Carnegie Mellon University

Database Recovery

Lecture #22

Database Systems
15-445/15-645
Fall 2017

Andy Pavlo
Computer Science Dept.
Carnegie Mellon Univ.

ADMINISTRIVIA

Homework #6: Monday November 27th @ 11:59pm

Project #4: Wednesday December 6th @ 11:59am

No class on Wednesday November 22nd

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.

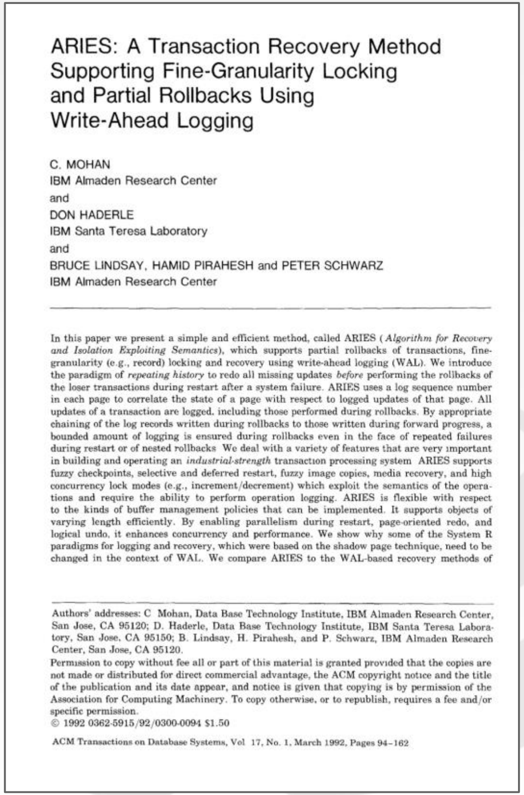
Today

ARIES

Algorithms for **R**ecovery and **I**solation **E**xploiting **S**emantics

Developed at IBM Research in early 1990s.

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



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.

Permission to copy without fee all or part of this material is granted provided that the copies are not made or distributed for direct commercial advantage, the ACM copyright notice and the title of the publication and its date appear, and notice is given that copying is by permission of the Association for Computing Machinery. To copy otherwise, or to republish, requires a fee and/or specific permission.
© 1992 0362-5915/92/0300-0094 $1.50

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

ARIES – MAIN IDEAS

Write-Ahead Logging:

→ Any change is recorded in log on stable storage before the database change is written to disk.

→ Has to be **STEAL** + **NO-FORCE**.

Repeating History During Redo:

→ On 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 now includes a globally unique **log sequence number** (LSN).

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

LOG SEQUENCE NUMBERS

Name	Where	Definition
`flushedLSN`	RAM	Last ***LSN*** in log on disk
`pageLSN`	@page$_i$	Newest update to page$_i$
`recLSN`	@page$_i$	Oldest update to page$_i$
`lastLSN`	T_j	Latest action of txn T_j
`MasterRecord`	Disk	***LSN*** of latest checkpoint

WRITING LOG RECORDS

Each data page contains a `pageLSN`.

→ The ***LSN*** of the most recent update to that page.

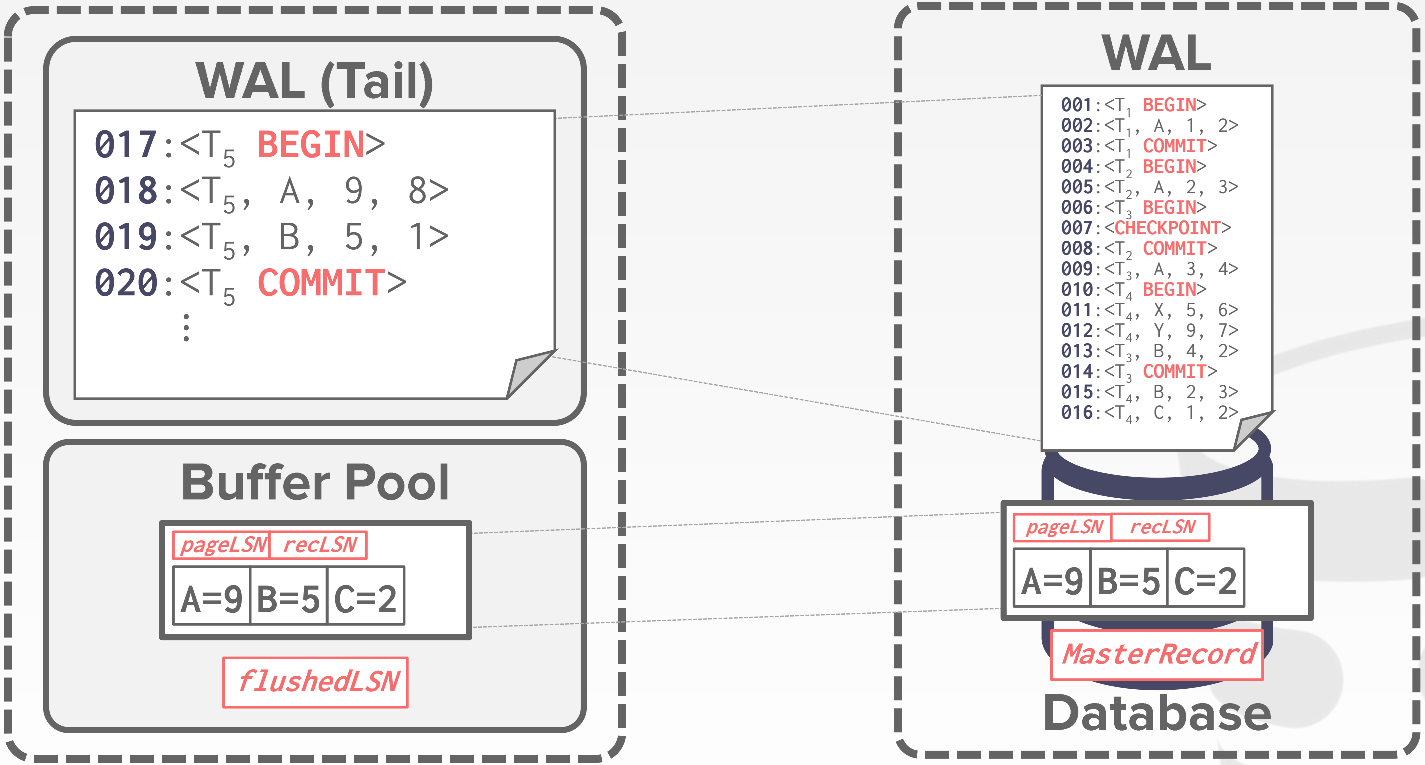
System keeps track of `flushedLSN`.

→ The max ***LSN*** flushed so far.

Before page `i` can be written to disk, we must flush log at least to the point where:

→ $\texttt{pageLSN}_i \le \texttt{flushedLSN}$

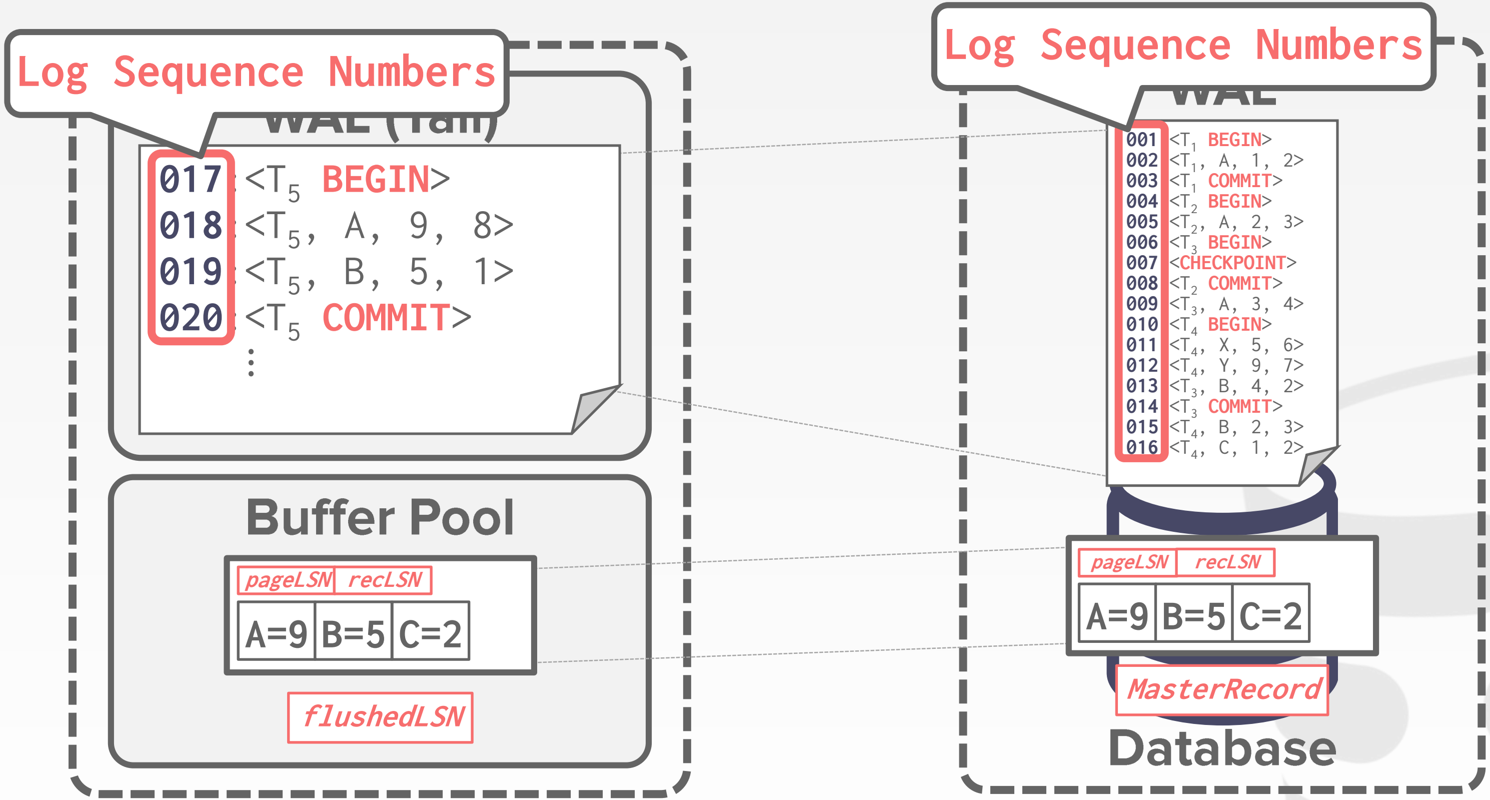
WRITING LOG RECORDS



017:<T_5 BEGIN>
018:<T_5, A, 9, 8>
019:<T_5, B, 5, 1>
020:<T_5 COMMIT>
⋮

001:<T_1 BEGIN>
002:<T_1, A, 1, 2>
003:<T_1 COMMIT>
004:<T_2 BEGIN>
005:<T_2, A, 2, 3>
006:<T_3 BEGIN>
007:<CHECKPOINT>
008:<T_2 COMMIT>
009:<T_3, A, 3, 4>
010:<T_4 BEGIN>
011:<T_4, X, 5, 6>
012:<T_4, Y, 9, 7>
013:<T_3, B, 4, 2>
014:<T_3 COMMIT>
015:<T_4, B, 2, 3>
016:<T_4, C, 1, 2>

WRITING LOG RECORDS



Log Sequence Numbers

WAL (Tail)

017 <T_5 BEGIN>
018 <T_5, A, 9, 8>
019 <T_5, B, 5, 1>
020 <T_5 COMMIT>
⋮

Buffer Pool

pageLSN	recLSN	
A=9	B=5	C=2

flushedLSN

Log Sequence Numbers

WAL

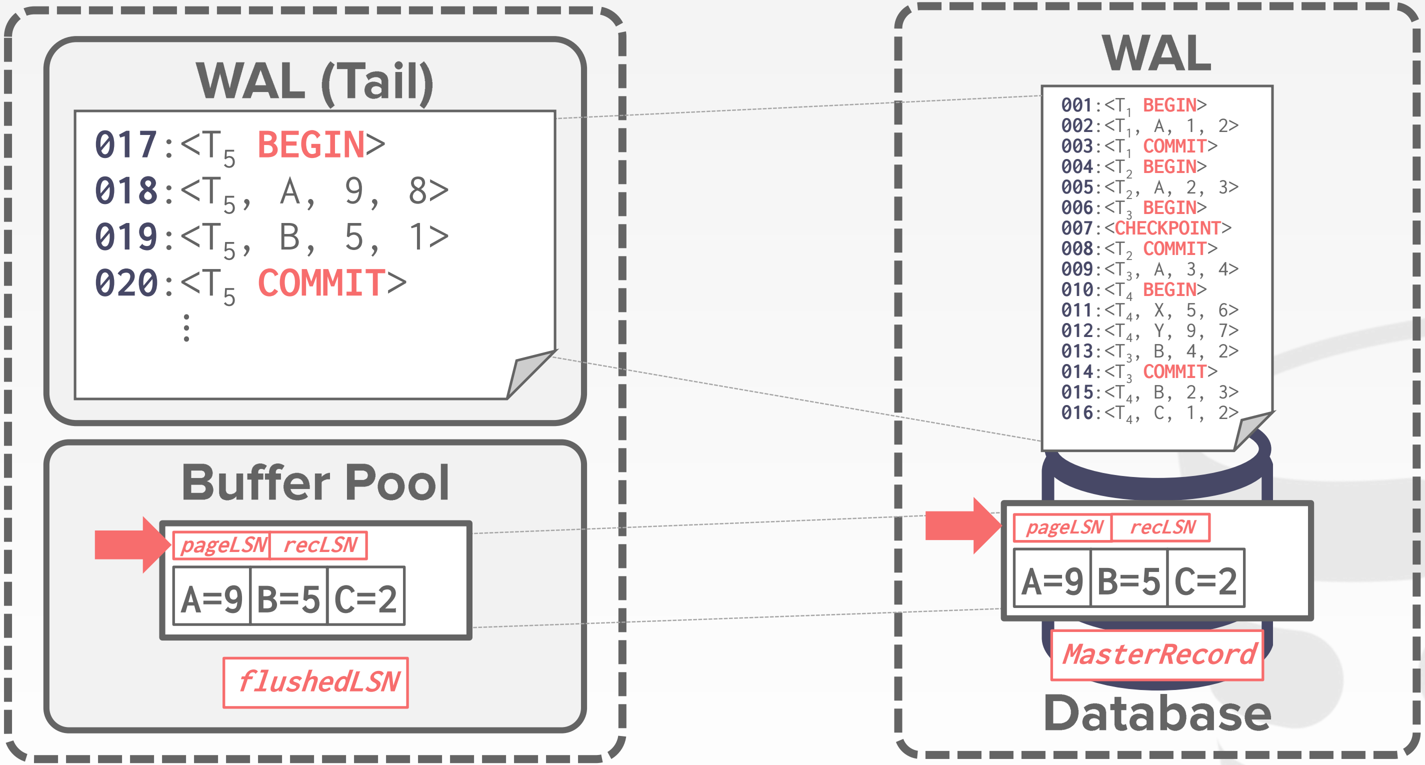
001 <T_1 BEGIN>
002 <T_1, A, 1, 2>
003 <T_1 COMMIT>
004 <T_2 BEGIN>
005 <T_2, A, 2, 3>
006 <T_3 BEGIN>
007 <CHECKPOINT>
008 <T_2 COMMIT>
009 <T_3, A, 3, 4>
010 <T_4 BEGIN>
011 <T_4, X, 5, 6>
012 <T_4, Y, 9, 7>
013 <T_3, B, 4, 2>
014 <T_3 COMMIT>
015 <T_4, B, 2, 3>
016 <T_4, C, 1, 2>

pageLSN	recLSN	
A=9	B=5	C=2

MasterRecord

Database

WRITING LOG RECORDS



WAL (Tail)

