20 Database Logging
Project #3 is due Sun Nov 22\textsuperscript{nd} @ 11:59pm.

Project #4 will be released this week.

Homework #5 will be released next week.
UPCOMING DATABASE TALKS

**FaunaDB Serverless DBMS**
→ Monday Nov 16\(^{th}\) @ 5pm ET

**Confluent ksqlDB (Kafka)**
→ Monday Nov 23\(^{rd}\) @ 5pm ET

**Microsoft SQL Server Optimizer**
→ Monday Nov 30\(^{th}\) @ 5pm ET
MOTIVATION

Schedule

$T_1$

BEGIN
R(A)
W(A)
...
COMMIT

Buffer Pool

Page
MOTIVATION

Schedule

\[ T_1 \]

BEGIN
R(A)
W(A)
\vdots
COMMIT

Buffer Pool

\begin{align*}
A = 1 \\
\end{align*}

Page A = 1

TIME

\[ CMU-DB \]

15-445/645 (Fall 2020)
MOTIVATION

Schedule

BEGIN
R(A)
W(A)
⋮
COMMIT

Buffer Pool

A=2

Page

TIME

T₁

A=1
**MOTIVATION**

**Schedule**

\[ T_1 \]

BEGIN
R(A)
W(A)
⋮
COMMIT

**Buffer Pool**

- A=1
- A=2

**Page**

TIME
BEGIN
R(A)
W(A)
\ldots
COMMIT

Buffer Pool

Page

A=1

T_1

Schedule
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’S AGENDA

Failure Classification
Buffer Pool Policies
Shadow Paging
Write-Ahead Log
Logging Schemes
Checkpoints
CRASH RECOVERY

DBMS is divided into different components based on the underlying storage device.
→ Volatile vs. Non-Volatile

We must also classify the different types of failures that the DBMS needs to handle.
FAILURE CLASSIFICATION

Type #1 – Transaction Failures
Type #2 – System Failures
Type #3 – Storage Media Failures
TRANSACTION FAILURES

Logical Errors:
→ Transaction cannot complete due to some internal error condition (e.g., integrity constraint violation).

Internal State Errors:
→ DBMS must terminate an active transaction due to an error condition (e.g., deadlock).
SYSTEM FAILURES

Software Failure:
→ Problem with the OS or DBMS implementation (e.g., uncaught divide-by-zero exception).

Hardware Failure:
→ The computer hosting the DBMS crashes (e.g., power plug gets pulled).
→ Fail-stop Assumption: Non-volatile storage contents are assumed to not be corrupted by system crash.
Non-Repairable Hardware Failure:
→ A head crash or similar disk failure destroys all or part of non-volatile storage.
→ Destruction is assumed to be detectable (e.g., disk controller use checksums to detect failures).

No DBMS can recover from this! Database must be restored from archived version.
OBSERVATION

The primary storage location of the database is on non-volatile storage, but this is much slower than volatile storage.

Use volatile memory for faster access:
→ First copy target record into memory.
→ Perform the writes in memory.
→ Write dirty records back to disk.
**OBSERVATION**

The DBMS needs to ensure the following guarantees:

→ The changes for any txn are durable once the DBMS has told somebody that it committed.

→ No partial changes are durable if the txn aborted.
UNDO VS. REDO

**Undo**: The process of removing the effects of an incomplete or aborted txn.

**Redo**: The process of re-instating the effects of a committed txn for durability.

How the DBMS supports this functionality depends on how it manages the buffer pool...
Buffer Pool

Schedule

\[
\begin{align*}
T_1 & : \\
& \text{BEGIN} \\
& \text{R(A)} \\
& \text{W(A)} \\
& \vdots \\
& \text{ABORT} \\
T_2 & : \\
& \text{BEGIN} \\
& \text{R(B)} \\
& \text{W(B)} \\
& \text{COMMIT}
\end{align*}
\]
BUFFER POOL

Schedule

BEGIN
R(A)
W(A)

BEGIN
R(B)
W(B)
COMMIT

::
ABORT

Buffer Pool

A=3  B=9  C=7

A=1  B=9  C=7
BUFFER POOL

Schedule

T₁  T₂

BEGIN
R(A)
W(A)

BEGIN
R(B)
W(B)
COMMIT

ABORT

Buffer Pool

A=3 B=9 C=7

A=1 B=9 C=7
Buffer Pool

Schedule

TIME

T₁

BEGIN
R(A)
W(A)

BEGIN
R(B)
W(B)
COMMIT

⋯
ABORT

T₂

Buffer Pool

A=3  B=8  C=7

A=1  B=9  C=7

(3x3) 15 - 445/645 (Fall 2020)
Do we force $T_2$’s changes to be written to disk?
Is $T_1$ allowed to overwrite A even though it has not committed?

