C→P_1, 4, 5>
009:<T_2, D→P_5, 6, 7>

CRASH!

101:<CLR Undo T_2 LSN=009 UndoNext=004>
102:<CLR Undo T_3 LSN=008>
103:<T_3 TXN-END>

CRASH!

201:<CLR Undo T_2 LSN=004 UndoNext=nil>
202:<T_2 TXN-END>

Additional Crash Issues (1)

What does the DBMS do if it crashes during recovery in the Analysis Phase?

→ Nothing. Just run recovery again.

What does the DBMS do if it crashes during recovery in the Redo Phase?

→ Again nothing. Redo everything again.

Additional Crash Issues (2)

How can the DBMS improve performance during recovery in the Redo Phase?

→ Assume that it is not going to crash again and flush all changes to disk asynchronously in the background.

How can the DBMS improve performance during recovery in the Undo Phase?

→ Lazily rollback changes before new txns access pages.

→ Rewrite the application to avoid long-running txns.

Conclusion

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

Log Sequence Numbers:

⟶ LSNs identify log records; linked into backwards chains per transaction via prevLSN.

⟶ pageLSN allows comparison of data page and log records.

Next Class

You now know how to build a single-node DBMS.

Next class, review for the final!