017:<T_5 BEGIN>
018:<T_5, A, 9, 8>
019:<T_5, B, 5, 1>
020:<T_5 COMMIT>
⋮

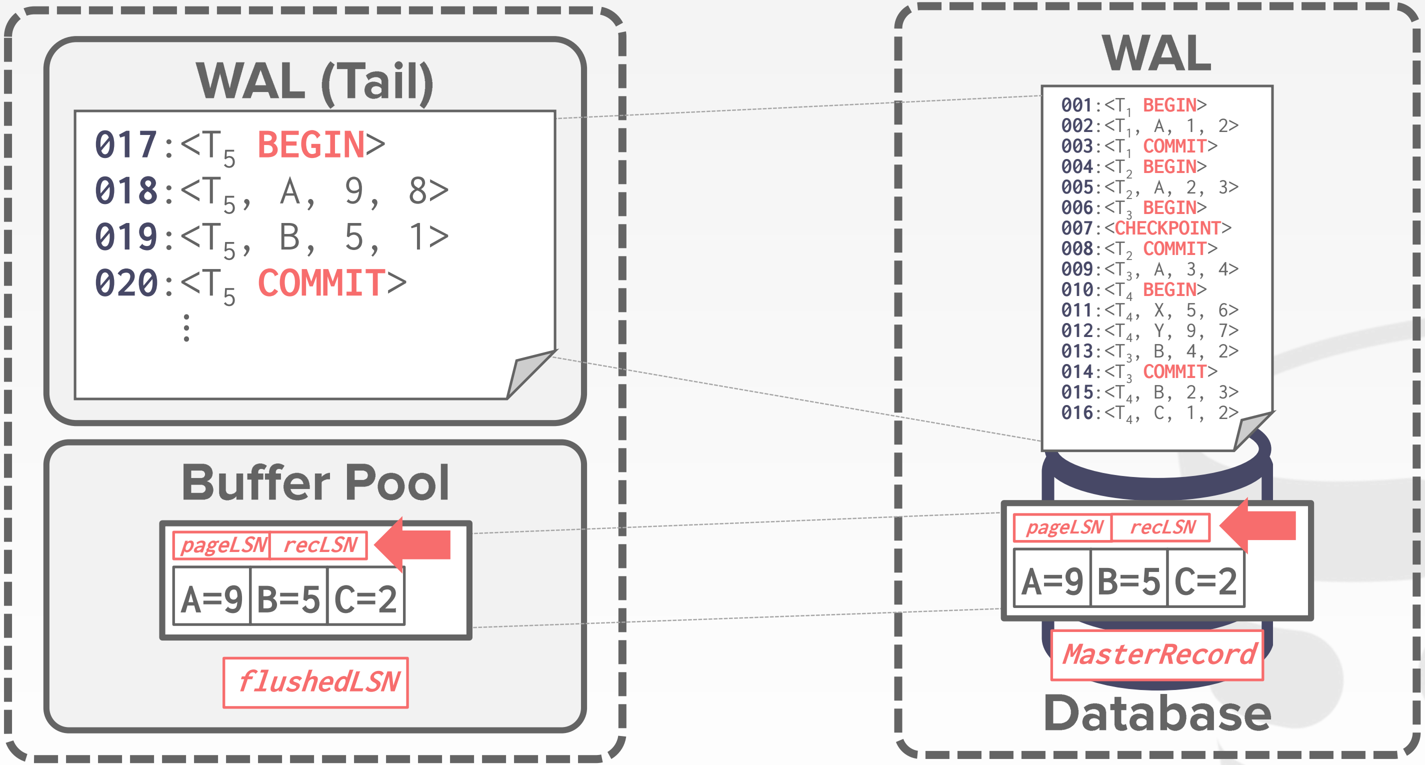
Buffer Pool

WAL

001:<T_1 BEGIN>
002:<T_1, A, 1, 2>
003:<T_1 COMMIT>
004:<T_2 BEGIN>
005:<T_2, A, 2, 3>
006:<T_3 BEGIN>
007:<CHECKPOINT>
008:<T_2 COMMIT>
009:<T_3, A, 3, 4>
010:<T_4 BEGIN>
011:<T_4, X, 5, 6>
012:<T_4, Y, 9, 7>
013:<T_3, B, 4, 2>
014:<T_3 COMMIT>
015:<T_4, B, 2, 3>
016:<T_4, C, 1, 2>

Database

WRITING LOG RECORDS



WAL (Tail)

017:<T_5 BEGIN>
018:<T_5, A, 9, 8>
019:<T_5, B, 5, 1>
020:<T_5 COMMIT>
⋮

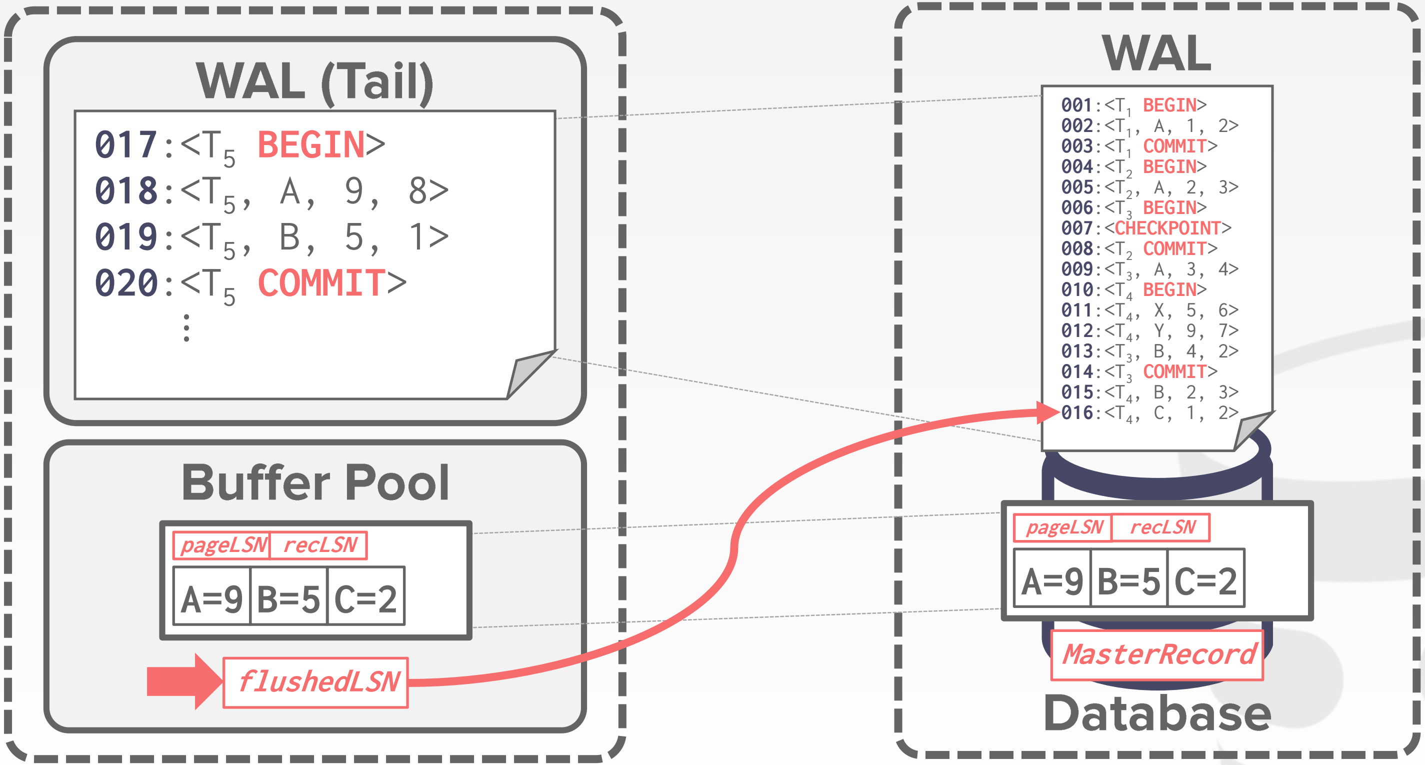
Buffer Pool

WAL

001:<T_1 BEGIN>
002:<T_1, A, 1, 2>
003:<T_1 COMMIT>
004:<T_2 BEGIN>
005:<T_2, A, 2, 3>
006:<T_3 BEGIN>
007:<CHECKPOINT>
008:<T_2 COMMIT>
009:<T_3, A, 3, 4>
010:<T_4 BEGIN>
011:<T_4, X, 5, 6>
012:<T_4, Y, 9, 7>
013:<T_3, B, 4, 2>
014:<T_3 COMMIT>
015:<T_4, B, 2, 3>
016:<T_4, C, 1, 2>

Database

WRITING LOG RECORDS



WAL (Tail)

017:<T_5 BEGIN>
018:<T_5, A, 9, 8>
019:<T_5, B, 5, 1>
020:<T_5 COMMIT>
⋮

Buffer Pool

pageLSN	*recLSN*

A=9	B=5	C=2

flushedLSN

WAL

001:<T_1 BEGIN>
002:<T_1, A, 1, 2>
003:<T_1 COMMIT>
004:<T_2 BEGIN>
005:<T_2, A, 2, 3>
006:<T_3 BEGIN>
007:<CHECKPOINT>
008:<T_2 COMMIT>
009:<T_3, A, 3, 4>
010:<T_4 BEGIN>
011:<T_4, X, 5, 6>
012:<T_4, Y, 9, 7>
013:<T_3, B, 4, 2>
014:<T_3 COMMIT>
015:<T_4, B, 2, 3>
016:<T_4, C, 1, 2>

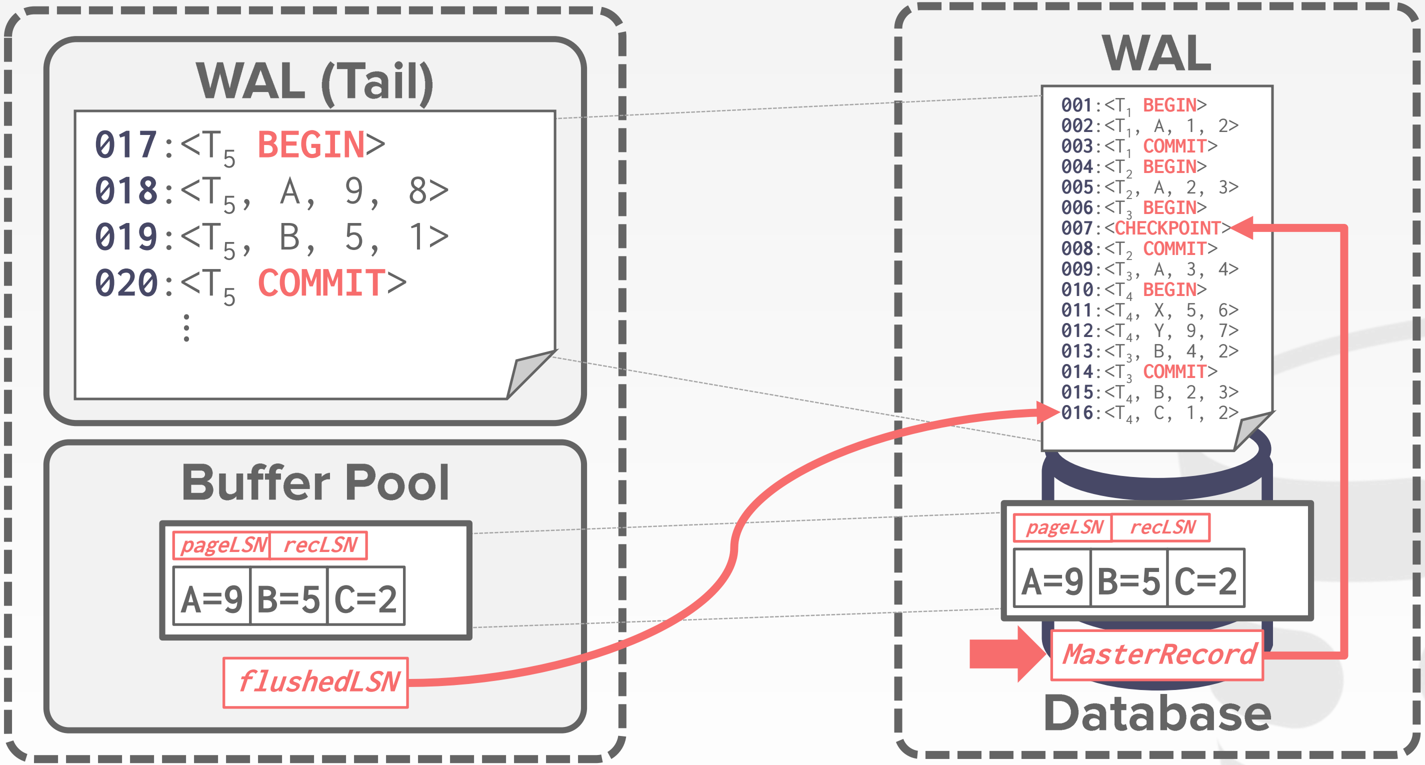
pageLSN	*recLSN*

A=9	B=5	C=2

MasterRecord

Database

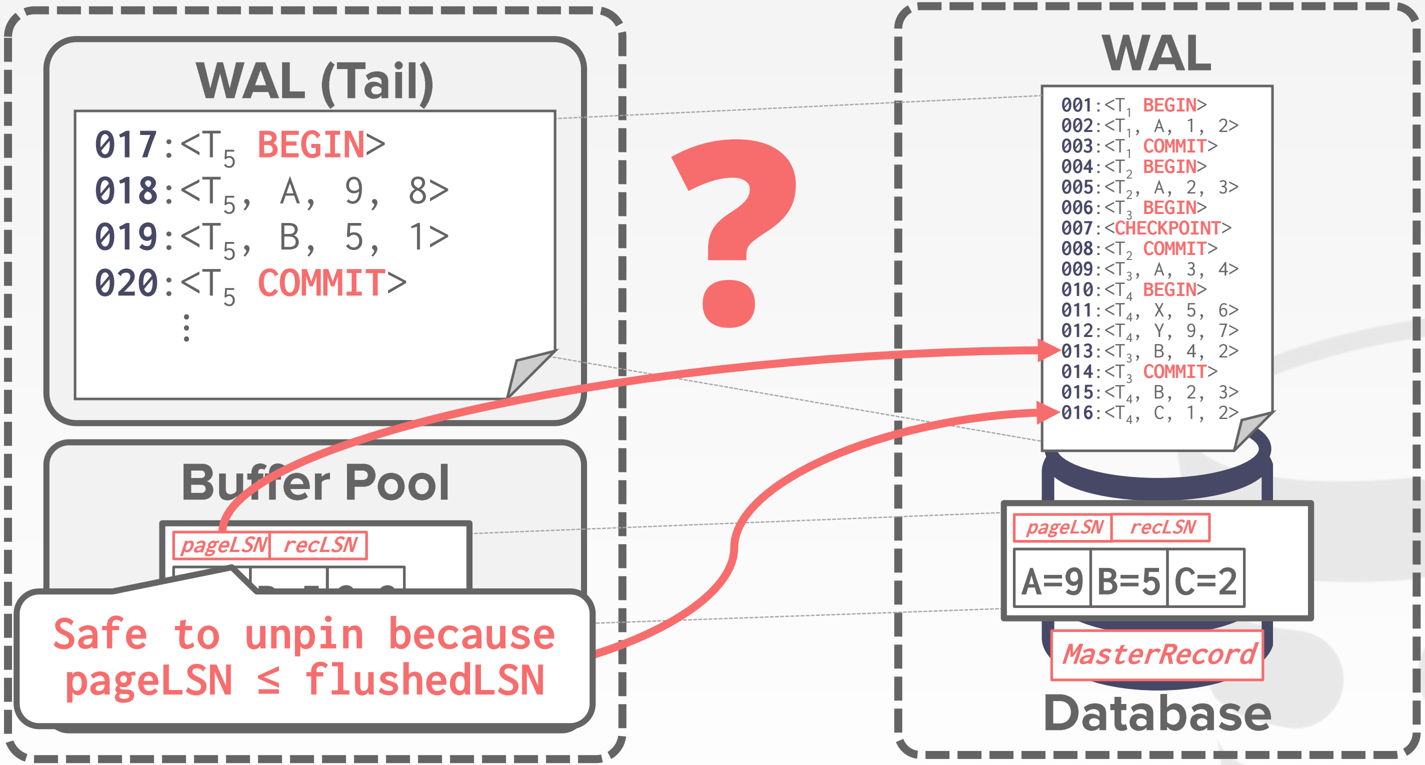
WRITING LOG RECORDS



017:<T_5 BEGIN>
018:<T_5, A, 9, 8>
019:<T_5, B, 5, 1>
020:<T_5 COMMIT>
⋮

001:<T_1 BEGIN>
002:<T_1, A, 1, 2>
003:<T_1 COMMIT>
004:<T_2 BEGIN>
005:<T_2, A, 2, 3>
006:<T_3 BEGIN>
007:<CHECKPOINT>
008:<T_2 COMMIT>
009:<T_3, A, 3, 4>
010:<T_4 BEGIN>
011:<T_4, X, 5, 6>
012:<T_4, Y, 9, 7>
013:<T_3, B, 4, 2>
014:<T_3 COMMIT>
015:<T_4, B, 2, 3>
016:<T_4, C, 1, 2>

WRITING LOG RECORDS



WAL (Tail)

017:<T_5 BEGIN>
018:<T_5, A, 9, 8>
019:<T_5, B, 5, 1>
020:<T_5 COMMIT>
⋮

?