Do we force $T_2$’s changes to be written to disk?

BEGIN
R(B)
W(B)
COMMIT

ABORT
Is $T_1$ allowed to overwrite $A$ even though it has not committed?

Do we force $T_2$'s changes to be written to disk?

BEGIN
R(B)
W(B)
COMMIT

$T_1$...
W($A$)

$T_2$...
What happens when we need to rollback $T_1$?
STEAL POLICY

Whether the DBMS allows an uncommitted txn to overwrite the most recent committed value of an object in non-volatile storage.

**STEAL**: Is allowed.

**NO-STEAL**: Is not allowed.
FORCE POLICY

Whether the DBMS requires that all updates made by a txn are reflected on non-volatile storage before the txn can commit.

FORCE: Is required.
NO-FORCE: Is not required.
NO-STEAL + FORCE

Schedule

TIME

T₁

BEGIN
R(A)
W(A)

⋮

ABORT

T₂

BEGIN
R(B)
W(B)
COMMIT

Buffer Pool

A=1 B=9 C=7

A=1 B=9 C=7

BEGIN
R(A)
W(A)

⋮

ABORT

BEGIN
R(B)
W(B)
COMMIT
NO-STEAL + FORCE

Schedule

Begin
R(A)
W(A)

\vdots

ABORT

Begin
R(B)
W(B)
Commit

Buffer Pool

A=3  B=9  C=7

A=1  B=9  C=7

TIME

T_1

T_2
NO-STEAL + FORCE

Schedule

\[\begin{align*}
T_1 &: \text{BEGIN R(A) W(A)} \\
T_2 &: \text{BEGIN R(B) W(B) COMMIT} \\
&: \text{ABORT}
\end{align*}\]

Buffer Pool

- A=1
- B=9
- C=7

- A=3
- B=9
- C=7

Time Arrow
NO-STEAL + FORCE

Schedule

\[ T_1 \quad T_2 \]

BEGIN
R(A)
W(A)

BEGIN
R(B)
W(B)
COMMIT

\[ \vdots \]

ABORT

Buffer Pool

A=1  B=9  C=7

A=3  B=8  C=7
NO-STEAL + FORCE

Schedule

\[ T_1 \]
BEGIN
R(A)
W(A)

\[ T_2 \]
BEGIN
R(B)
W(B)
COMMIT

Buffer Pool

A=3  B=8  C=7

FORCE means that \( T_2 \) changes must be written to disk at this point.
NO-STEAL + FORCE

Schedule

NO-STEAL means that $T_1$ changes cannot be written to disk yet.

FORCE means that $T_2$ changes must be written to disk at this point.
**NO-STEAL + FORCE**

Schedule

<table>
<thead>
<tr>
<th>Time</th>
<th>Operation</th>
<th>ロック</th>
<th>書き込み</th>
</tr>
</thead>
<tbody>
<tr>
<td>T1</td>
<td>BEGIN R(B) W(B) ABORT</td>
<td>A=3 B=8 C=7</td>
<td></td>
</tr>
<tr>
<td>T2</td>
<td>BEGIN R(B) W(B) COMMIT</td>
<td>A=1 B=8 C=7</td>
<td></td>
</tr>
</tbody>
</table>

**NO-STEAL** means that $T_1$ changes cannot be written to disk yet.

**FORCE** means that $T_2$ changes must be written to disk at this point.
NO-STEAL + FORCE

Schedule

\[ T_1 \]
BEGIN
R(A)
W(A)
\[ \vdots \]
ABORT

\[ T_2 \]
BEGIN
R(B)
W(B)
COMMIT

Now it's trivial to rollback \( T_1 \)

Buffer Pool

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>A=3</td>
<td>B=8</td>
<td>C=7</td>
</tr>
</tbody>
</table>

A=1
B=8
C=7
This approach is the easiest to implement:

→ Never have to undo changes of an aborted txn because the changes were not written to disk.