Buffer Pool

pageLSN | recLSN

Safe to unpin because pageLSN ≤ flushedLSN

WAL

001:<T_1 BEGIN>
002:<T_1, A, 1, 2>
003:<T_1 COMMIT>
004:<T_2 BEGIN>
005:<T_2, A, 2, 3>
006:<T_3 BEGIN>
007:<CHECKPOINT>
008:<T_2 COMMIT>
009:<T_3, A, 3, 4>
010:<T_4 BEGIN>
011:<T_4, X, 5, 6>
012:<T_4, Y, 9, 7>
013:<T_3, B, 4, 2>
014:<T_3 COMMIT>
015:<T_4, B, 2, 3>
016:<T_4, C, 1, 2>

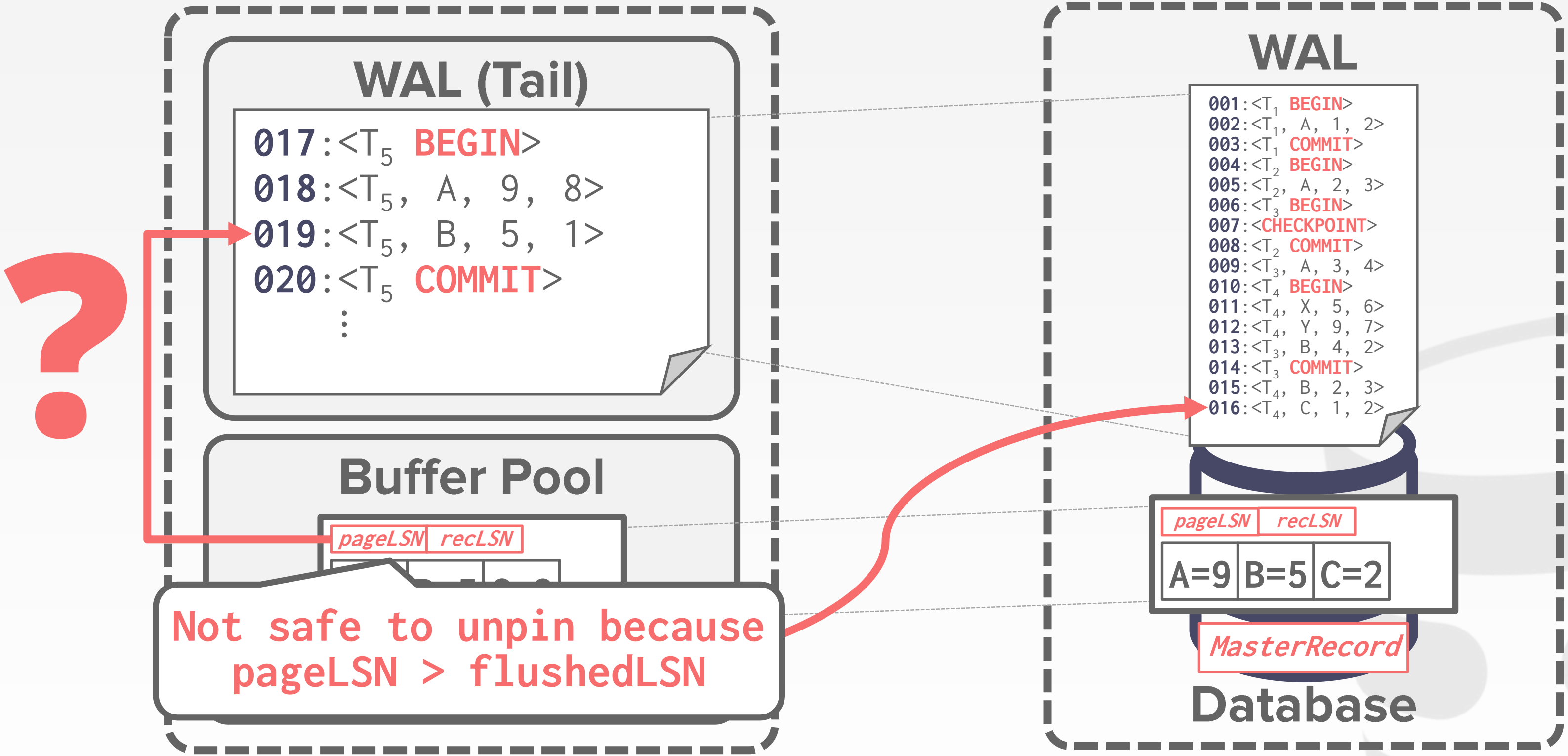
pageLSN | recLSN

A=9 | B=5 | C=2

MasterRecord

Database

WRITING LOG RECORDS



Not safe to unpin because pageLSN > flushedLSN

WRITING LOG RECORDS

All log records have an ***LSN***.

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

Record the **`flushedLSN`** in memory.

NORMAL EXECUTION

Series of reads & writes, followed by commit or abort.

Assumptions:

→ Disk writes are atomic.
→ Strict 2PL.
→ **STEAL** + **NO-FORCE** buffer management, with Write-Ahead Logging.

TRANSACTION COMMIT

Write `COMMIT` record to log.

All log records up to txn's `COMMIT` record are flushed to disk.

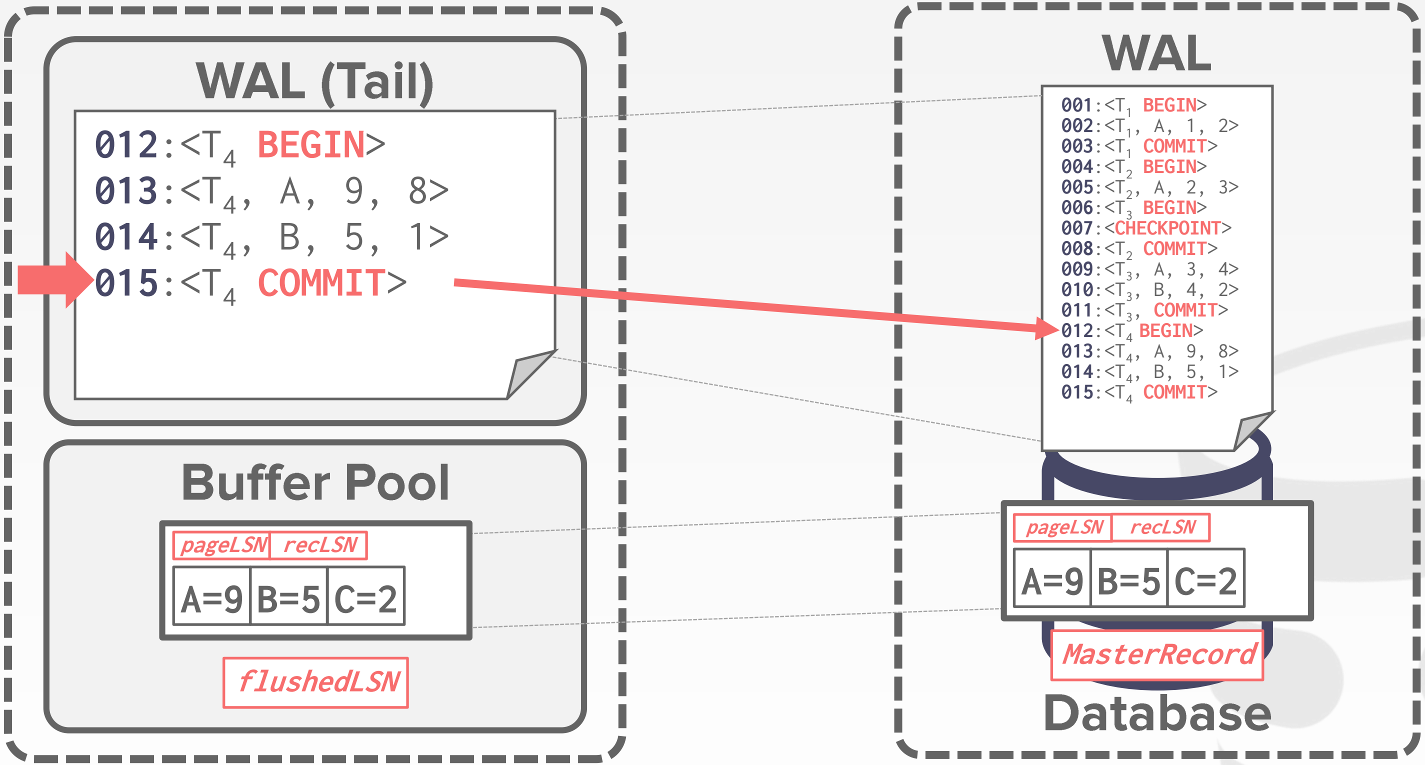
→ Note that log flushes are sequential, synchronous writes to disk.

→ Many log records per log page.

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

→ This does not need to be flushed immediately.

TRANSACTION COMMIT



WAL (Tail)

012:<T_4 BEGIN>
013:<T_4, A, 9, 8>
014:<T_4, B, 5, 1>
015:<T_4 COMMIT>

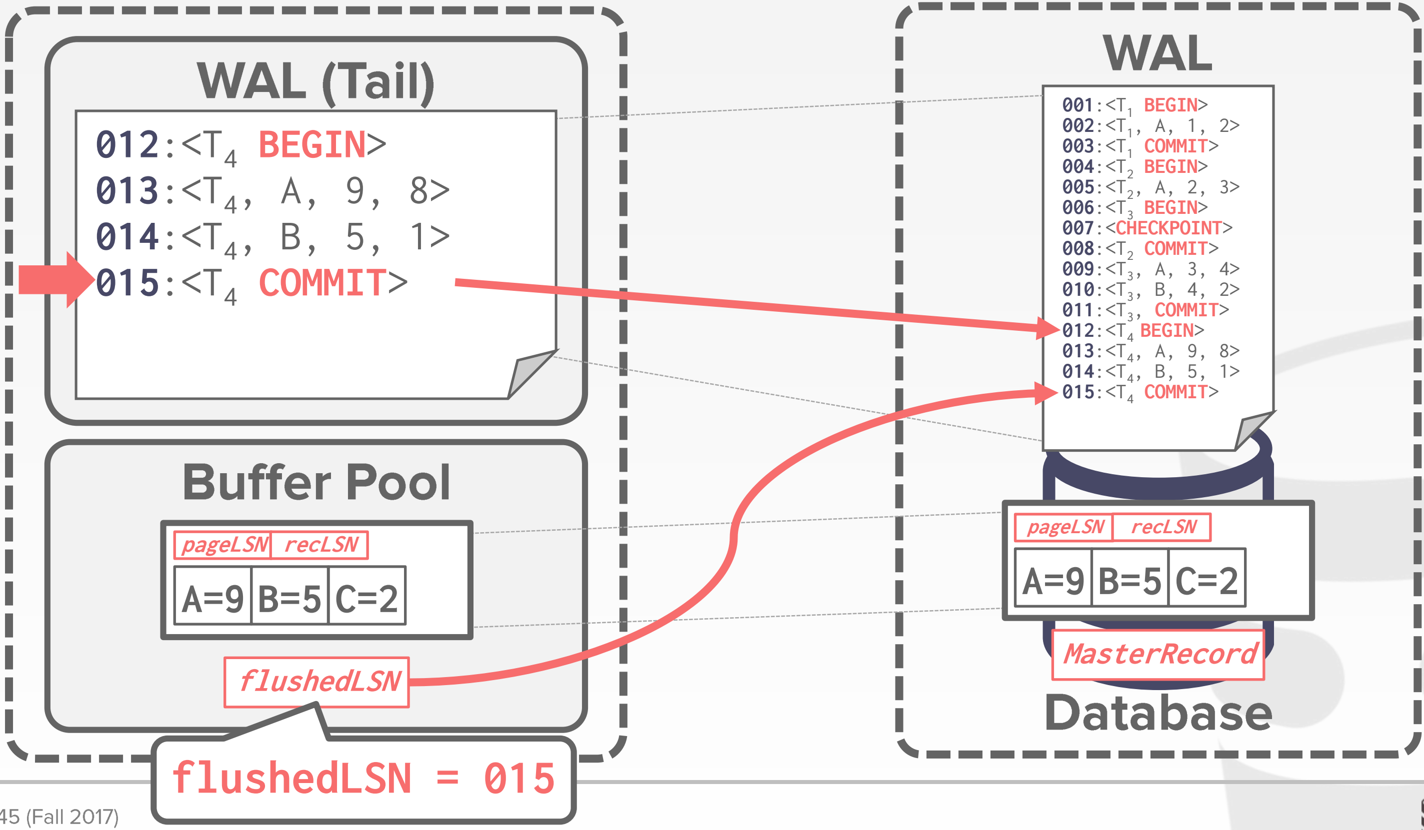
Buffer Pool

WAL

001:<T_1 BEGIN>
002:<T_1, A, 1, 2>
003:<T_1 COMMIT>
004:<T_2 BEGIN>
005:<T_2, A, 2, 3>
006:<T_3 BEGIN>
007:<CHECKPOINT>
008:<T_2 COMMIT>
009:<T_3, A, 3, 4>
010:<T_3, B, 4, 2>
011:<T_3, COMMIT>
012:<T_4 BEGIN>
013:<T_4, A, 9, 8>
014:<T_4, B, 5, 1>
015:<T_4 COMMIT>

Database

TRANSACTION COMMIT



WAL (Tail)

```
012:<T4 BEGIN>
013:<T4, A, 9, 8>
014:<T4, B, 5, 1>
015:<T4 COMMIT>
```

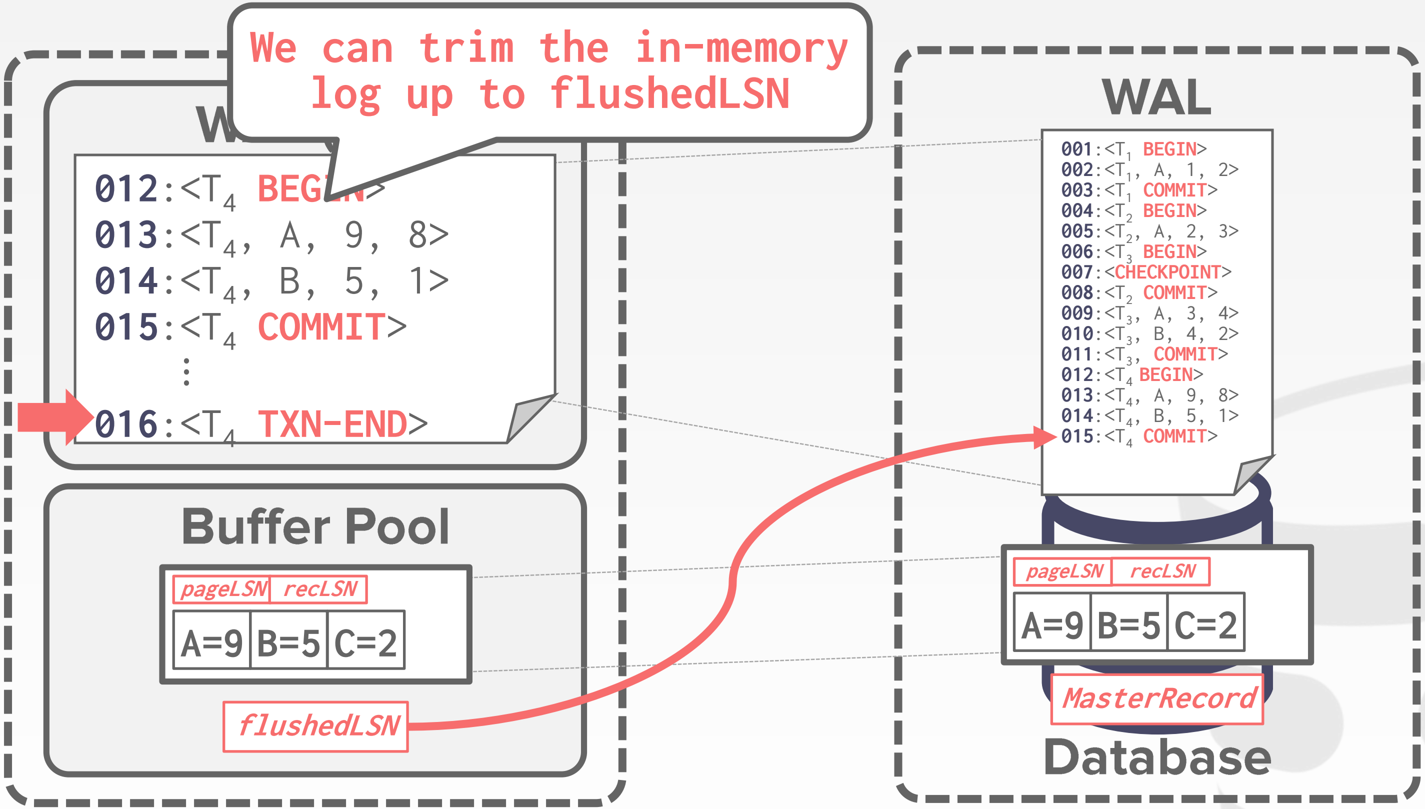
Buffer Pool

flushedLSN = 015

WAL

```
001:<T1 BEGIN>
002:<T1, A, 1, 2>
003:<T1 COMMIT>
004:<T2 BEGIN>
005:<T2, A, 2, 3>
006:<T3 BEGIN>
007:<CHECKPOINT>
008:<T2 COMMIT>
009:<T3, A, 3, 4>
010:<T3, B, 4, 2>
011:<T3, COMMIT>
012:<T4 BEGIN>
013:<T4, A, 9, 8>
014:<T4, B, 5, 1>
015:<T4 COMMIT>
```

TRANSACTION COMMIT



We can trim the in-memory log up to flushedLSN

W

012:<T_4 BEGIN>
013:<T_4, A, 9, 8>
014:<T_4, B, 5, 1>
015:<T_4 COMMIT>
⋮
016:<T_4 TXN-END>

Buffer Pool

pageLSN *recLSN*

A=9 B=5 C=2

flushedLSN

WAL

001:<T_1 BEGIN>
002:<T_1, A, 1, 2>
003:<T_1 COMMIT>
004:<T_2 BEGIN>
005:<T_2, A, 2, 3>
006:<T_3 BEGIN>
007:<CHECKPOINT>
008:<T_2 COMMIT>
009:<T_3, A, 3, 4>
010:<T_3, B, 4, 2>
011:<T_3, COMMIT>
012:<T_4 BEGIN>
013:<T_4, A, 9, 8>
014:<T_4, B, 5, 1>
015:<T_4 COMMIT>

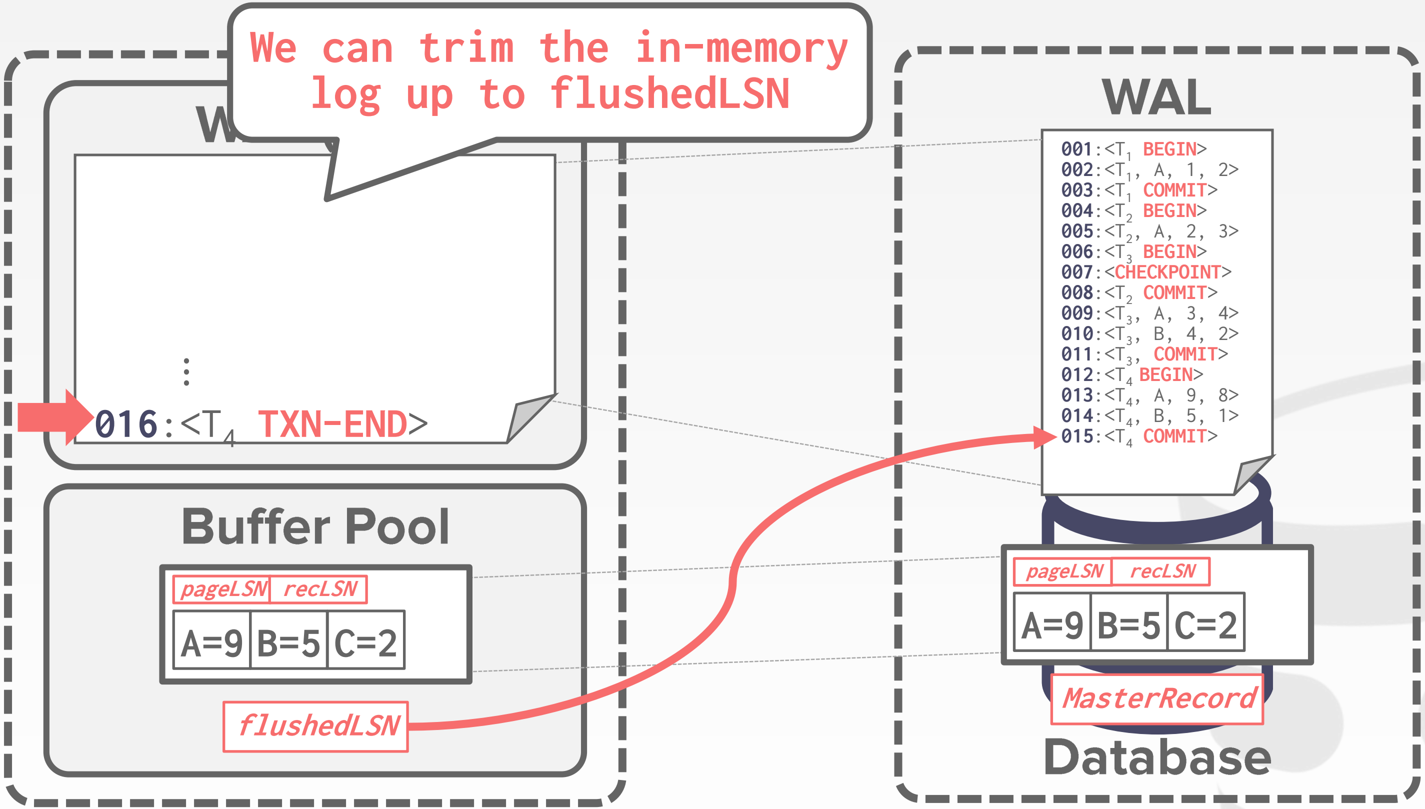
pageLSN *recLSN*

A=9 B=5 C=2

MasterRecord

Database

TRANSACTION COMMIT



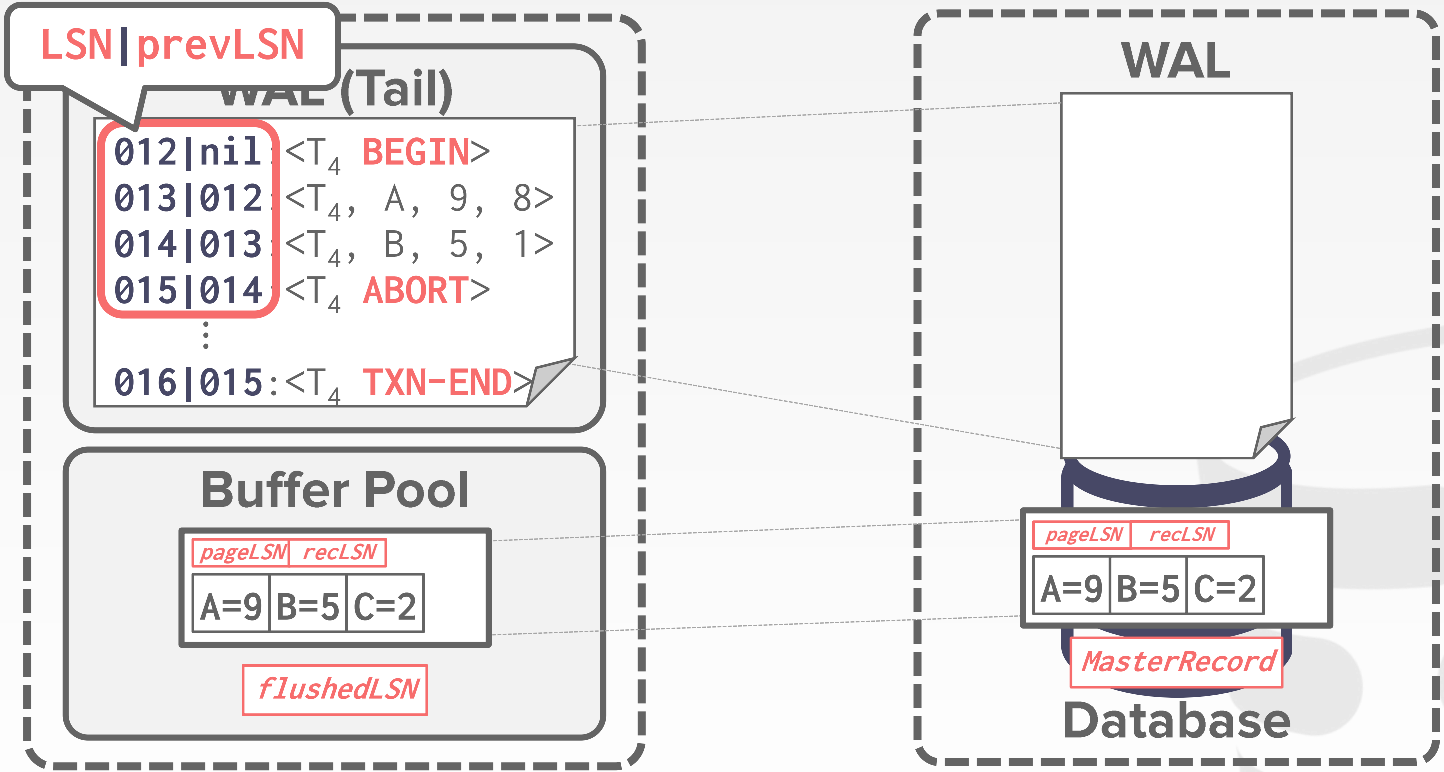
TRANSACTION ABORT

Aborting a txn is actually a special case of the ARIES undo operation applied to only one transaction.

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 COMMIT



LSN|prevLSN

WAL (Tail)

012|nil:<T_4 BEGIN>
013|012:<T_4, A, 9, 8>
014|013:<T_4, B, 5, 1>
015|014:<T_4 ABORT>
⋮
016|015:<T_4 TXN-END>

Buffer Pool

pageLSN	*recLSN*

A=9	B=5	C=2

flushedLSN

WAL

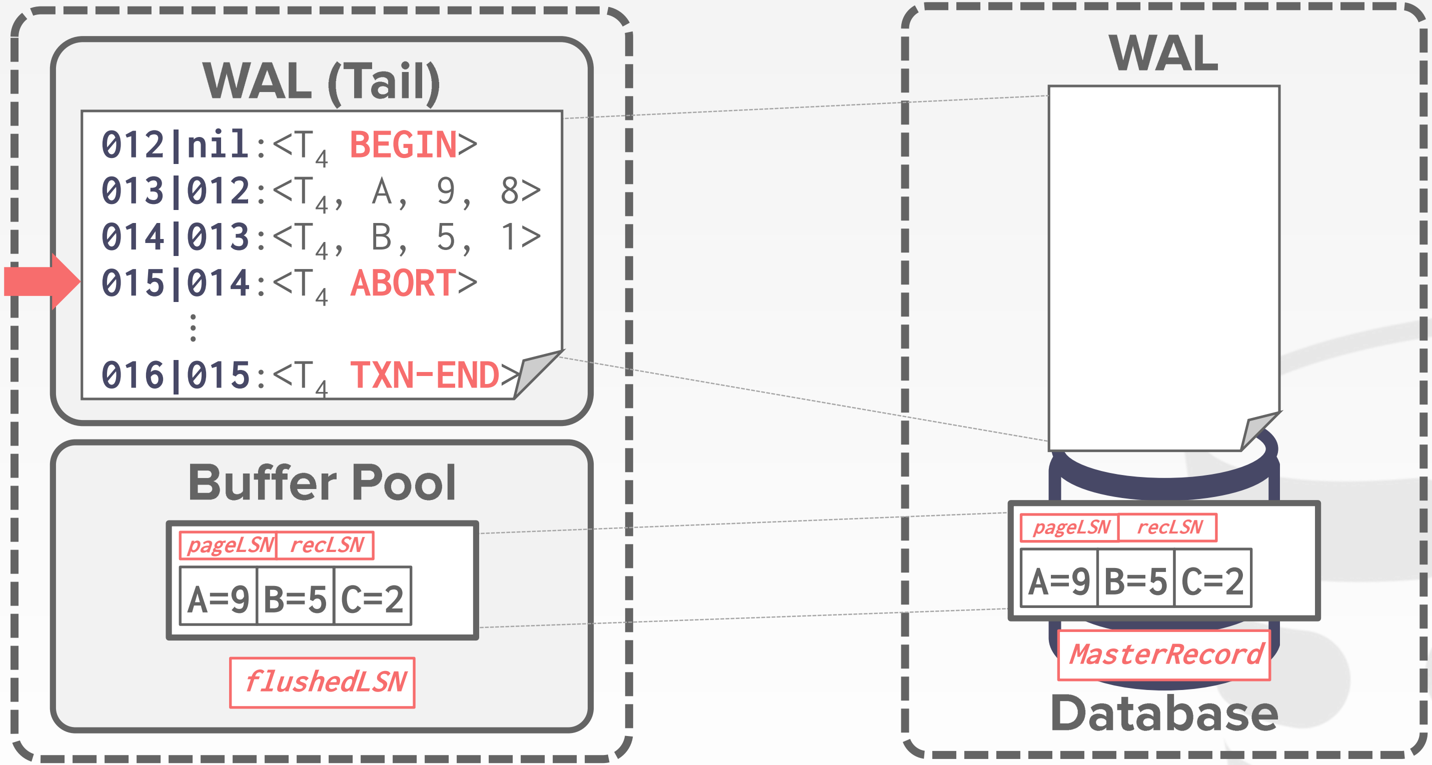
pageLSN	*recLSN*

A=9	B=5	C=2

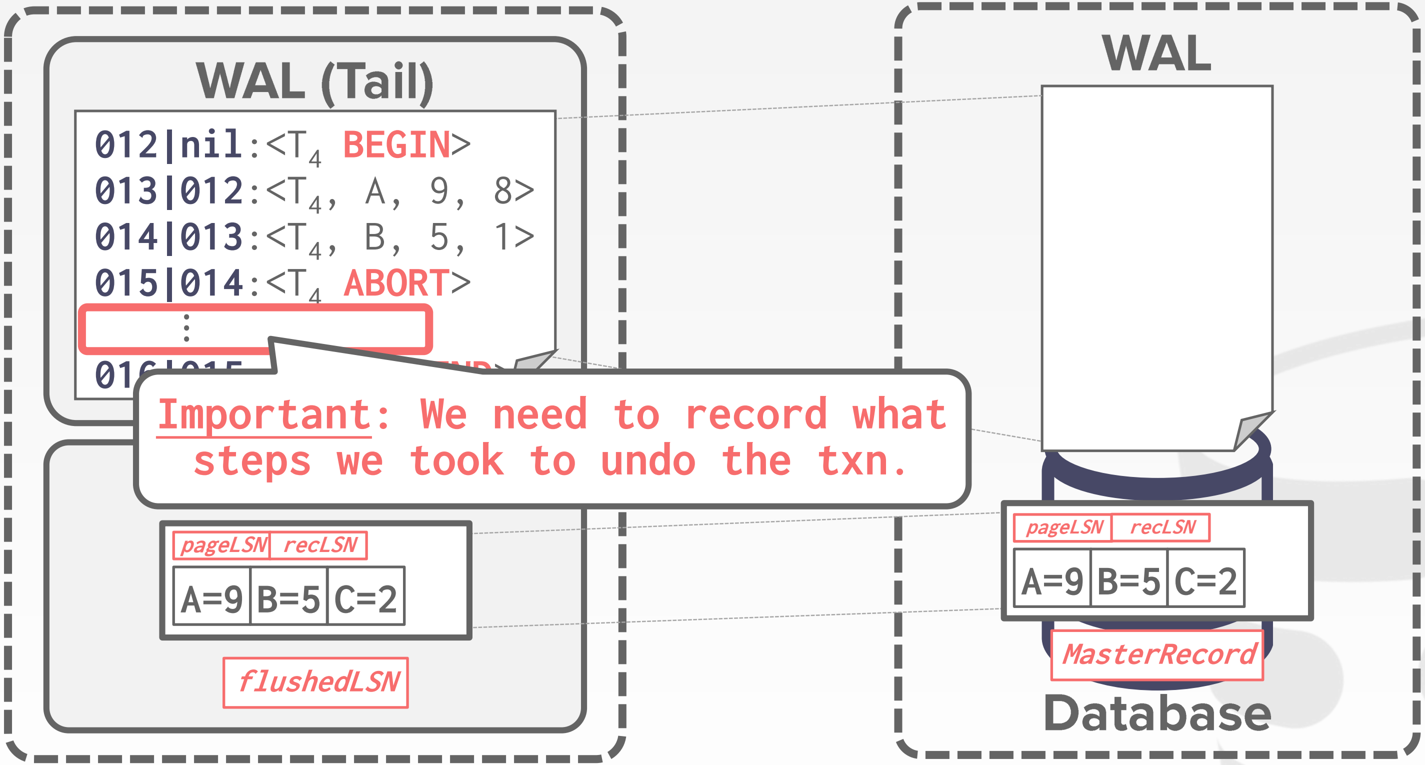
MasterRecord

Database

TRANSACTION COMMIT



TRANSACTION COMMIT



```
012|nil:<T4 BEGIN>
013|012:<T4, A, 9, 8>
014|013:<T4, B, 5, 1>
015|014:<T4 ABORT>
   ⋮
```

Important: We need to record what steps we took to undo the txn.