→ Never have to redo changes of a committed txn because all the changes are guaranteed to be written to disk at commit time (assuming atomic hardware writes).

Previous example cannot support write sets that exceed the amount of physical memory available.
SHADOW PAGING

Maintain two separate copies of the database:
→ **Master**: Contains only changes from committed txns.
→ **Shadow**: Temporary database with changes made from uncommitted txns.

Txns only make updates in the shadow copy.
When a txn commits, atomically switch the shadow to become the new master.

Buffer Pool Policy: **NO-STEAL + FORCE**
SHADOW PAGING – EXAMPLE

Memory

Disk

DB Root

Database Root
SHADOW PAGING

To install the updates, overwrite the root so it points to the shadow, thereby swapping the master and shadow:

→ Before overwriting the root, none of the txn's updates are part of the disk-resident database
→ After overwriting the root, all the txn's updates are part of the disk-resident database.

Source: The Great Phil Bernstein
SHADOW PAGING – EXAMPLE

Memory

1
2
3
4

Master Page Table

DB Root

Disk

Database Root

txn $T_1$
**SHADOW PAGING – EXAMPLE**

**Memory**
- DB Root
- Master Page Table
- Shadow Page Table

**Disk**
- Database Root

**Transaction** $T_1$
**SHADOW PAGING – EXAMPLE**

Read-only txns access the current master.

Active modifying txn updates shadow pages.

Disk

Master Page Table

Shadow Page Table

Database Root

Read-only txns access the current master.

Active modifying txn updates shadow pages.

**TXn T₁**
**SHADOW PAGING – EXAMPLE**

- **Read-only txns access the current master.**
- **Active modifying txn updates shadow pages.**

*Diagram showing the relationship between the master page table, shadow page table, and disk.*

- **Master Page Table**
  - 1
  - 2
  - 3
  - 4

- **Shadow Page Table**
  - 1
  - 2
  - 3
  - 4

- **Disk**
  - Database Root
  - Pages

- **Transation T1**
  - DB Root
  - Update to Shadow Page Table
  - Read-only txns access the current master.
Read-only txns access the current master.

Active modifying txn updates shadow pages.

Disk

Master Page Table

Shadow Page Table

Database Root

Txn $T_1$
SHADOW PAGING – EXAMPLE

Read-only txns access the current master.

Active modifying txn updates shadow pages.

Disk

Master Page Table

Shadow Page Table

DB Root

Txn \( T_1 \)

Read-only txns access the current master.

Active modifying txn updates shadow pages.
SHADOW PAGING – EXAMPLE

Read-only txns access the current master.

Active modifying txn updates shadow pages.

Disk

Database Root

Master Page Table

1
2
3
4

1
2
3
4

T_xn T_1

COMMIT

DB Root

Read-only txns access the current master.

Active modifying txn updates shadow pages.
**SHADOW PAGING – EXAMPLE**

**Read-only txns access the current master.**

**Active modifying txn updates shadow pages.**

- **Txn T₁**
  - COMMIT

---

**Disk**

- **Database Root**
  - Update
  - Master Page Table
  - Shadow Page Table
  - DB Root

- **Read-only txns access the current master.**

- **Active modifying txn updates shadow pages.**
Read-only txns access the current master.

Active modifying txn updates shadow pages.

COMMIT

Txn $T_1$
SHADOW PAGING – EXAMPLE

Memory

Disk

Update

Database Root

Txn $T_1$

COMMIT

DB Root

Shadow Page Table

1
2
3
4
SHADOW PAGING – UNDO/REDO

Supporting rollbacks and recovery is easy.

**Undo**: Remove the shadow pages. Leave the master and the DB root pointer alone.

**Redo**: Not needed at all.
SHADOW PAGING – DISADVANTAGES

Copying the entire page table is expensive:
→ Use a page table structured like a B+tree.
→ No need to copy entire tree, only need to copy paths in the tree that lead to updated leaf nodes.