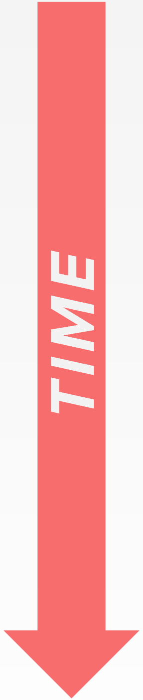
COMPENSATION LOG RECORDS

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 `undoNext` pointer (i.e., the next-to-be-undone LSN).

CLRs are added to log like any other record.

TRANSACTION ABORT – CLR EXAMPLE



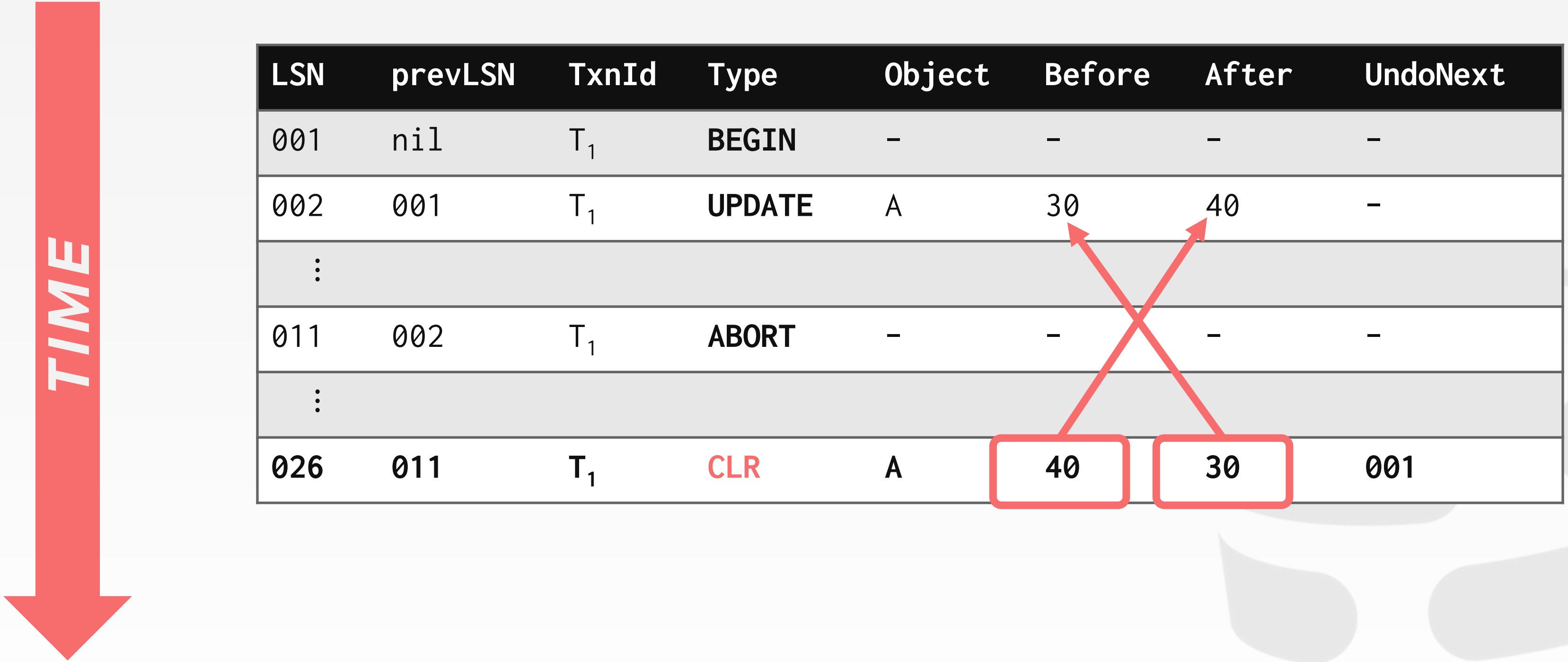
LSN	prevLSN	TxnId	Type	Object	Before	After	UndoNext
001	nil	T_1	**BEGIN**	-	-	-	-
002	001	T_1	**UPDATE**	A	30	40	-
⋮							
011	002	T_1	**ABORT**	-	-	-	-
⋮							
026	**011**	$\mathbf{T_1}$	**CLR**	**A**	**40**	**30**	**001**

TRANSACTION ABORT – CLR EXAMPLE

TIME

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

TRANSACTION ABORT – CLR EXAMPLE



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

CARNEGIE MELLON
DATABASE GROUP

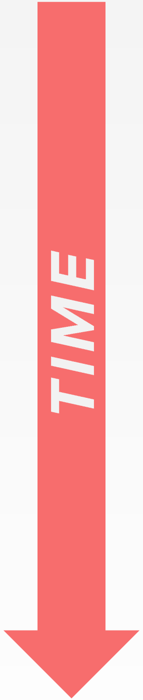
TRANSACTION ABORT – CLR EXAMPLE

TIME

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

The LSN of the next log record to be undone.

TRANSACTION ABORT – CLR EXAMPLE



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

ABORT ALGORITHM

First write an `ABORT` record to log.

Then play back updates in reverse order.
For each update:
- → Write a `CLR` entry.
- → Restore old value.

At end, write a `TXN-END` log record.

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:

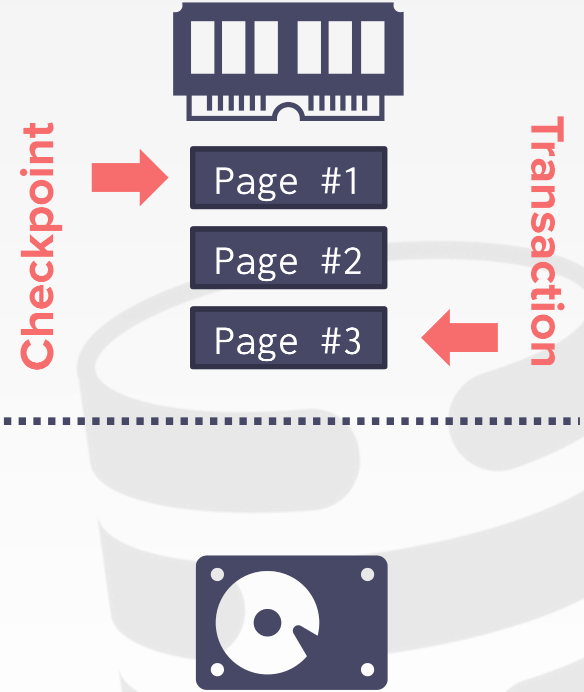
→ Halt the start of any new txns.
→ Wait until all active txns finish executing.
→ Flushes dirty pages on disk.

This is obviously bad…

BETTER CHECKPOINTS

Pause txns while the DBMS takes the checkpoint.

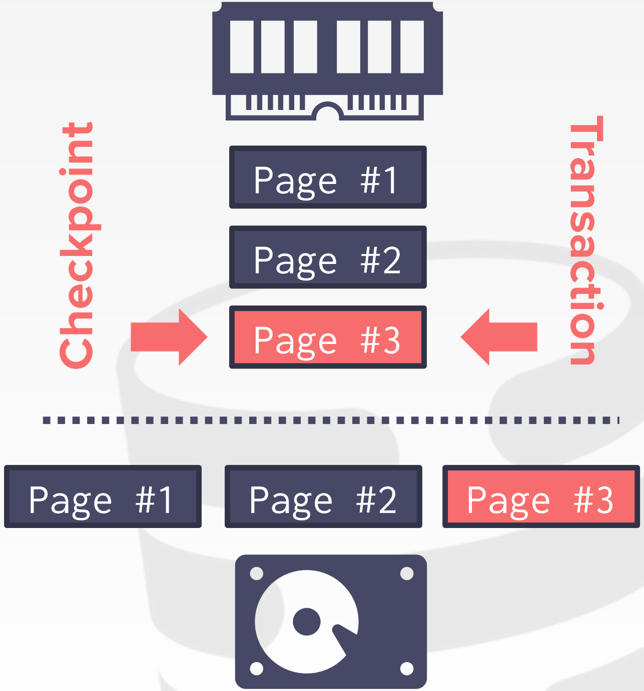
→ We don't have to wait until all txns finish before taking the checkpoint.



BETTER CHECKPOINTS

Pause txns while the DBMS takes the checkpoint.

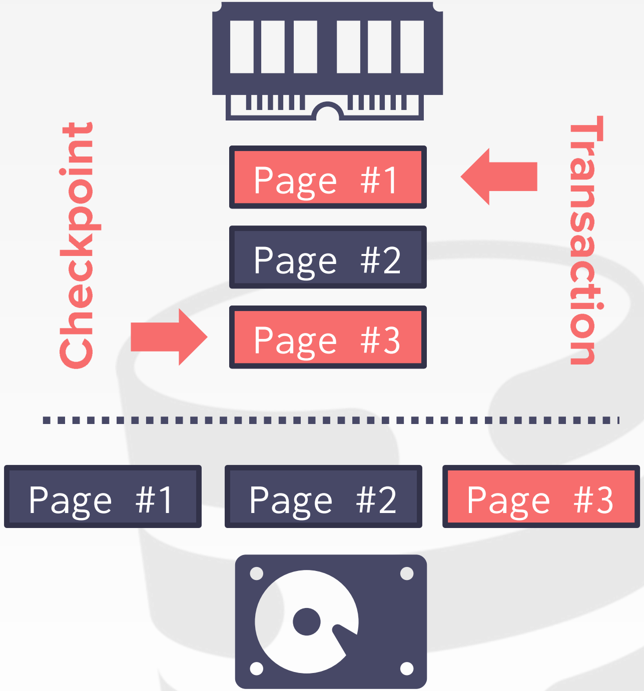
→ We don't have to wait until all txns finish before taking the checkpoint.



BETTER CHECKPOINTS

Pause txns while the DBMS takes the checkpoint.
→ We don't have to wait until all txns finish before taking the checkpoint.

We have to now record internal system state as of the beginning of the checkpoint.
→ **Active Transaction Table (ATT)**
→ **Dirty Page Table (DPT)**



ACTIVE TRANSACTION TABLE

One entry per currently active txn.
- → `txnId`: Unique txn identifier.
- → `status`: The current "mode" of the txn.
- → `lastLSN`: Most recent ***LSN*** written by txn.

Entry removed when txn commits or aborts.

Status Codes:
- → `R` → Running
- → `C` → Committing
- → `U` → Candidate for Undo

DIRTY PAGE TABLE

Keep track of which pages in the buffer pool contain changes from uncommitted transactions.

One entry per dirty page:

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

BETTER CHECKPOINTS

At the first checkpoint, T_2 is still running and there are two dirty pages (i.e., P_{10}, P_{12}).

WAL

<T_1 BEGIN>
...
<T_1 COMMIT>
...
<T_2, C, 100, 120>
<CHECKPOINT
ATT={T_2},
DPT={P_{10},P_{12}}>
<T_3 START>
<T_2 COMMIT>
<T_3, A, 200, 400>
<CHECKPOINT
ATT={T_3},
DPT={P_{10},P_{33}}>
<T_3, B, 10, 12>

BETTER CHECKPOINTS

At the first checkpoint, T_2 is still running and there are two dirty pages (i.e., P_{10}, P_{12}).

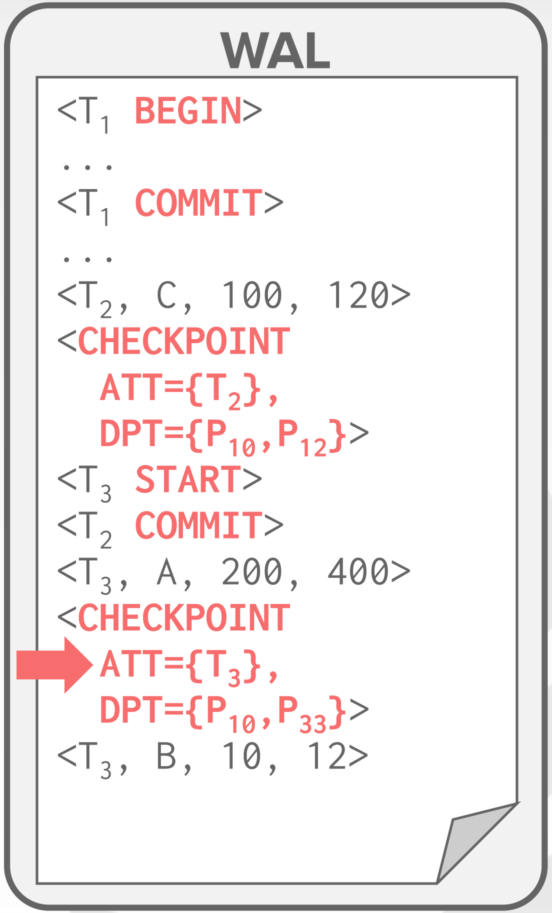
WAL

<T_1 BEGIN>
. . .
<T_1 COMMIT>
. . .
<T_2, C, 100, 120>
<CHECKPOINT
  ATT={T_2},
  DPT={P_{10},P_{12}}> ⬅
<T_3 START>
<T_2 COMMIT>
<T_3, A, 200, 400>
<CHECKPOINT
  ATT={T_3},
  DPT={P_{10},P_{33}}>
<T_3, B, 10, 12>

BETTER CHECKPOINTS

At the first checkpoint, T_2 is still running and there are two dirty pages (i.e., P_{10}, P_{12}).

At the second checkpoint, T_3 is active and there are two dirty pages (i.e., P_{10}, P_{33}).



WAL

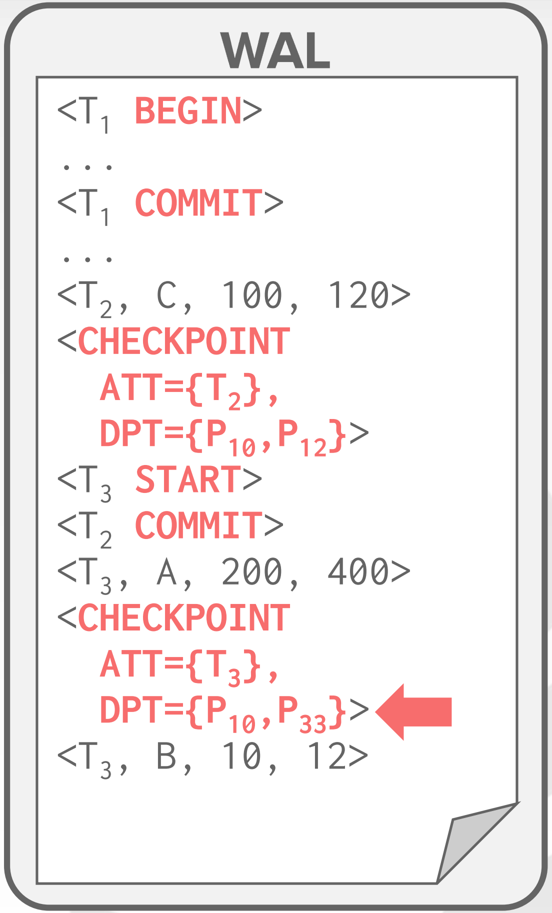
<T_1 BEGIN>
...
<T_1 COMMIT>
...
<T_2, C, 100, 120>
<CHECKPOINT
ATT={T_2},
DPT={P_{10},P_{12}}>
<T_3 START>
<T_2 COMMIT>
<T_3, A, 200, 400>
<CHECKPOINT
ATT={T_3},
DPT={P_{10},P_{33}}>
<T_3, B, 10, 12>

BETTER CHECKPOINTS

At the first checkpoint, T_2 is still running and there are two dirty pages (i.e., P_{10}, P_{12}).

At the second checkpoint, T_3 is active and there are two dirty pages (i.e., P_{10}, P_{33}).

This still isn't great because we have to stall all txns during checkpoint…



WAL

<T_1 BEGIN>
. . .
<T_1 COMMIT>
. . .
<T_2, C, 100, 120>
<CHECKPOINT
ATT={T_2},
DPT={P_{10},P_{12}}>
<T_3 START>
<T_2 COMMIT>
<T_3, A, 200, 400>
<CHECKPOINT
ATT={T_3},
DPT={P_{10},P_{33}}>
<T_3, B, 10, 12>

FUZZY CHECKPOINTS

A **fuzzy checkpoint** is where the DBMS allows other txns to continue the run.

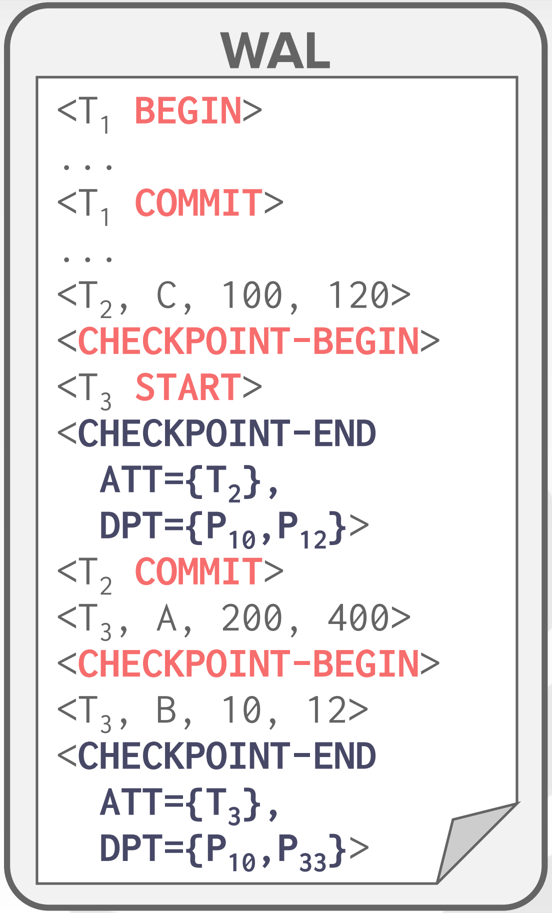
New log records to track checkpoint boundaries:

- → `CHECKPOINT-BEGIN`: Indicates start of checkpoint
- → `CHECKPOINT-END`: Contains **ATT** + **DPT**.

FUZZY CHECKPOINTS

The ***LSN*** of the `CHECKPOINT-BEGIN` record is written to the `MasterRecord` entry.

Any txn that starts after the checkpoint is excluded from the txn table listing.



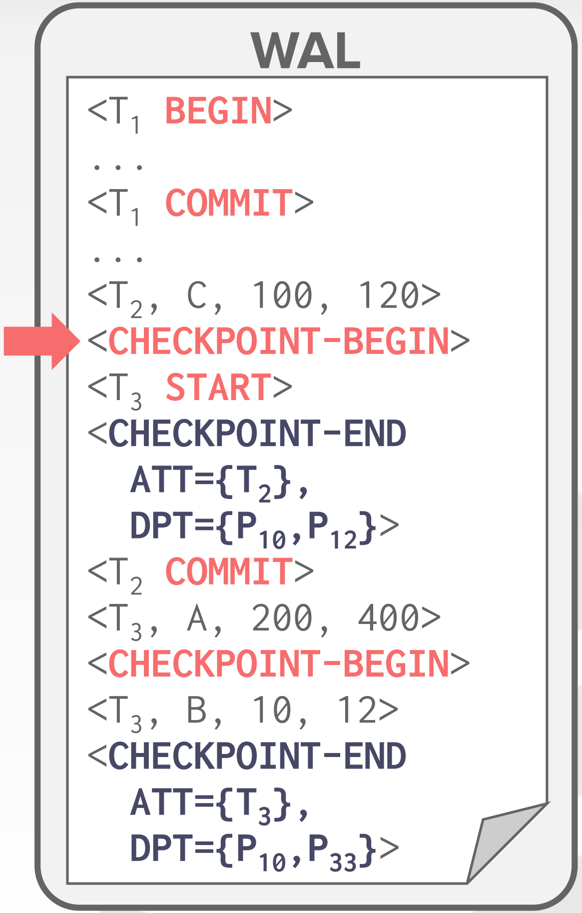
WAL

<T_1 BEGIN>
...
<T_1 COMMIT>
...
<T_2, C, 100, 120>
<CHECKPOINT-BEGIN>
<T_3 START>
<CHECKPOINT-END
  ATT={T_2},
  DPT={P_{10},P_{12}}>
<T_2 COMMIT>
<T_3, A, 200, 400>
<CHECKPOINT-BEGIN>
<T_3, B, 10, 12>
<CHECKPOINT-END
  ATT={T_3},
  DPT={P_{10},P_{33}}>

FUZZY CHECKPOINTS

The ***LSN*** of the **CHECKPOINT-BEGIN** record is written to the **MasterRecord** entry.

Any txn that starts after the checkpoint is excluded from the txn table listing.



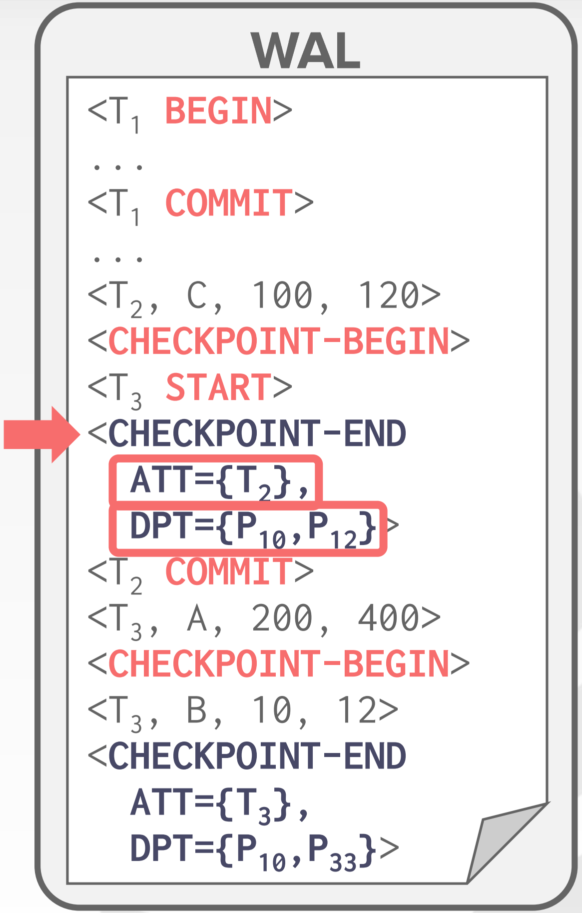
WAL

<T_1 BEGIN>
...
<T_1 COMMIT>
...
<T_2, C, 100, 120>
<CHECKPOINT-BEGIN>
<T_3 START>
<CHECKPOINT-END
  ATT={T_2},
  DPT={P_{10},P_{12}}>
<T_2 COMMIT>
<T_3, A, 200, 400>
<CHECKPOINT-BEGIN>
<T_3, B, 10, 12>
<CHECKPOINT-END
  ATT={T_3},
  DPT={P_{10},P_{33}}>

FUZZY CHECKPOINTS

The ***LSN*** of the **CHECKPOINT-BEGIN** record is written to the **MasterRecord** entry.

Any txn that starts after the checkpoint is excluded from the txn table listing.



WAL

<T_1 **BEGIN**>
...
<T_1 **COMMIT**>
...
<T_2, C, 100, 120>
<**CHECKPOINT-BEGIN**>
<T_3 **START**>
<**CHECKPOINT-END**
ATT={T_2},
DPT={P_{10},P_{12}}>
<T_2 **COMMIT**>
<T_3, A, 200, 400>
<**CHECKPOINT-BEGIN**>
<T_3, B, 10, 12>
<**CHECKPOINT-END**
ATT={T_3},
DPT={P_{10},P_{33}}>

ARIES – RECOVERY PHASES

Analysis: Read the WAL to identify dirty pages in the buffer pool and active txns at the time of the crash.

Redo: Repeat all actions starting from an appropriate point in the log.

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

ARIES – OVERVIEW

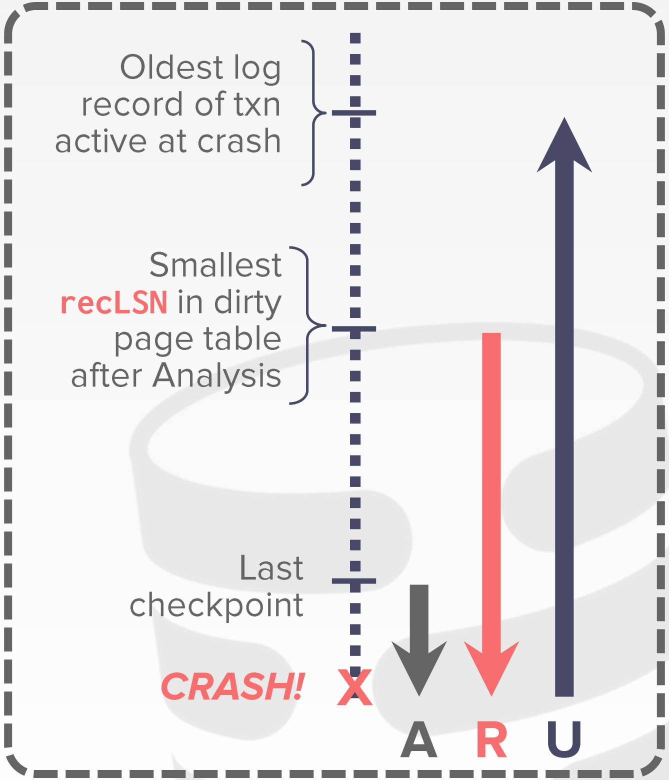
Start from last checkpoint found via **`MasterRecord`**.

Three phases.

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

→ Redo: Repeat **all** actions.

→ Undo: Reverse effects of failed txns.



ANALYSIS PHASE

Re-establish knowledge of state at checkpoint.

→ Examine **ATT** and **DPT** stored in the checkpoint.

ANALYSIS PHASE

Scan log forward from checkpoint.

If you find a `TXN-END` record, remove its txn from **ATT**.

All other records:

- → Add txn to **ATT** with status `UNDO`.
- → On commit, change txn status to `COMMIT`.

For `UPDATE` records:

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

ANALYSIS PHASE

At end of the Analysis Phase:

- → **ATT** tells the DBMS which txns were active at time of crash.
- → **DPT** tells the DBMS which dirty pages might not have made it to disk.

ANALYSIS PHASE EXAMPLE

WAL

010:<CHECKPOINT-BEGIN>

⋮

020:<T_{96}, A→P_{33}, 10, 15>

⋮

030:<CHECKPOINT-END
ATT={T_{96},T_{97}},
DPT={P_{20},P_{33}}>

⋮

040:<T_{96} COMMIT>

⋮

050:<T_{96} TXN-END>

⋮

CRASH!

LSN	ATT	DPT
010		
020		
030		
040		
050		

ANALYSIS PHASE EXAMPLE

WAL

010:<CHECKPOINT-BEGIN>

⋮

020:<T_{96}, A→P_{33}, 10, 15>

⋮

030:<CHECKPOINT-END
ATT={T_{96},T_{97}},
DPT={P_{20},P_{33}}>

⋮

040:<T_{96} COMMIT>

⋮

050:<T_{96} TXN-END>

⋮

CRASH!

LSN	ATT	DPT
010		
020	(T_{96}, U)	
030		
040		
050		

ANALYSIS PHASE EXAMPLE

WAL

010:<CHECKPOINT-BEGIN>
⋮
020:<T_{96}, A→P_{33}, 10, 15>
⋮
030:<CHECKPOINT-END
ATT={T_{96},T_{97}},
DPT={P_{20},P_{33}}>
⋮
040:<T_{96} COMMIT>
⋮
050:<T_{96} TXN-END>
⋮
CRASH!

(TxnId, Status)

LSN	ATT
010	
020	(T_{96}, U)
030	
040	
050	

CARNEGIE MELLON
DATABASE GROUP

ANALYSIS PHASE EXAMPLE

WAL

010:<CHECK

⋮

020:<T_{96}, A→P_{33}, 10, 15>

⋮

030:<CHECKPOINT-END
ATT={T_{96},T_{97}},
DPT={P_{20},P_{33}}>

⋮

040:<T_{96} COMMIT>

⋮

050:<T_{96} TXN-END>

⋮

CRASH!

Modify A in page P_{33}

	ATT	DPT
010		
020	(T_{96}, U)	(P_{33})
030		
040		
050		

ANALYSIS PHASE EXAMPLE

WAL

010:<CHECKPOINT-BEGIN>
⋮
020:<T_{96}, A→P_{33}, 10, 15>
⋮
030:<CHECKPOINT-END
ATT={T_{96},T_{97}},
DPT={P_{20},P_{33}}>
⋮
040:<T_{96} COMMIT>
⋮
050:<T_{96} TXN-END>
⋮
CRASH!

LSN	ATT	DPT
010		
020	(T_{96}, U)	(P_{33})
030	(T_{96},U), (T_{97},U)	(P_{33}), (P_{20})
040		
050		

ANALYSIS PHASE EXAMPLE

WAL

010:<CHECKPOINT-BEGIN>
⋮
020:<T_{96}, A→P_{33}, 10, 15>
⋮
030:<CHECKPOINT-END
ATT={T_{96},T_{97}},
DPT={P_{20},P_{33}}>
⋮
040:<T_{96} COMMIT>
⋮
050:<T_{96} TXN-END>
⋮
CRASH!

LSN	ATT	DPT
010		
020	(T_{96}, U)	(P_{33})
030	(T_{96},U), (T_{97},U)	(P_{33}), (P_{20})
040	(T_{96},C), (T_{97},U)	(P_{33}), (P_{20})
050		

ANALYSIS PHASE EXAMPLE

WAL

010:<CHECKPOINT-BEGIN>
⋮
020:<T_{96}, A→P_{33}, 10, 15>
⋮
030:<CHECKPOINT-END
ATT={T_{96},T_{97}},
DPT={P_{20},P_{33}}>
⋮
040:<T_{96} COMMIT>
⋮
050:<T_{96} TXN-END>
⋮
CRASH!

LSN	ATT	DPT
010		
020	(T_{96}, U)	(P_{33})
030	(T_{96},U), (T_{97},U)	(P_{33}), (P_{20})
040	(T_{96},C), (T_{97},U)	(P_{33}), (P_{20})
050	(T_{97},U)	(P_{33}), (P_{20})

REDO PHASE

The goal is to repeat history to reconstruct state at the moment of the crash:

- → Reapply all updates (even aborted txns!) and redo **CLRs**.

We can 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:

- → Affected page is not in the **DPT**, or
- → Affected page is in **DPT** but that record's ***LSN*** is greater than smallest `recLSN`, or
- → Affected `pageLSN` (on disk) ≥ `LSN`

REDO PHASE

To redo an action:

- → Reapply logged action.
- → Set **pageLSN** to log record's ***LSN***.
- → No additional logging, no forcing!

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

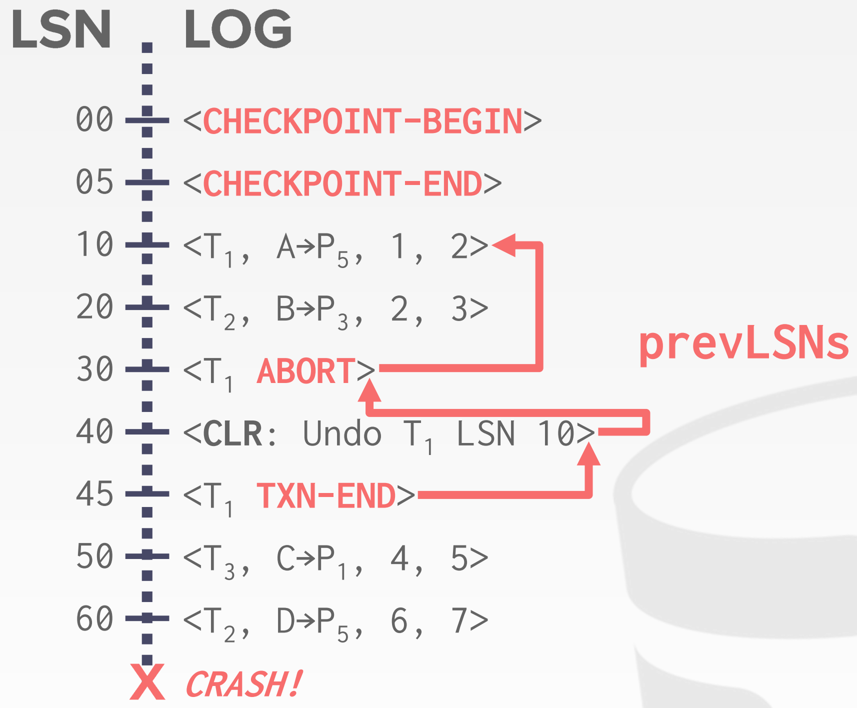
Undo all txns that were active at the time of crash ("loser" txns).