Commit overhead is high:
→ Flush every updated page, page table, and root.
→ Data gets fragmented.
→ Need garbage collection.
→ Only supports one writer txn at a time or txns in a batch.
When a txn modifies a page, the DBMS copies the original page to a separate journal file before overwriting master version.

After restarting, if a journal file exists, then the DBMS restores it to undo changes from uncommitted txns.
SQLITE (PRE-2010)

When a txn modifies a page, the DBMS copies the original page to a separate journal file before overwriting master version.

After restarting, if a journal file exists, then the DBMS restores it to undo changes from uncommitted txns.
When a txn modifies a page, the DBMS copies the original page to a separate journal file before overwriting master version.

After restarting, if a journal file exists, then the DBMS restores it to undo changes from uncommitted txns.
When a txn modifies a page, the DBMS copies the original page to a separate journal file before overwriting master version.

After restarting, if a journal file exists, then the DBMS restores it to undo changes from uncommitted txns.
When a txn modifies a page, the DBMS copies the original page to a separate journal file before overwriting master version.

After restarting, if a journal file exists, then the DBMS restores it to undo changes from uncommitted txns.
When a txn modifies a page, the DBMS copies the original page to a separate journal file before overwriting master version.

After restarting, if a journal file exists, then the DBMS restores it to undo changes from uncommitted txns.
When a txn modifies a page, the DBMS copies the original page to a separate journal file before overwriting master version.

After restarting, if a journal file exists, then the DBMS restores it to undo changes from uncommitted txns.
When a txn modifies a page, the DBMS copies the original page to a separate journal file before overwriting master version.

After restarting, if a journal file exists, then the DBMS restores it to undo changes from uncommitted txns.
When a txn modifies a page, the DBMS copies the original page to a separate journal file before overwriting master version.

After restarting, if a journal file exists, then the DBMS restores it to undo changes from uncommitted txns.
Observation

Shadowing page requires the DBMS to perform writes to random non-contiguous pages on disk.

We need a way for the DBMS to convert random writes into sequential writes.
WRITE-AHEAD LOG

Maintain a log file separate from data files that contains the changes that txns make to database.
→ Assume that the log is on stable storage.
→ Log contains enough information to perform the necessary undo and redo actions to restore the database.

DBMS must write to disk the log file records that correspond to changes made to a database object before it can flush that object to disk.

Buffer Pool Policy: STEAL + NO-FORCE
The DBMS stages all a txn's log records in volatile storage (usually backed by buffer pool).

All log records pertaining to an updated page are written to non-volatile storage before the page itself is over-written in non-volatile storage.

A txn is not considered committed until all its log records have been written to stable storage.
WAL PROTOCOL

Write a `<BEGIN>` record to the log for each txn to mark its starting point.

When a txn finishes, the DBMS will:
→ Write a `<COMMIT>` record on the log
→ Make sure that all log records are flushed before it returns an acknowledgement to application.
WAL PROTOCOL

Each log entry contains information about the change to a single object:
→ Transaction Id
→ Object Id
→ Before Value (UNDO)
→ After Value (REDO)
WAL – EXAMPLE

Schedule

\( T_1 \)

BEGIN
W(A)
W(B)
::
COMMIT

WAL Buffer

\(<T_1 \text{ BEGIN}>\)

Buffer Pool

A=1 B=5 C=7

EXAMPLE
WAL – EXAMPLE

Schedule

\[ T_1 \]

BEGIN
W(A)
W(B)
\vdots
COMMIT

WAL Buffer

\[ \langle T_1 \text{ BEGIN} \rangle \]
\[ \langle T_1, A, 1, 8 \rangle \]