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

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

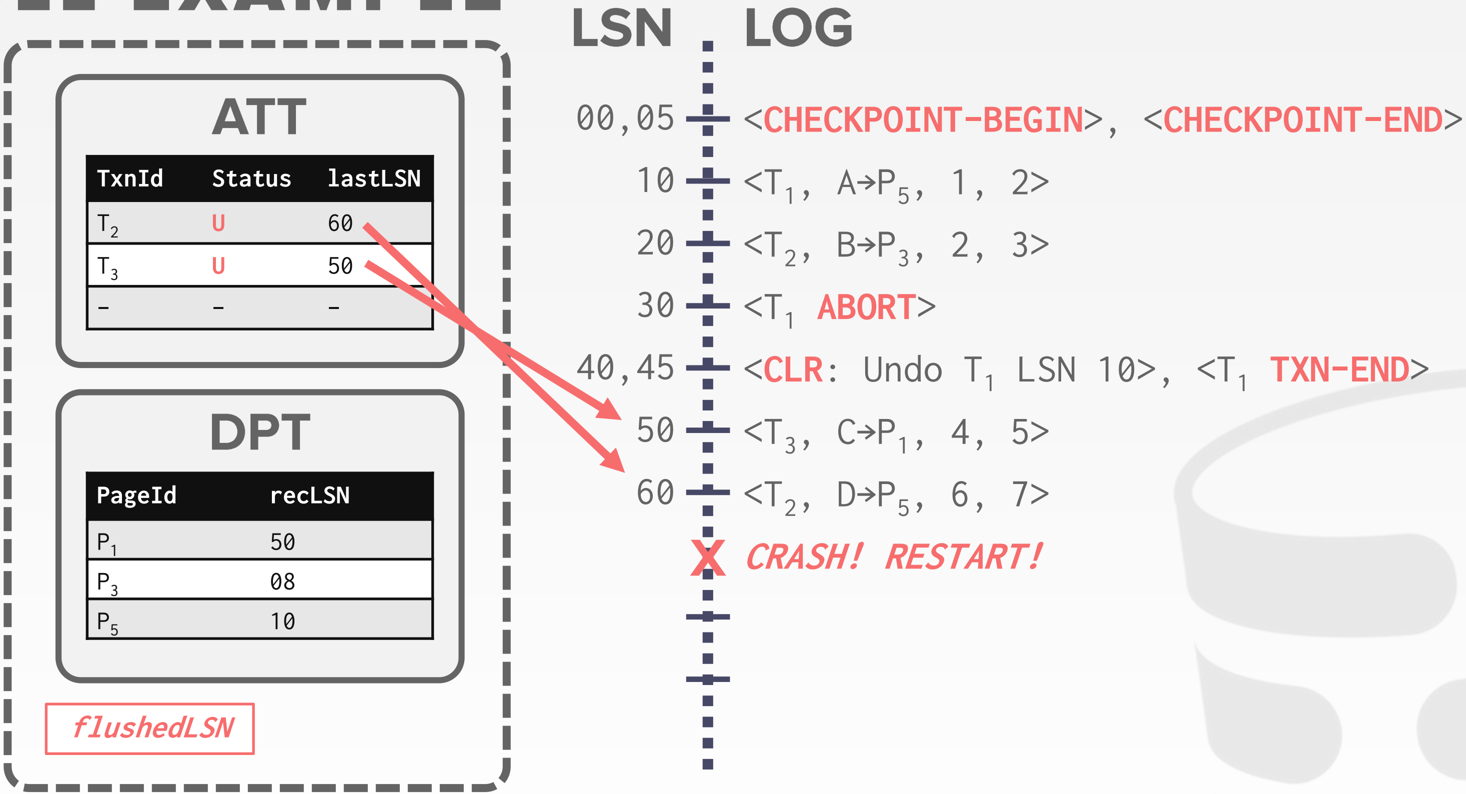
Write a **CLR** for every modification.

FULL EXAMPLE



LSN	LOG
00	<**CHECKPOINT-BEGIN**>
05	<**CHECKPOINT-END**>
10	<T_1, A→P_5, 1, 2>
20	<T_2, B→P_3, 2, 3>
30	<T_1 **ABORT**>
40	<**CLR**: Undo T_1 LSN 10>
45	<T_1 **TXN-END**>
50	<T_3, C→P_1, 4, 5>
60	<T_2, D→P_5, 6, 7>
X	*CRASH!*

FULL EXAMPLE



ATT

TxnId	Status	lastLSN
T_2	U	60
T_3	U	50
–	–	–

DPT

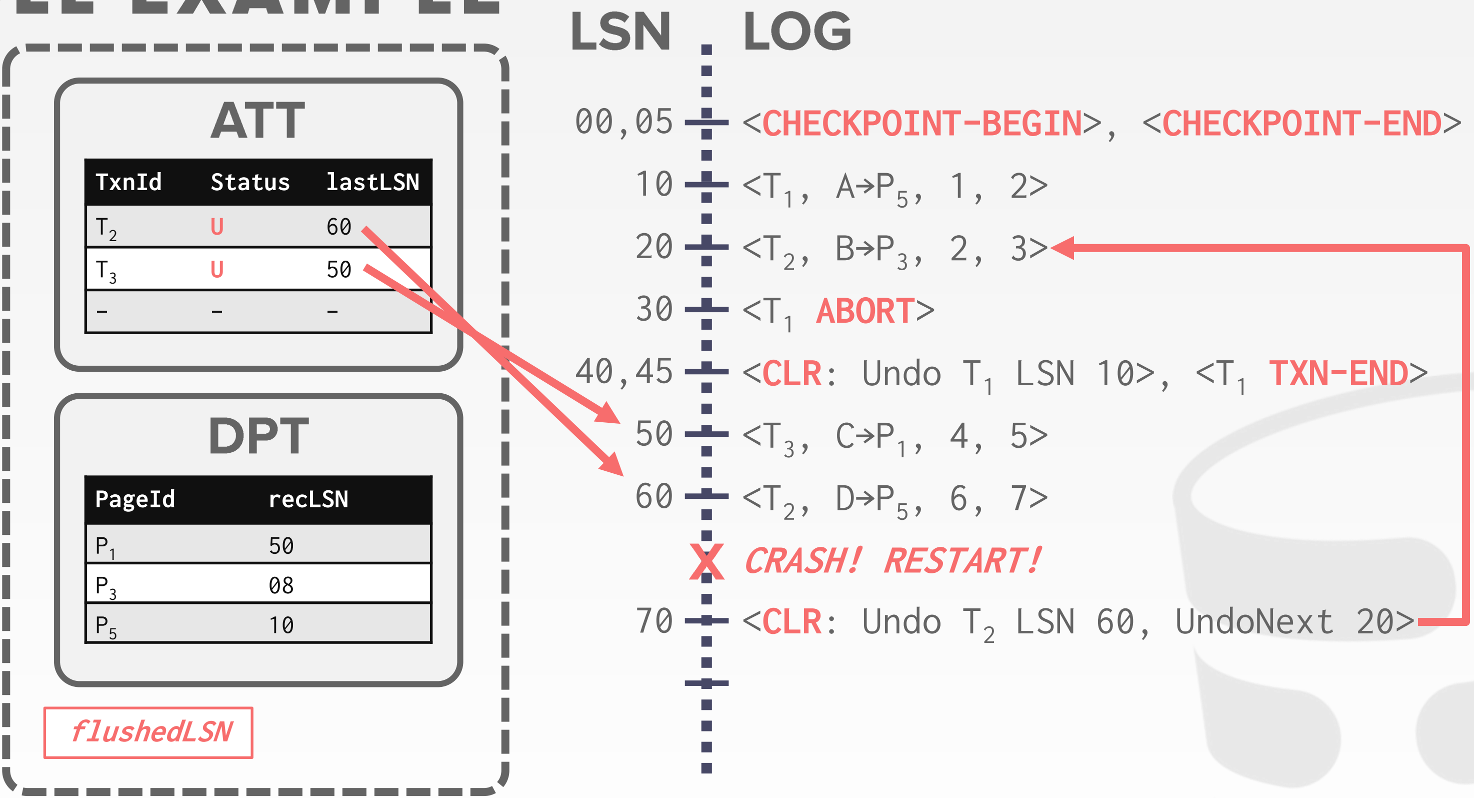
PageId	recLSN
P_1	50
P_3	08
P_5	10

flushedLSN

LSN **LOG**

00,05 <CHECKPOINT-BEGIN>, <CHECKPOINT-END>
10 <T_1, A→P_5, 1, 2>
20 <T_2, B→P_3, 2, 3>
30 <T_1 ABORT>
40,45 <CLR: Undo T_1 LSN 10>, <T_1 TXN-END>
50 <T_3, C→P_1, 4, 5>
60 <T_2, D→P_5, 6, 7>
X *CRASH! RESTART!*

FULL EXAMPLE



ATT

TxnId	Status	lastLSN
T_2	U	60
T_3	U	50
–	–	–

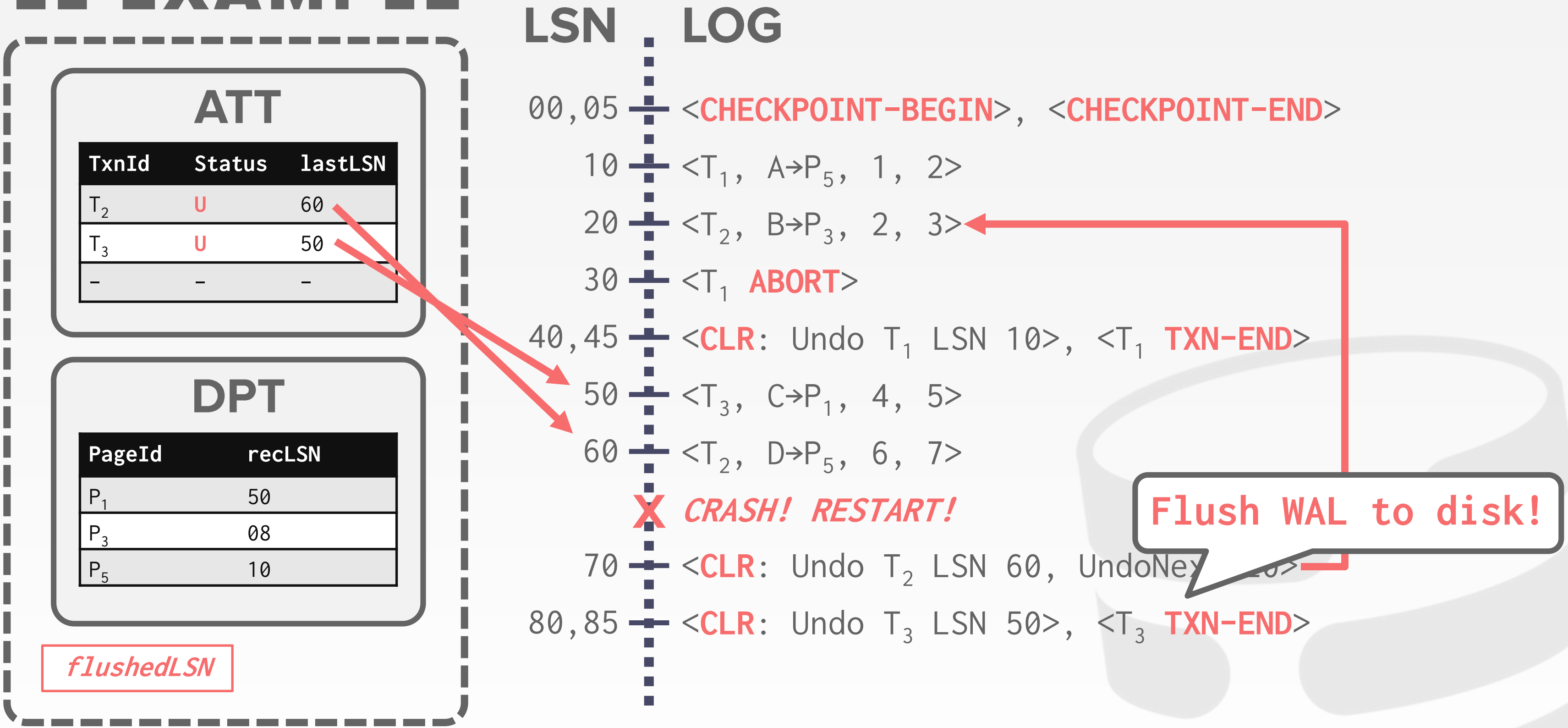
DPT

PageId	recLSN
P_1	50
P_3	08
P_5	10

flushedLSN

LSN	LOG
00,05	<**CHECKPOINT-BEGIN**>, <**CHECKPOINT-END**>
10	<T_1, A→P_5, 1, 2>
20	<T_2, B→P_3, 2, 3>
30	<T_1 **ABORT**>
40,45	<**CLR**: Undo T_1 LSN 10>, <T_1 **TXN-END**>
50	<T_3, C→P_1, 4, 5>
60	<T_2, D→P_5, 6, 7>
X	***CRASH! RESTART!***
70	<**CLR**: Undo T_2 LSN 60, UndoNext 20>

FULL EXAMPLE



ATT

TxnId	Status	lastLSN
T_2	U	60
T_3	U	50
–	–	–

DPT

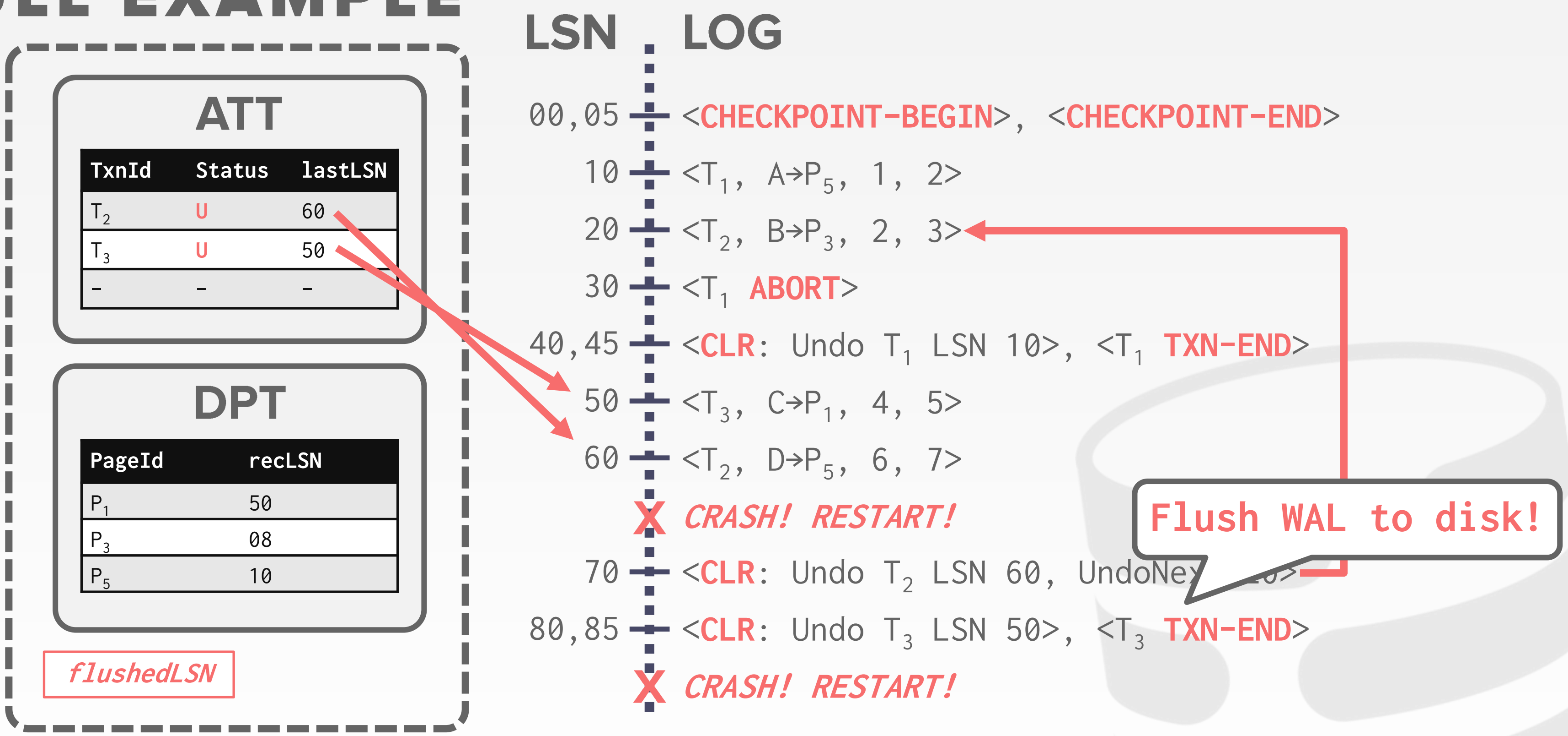
PageId	recLSN
P_1	50
P_3	08
P_5	10

flushedLSN

LSN	LOG
00,05	<CHECKPOINT-BEGIN>, <CHECKPOINT-END>
10	<T_1, A→P_5, 1, 2>
20	<T_2, B→P_3, 2, 3>
30	<T_1 ABORT>
40,45	<CLR: Undo T_1 LSN 10>, <T_1 TXN-END>
50	<T_3, C→P_1, 4, 5>
60	<T_2, D→P_5, 6, 7>
	X *CRASH! RESTART!*
70	<CLR: Undo T_2 LSN 60, UndoNex 20>
80,85	<CLR: Undo T_3 LSN 50>, <T_3 TXN-END>

Flush WAL to disk!