Buffer Pool

A=1 B=5 C=7
WAL – EXAMPLE

Schedule

TIME

T₁

BEGIN
W(A)
W(B)
⋮
COMMIT

WAL Buffer

1

<T₁ BEGIN>
<T₁, A, 1, 8>

Buffer Pool

2

A=8  B=5  C=7

EXAMP L E

3

<T₁, A, 1, 8>

WAL Buffer

A=1  B=5  C=7
WAL – EXAMPLE

Schedule

T₁

BEGIN
W(A)
W(B)
⋮
COMMIT

WAL Buffer

<T₁ BEGIN>
<T₁, A, 1, 8>
<T₁, B, 5, 9>

Buffer Pool

A=8 B=9 C=7
WAL – EXAMPLE

Schedule

T₁

BEGIN
W(A)
W(B)
⋮
COMMIT

WAL Buffer

<T₁ BEGIN>
<T₁, A, 1, 8>
<T₁, B, 5, 9>
<T₁ COMMIT>

Buffer Pool

A=8 B=9 C=7

WAL

<T₁ BEGIN>
<T₁, A, 1, 8>
<T₁, B, 5, 9>
<T₁ COMMIT>

Example

T₁ result is now safe to return to application.
**WAL – EXAMPLE**

**Schedule**

```
T_1
BEGIN
W(A)
W(B)
...
COMMIT
```

**WAL Buffer**

```
<T_1 BEGIN>
<T_1, A, 1, 8>
<T_1, B, 5, 9>
<T_1 COMMIT>
```

**Buffer Pool**

```
A=8  B=9  C=7
```

*Txn result is now safe to return to application.*
WAL – EX

Schedule

\[ T_1 \]

BEGIN
W(A)
W(B)
\
COMMIT

WAL Buffer

\(<T_1, \text{BEGIN}>\)
\(<T_1, A, 1, 8>\)
\(<T_1, B, 5, 9>\)
\(<T_1, \text{COMMIT}>\)

Buffer Pool

A=1 \quad B=5 \quad C=7

Everything we need to restore \( T_1 \) is in the log!

Txn result is now safe to return to application.
WAL – IMPLEMENTATION

When should the DBMS write log entries to disk?
→ When the transaction commits.
→ Can use group commit to batch multiple log flushes together to amortize overhead.
WAL – GROUP COMMIT

Schedule

T₁
BEGIN
W(A)
W(B)

... 

COMMIT

T₂
BEGIN
W(C)
W(D)

... 

COMMIT

WAL Buffers

<T₁ BEGIN>
**WAL – GROUP COMMIT**

**Schedule**

<table>
<thead>
<tr>
<th>T₁</th>
<th>T₂</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>BEGIN</strong></td>
<td><strong>BEGIN</strong></td>
</tr>
<tr>
<td>W(A)</td>
<td>W(C)</td>
</tr>
<tr>
<td>W(B)</td>
<td>W(D)</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td><strong>COMMIT</strong></td>
<td><strong>COMMIT</strong></td>
</tr>
</tbody>
</table>

**WAL Buffers**

- `<T₁, BEGIN>`
- `<T₁, A, 1, 8>`

TIME
WAL – GROUP COMMIT

Schedule

T₁
BEGIN
W(A)
W(B)

⋮

COMMIT

T₂
BEGIN
W(C)
W(D)

⋮

COMMIT

WAL Buffers

<T₁ BEGIN>
<T₁, A, 1, 8>
<T₁, B, 5, 9>
WAL – GROUP COMMIT

Schedule

\[
\begin{array}{c|c}
T_1 & T_2 \\
\hline
\text{BEGIN} & \text{BEGIN} \\
\text{W(A)} & \text{W(C)} \\
\text{W(B)} & \text{W(D)} \\
\vdots & \vdots \\
\text{COMMIT} & \text{COMMIT} \\
\end{array}
\]

WAL Buffers

\[
\begin{array}{c}
<T_1 \text{ BEGIN}> \\
<T_1, A, 1, 8> \\
<T_1, B, 5, 9> \\
<T_2 \text{ BEGIN}> \\
\end{array}
\]
WAL – GROUP COMMIT

Flush the buffer when it is full.

Schedule

\[ \begin{array}{c|c}
T_1 & T_2 \\
\hline
\text{BEGIN} & \text{BEGIN} \\
\text{W(A)} & \text{W(A)} \\
\text{W(B)} & \text{W(B)} \\
\vdots & \vdots \\
\text{COMMIT} & \text{COMMIT} \\
\end{array} \]

\[ \begin{array}{c|c}
T_1 & T_2 \\
\hline
\text{BEGIN} & \text{BEGIN} \\
\text{W(C)} & \text{W(C)} \\
\text{W(D)} & \text{W(D)} \\
\vdots & \vdots \\
\text{COMMIT} & \text{COMMIT} \\
\end{array} \]
WAL – GROUP COMMIT