FULL EXAMPLE



ATT

TxnId	Status	lastLSN
T_2	U	60
T_3	U	50
–	–	–

DPT

PageId	recLSN
P_1	50
P_3	08
P_5	10

flushedLSN

LSN	LOG
00,05	<CHECKPOINT-BEGIN>, <CHECKPOINT-END>
10	<T_1, A→P_5, 1, 2>
20	<T_2, B→P_3, 2, 3>
30	<T_1 ABORT>
40,45	<CLR: Undo T_1 LSN 10>, <T_1 TXN-END>
50	<T_3, C→P_1, 4, 5>
60	<T_2, D→P_5, 6, 7>
	CRASH! RESTART!
70	<CLR: Undo T_2 LSN 60, UndoNex 20>
80,85	<CLR: Undo T_3 LSN 50>, <T_3 TXN-END>
	CRASH! RESTART!

Flush WAL to disk!

FULL EXAMPLE

LSN LOG

00,05 <CHECKPOINT-BEGIN>, <CHECKPOINT-END>

10 <T_1, A→P_5, 1, 2>

20 <T_2, B→P_3, 2, 3>

30 <T_1 ABORT>

40,45 <CLR: Undo T_1 LSN 10>, <T_1 TXN-END>

50 <T_3, C→P_1, 4, 5>

60 <T_2, D→P_5, 6, 7>

CRASH! RESTART!

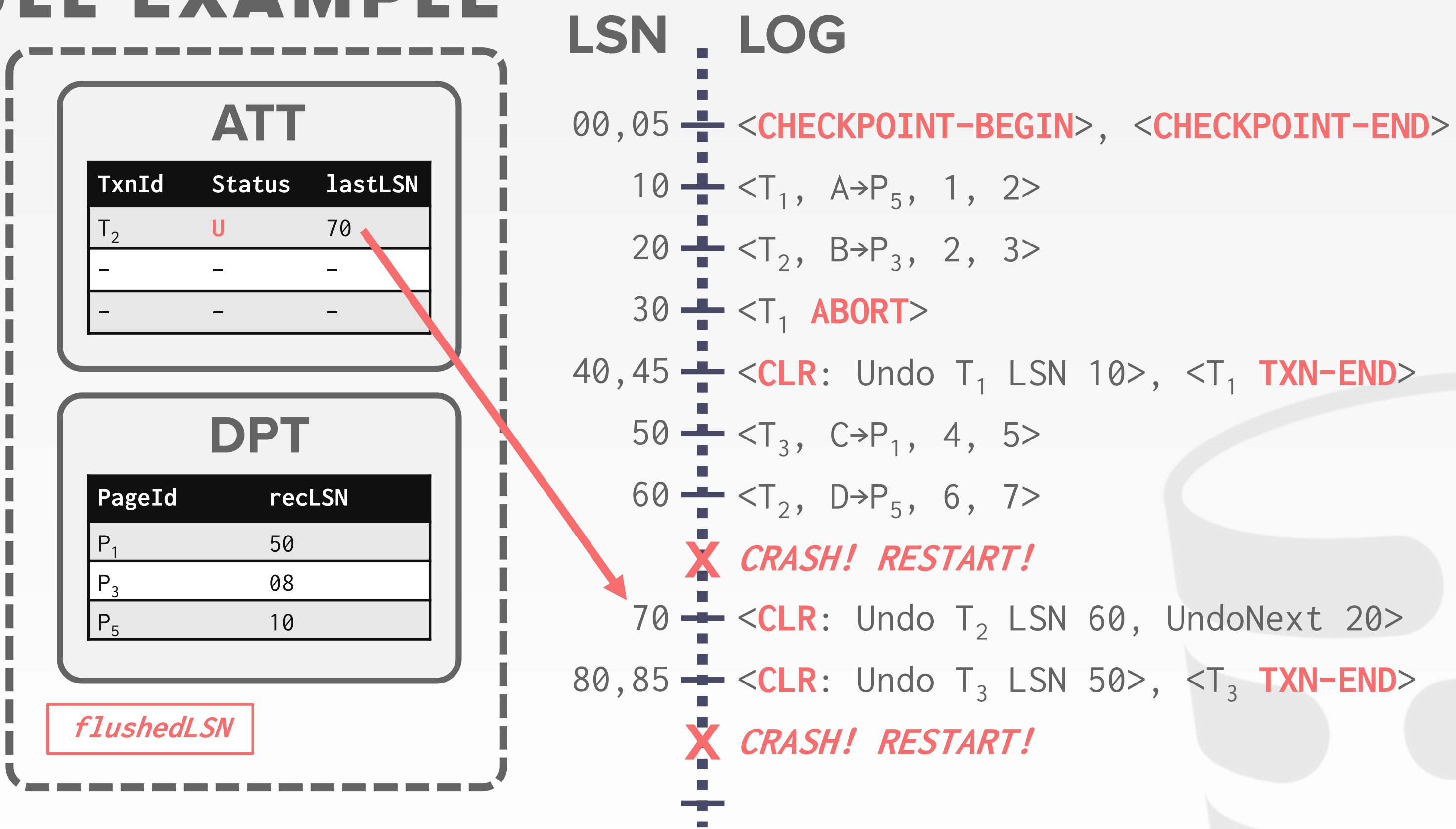
70 <CLR: Undo T_2 LSN 60, UndoNex 20>

80,85 <CLR: Undo T_3 LSN 50>, <T_3 TXN-END>

CRASH! RESTART!

Flush WAL to disk!

FULL EXAMPLE



ATT

TxnId	Status	lastLSN
T_2	U	70
–	–	–
–	–	–

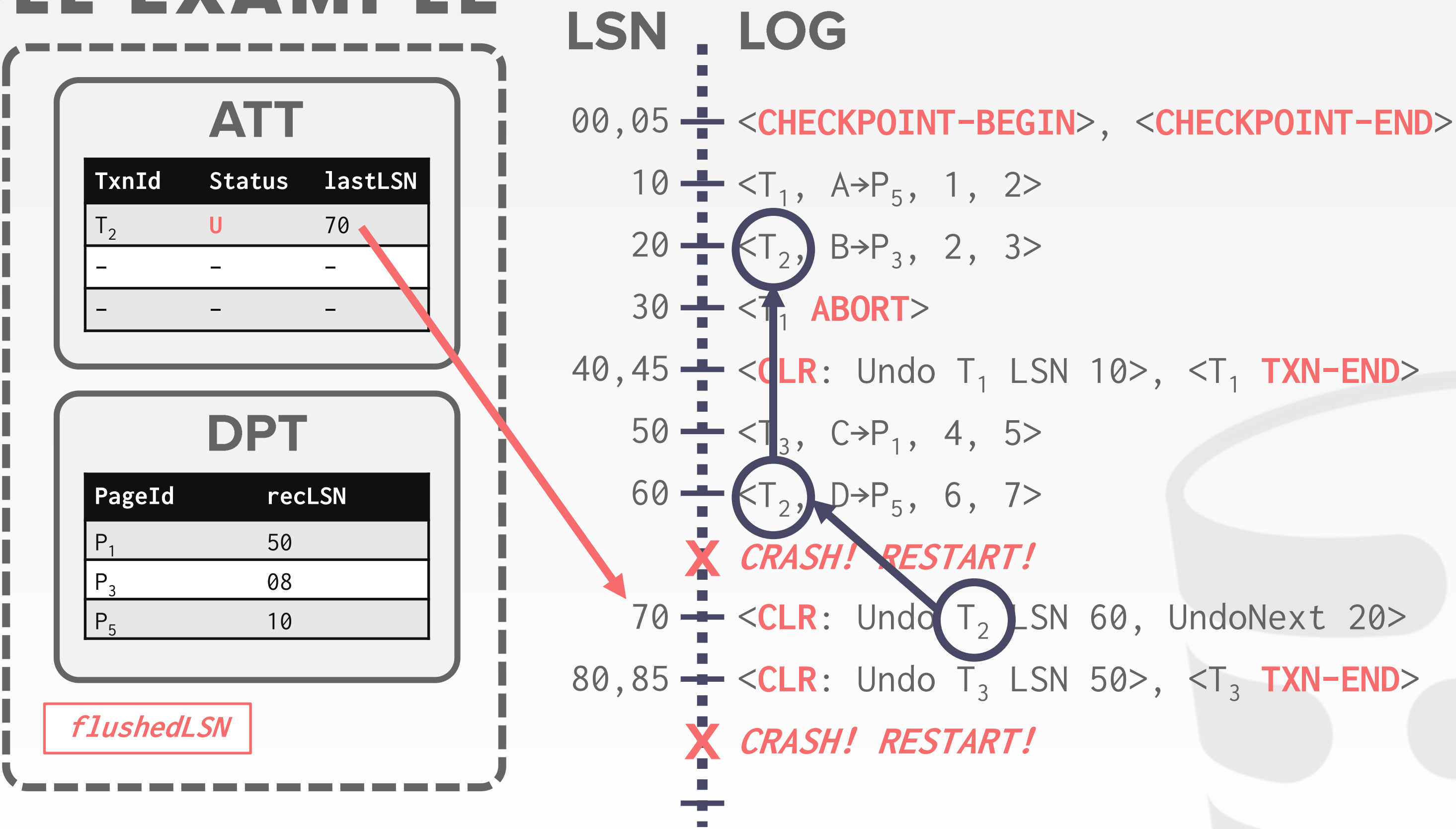
DPT

PageId	recLSN
P_1	50
P_3	08
P_5	10

flushedLSN

LSN	LOG
00,05	<CHECKPOINT-BEGIN>, <CHECKPOINT-END>
10	$<T_1, A \rightarrow P_5, 1, 2>$
20	$<T_2, B \rightarrow P_3, 2, 3>$
30	$<T_1$ ABORT>
40,45	<CLR: Undo T_1 LSN 10>, $<T_1$ TXN-END>
50	$<T_3, C \rightarrow P_1, 4, 5>$
60	$<T_2, D \rightarrow P_5, 6, 7>$
X	*CRASH! RESTART!*
70	<CLR: Undo T_2 LSN 60, UndoNext 20>
80,85	<CLR: Undo T_3 LSN 50>, $<T_3$ TXN-END>
X	*CRASH! RESTART!*

FULL EXAMPLE



ATT

TxnId	Status	lastLSN
T_2	U	70
–	–	–
–	–	–

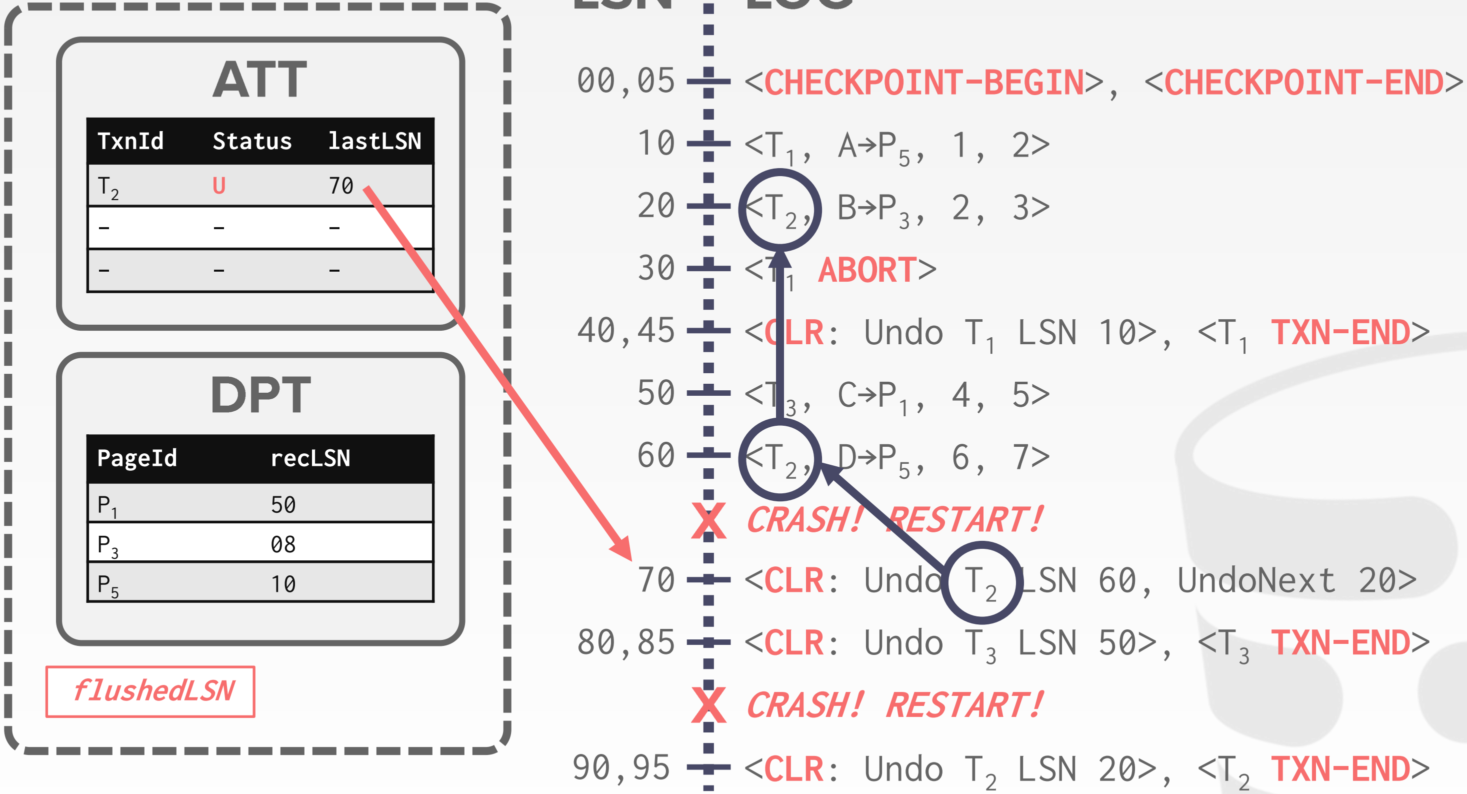
DPT

PageId	recLSN
P_1	50
P_3	08
P_5	10

flushedLSN

LSN	LOG
00,05	<CHECKPOINT-BEGIN>, <CHECKPOINT-END>
10	<T_1, A→P_5, 1, 2>
20	<T_2, B→P_3, 2, 3>
30	<T_1 ABORT>
40,45	<CLR: Undo T_1 LSN 10>, <T_1 TXN-END>
50	<T_3, C→P_1, 4, 5>
60	<T_2, D→P_5, 6, 7>
X	*CRASH! RESTART!*
70	<CLR: Undo T_2 LSN 60, UndoNext 20>
80,85	<CLR: Undo T_3 LSN 50>, <T_3 TXN-END>
X	*CRASH! RESTART!*

FULL EXAMPLE



ATT

TxnId	Status	lastLSN
T_2	U	70
–	–	–
–	–	–

DPT

PageId	recLSN
P_1	50
P_3	08
P_5	10

flushedLSN

LSN	LOG
00,05	<CHECKPOINT-BEGIN>, <CHECKPOINT-END>
10	<T_1, A→P_5, 1, 2>
20	<T_2, B→P_3, 2, 3>
30	<T_1 ABORT>
40,45	<CLR: Undo T_1 LSN 10>, <T_1 TXN-END>
50	<T_3, C→P_1, 4, 5>
60	<T_2, D→P_5, 6, 7>
	X *CRASH! RESTART!*
70	<CLR: Undo T_2 LSN 60, UndoNext 20>
80,85	<CLR: Undo T_3 LSN 50>, <T_3 TXN-END>
	X *CRASH! RESTART!*
90,95	<CLR: Undo T_2 LSN 20>, <T_2 TXN-END>

ADDITIONAL CRASH ISSUES

What happens if system crashes during the Analysis Phase? During the Redo Phase?

How do you limit the amount of work in the Redo Phase?

→ Flush asynchronously in the background.

How do you limit the amount of work in the Undo Phase?

→ Avoid long-running txns.

CONCLUSION

Mains ideas of ARIES:

- → WAL (write ahead log), **STEAL**/**NO-FORCE**
- → Fuzzy Checkpoints (snapshot of dirty page ids)
- → Redo everything since the earliest dirty page; undo "loser" transactions
- → 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 an entire single-node ACID DBMS.

So now we can talk about distributed databases!