Schedule

T₁

BEGIN
W(A)

BEGIN
W(B)

⋮

COMMIT

T₂

BEGIN
W(C)

BEGIN
W(D)

⋮

COMMIT

WAL Buffers

<T₁ BEGIN>
<T₁, A, 1, 8>
<T₁, B, 5, 9>
<T₂ BEGIN>
<T₂, C, 1, 2>

<T₂, D, 3, 4>

<T₁, A, 1, 8>
<T₁, B, 5, 9>
<T₂ BEGIN>
<T₂, C, 1, 2>
WAL – GROUP COMMIT

Schedule

<table>
<thead>
<tr>
<th>TIME</th>
<th>T₁</th>
<th>T₂</th>
</tr>
</thead>
<tbody>
<tr>
<td>BEGIN</td>
<td>W(A)</td>
<td>W(B)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>COMMIT</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>TIME</th>
<th>T₁</th>
<th>T₂</th>
</tr>
</thead>
<tbody>
<tr>
<td>BEGIN</td>
<td>W(C)</td>
<td>W(D)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>COMMIT</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

WAL Buffers

<table>
<thead>
<tr>
<th>TIME</th>
<th>T₁</th>
<th>T₂</th>
</tr>
</thead>
<tbody>
<tr>
<td>BEGIN</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>COMMIT</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>TIME</th>
<th>T₁</th>
<th>T₂</th>
</tr>
</thead>
<tbody>
<tr>
<td>BEGIN</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>COMMIT</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- <T₁ BEGIN>
- <T₁, A, 1, 8>
- <T₁, B, 5, 9>
- <T₂ BEGIN>
- <T₂, C, 1, 2>
- <T₂, D, 3, 4>
**WAL – GROUP COMMIT**

### Schedule

<table>
<thead>
<tr>
<th>Time</th>
<th>Event</th>
<th>Buffer</th>
</tr>
</thead>
<tbody>
<tr>
<td>$T_1$</td>
<td>BEGIN</td>
<td>W(A)</td>
</tr>
<tr>
<td>$T_1$</td>
<td></td>
<td>W(B)</td>
</tr>
<tr>
<td>$T_2$</td>
<td>BEGIN</td>
<td>W(C)</td>
</tr>
<tr>
<td>$T_2$</td>
<td></td>
<td>W(D)</td>
</tr>
<tr>
<td></td>
<td>COMMIT</td>
<td></td>
</tr>
</tbody>
</table>

### WAL Buffers

<table>
<thead>
<tr>
<th>Time</th>
<th>Event</th>
<th>Buffer</th>
</tr>
</thead>
<tbody>
<tr>
<td>$T_1$</td>
<td>BEGIN</td>
<td></td>
</tr>
<tr>
<td>$T_1$</td>
<td></td>
<td>&lt;T, A, 1, 8&gt;</td>
</tr>
<tr>
<td>$T_1$</td>
<td></td>
<td>&lt;T, B, 5, 9&gt;</td>
</tr>
<tr>
<td>$T_2$</td>
<td>BEGIN</td>
<td></td>
</tr>
<tr>
<td>$T_2$</td>
<td></td>
<td>&lt;T, D, 3, 4&gt;</td>
</tr>
<tr>
<td>$T_2$</td>
<td></td>
<td>&lt;T, C, 1, 2&gt;</td>
</tr>
</tbody>
</table>

Flush after an elapsed amount of time.
WAL – GROUP COMMIT

Schedule

T₁
BEGIN
W(A)
W(B)
⋮
COMMIT

T₂
BEGIN
W(C)
W(D)
⋮
COMMIT

WAL Buffers

<T₁, BEGIN>
<T₁, A, 1, 8>
<T₁, B, 5, 9>
<T₂, BEGIN>
<T₂, C, 1, 2>

Flush after an elapsed amount of time.
WAL – GROUP COMMIT

Schedule

T₁

BEGIN
W(A)
W(B)

⋮

COMMIT

T₂

BEGIN
W(C)
W(D)

⋮

COMMIT

WAL Buffers

Flush after an elapsed amount of time.

<T₁, BEGIN>
<T₁, A, 1, 8>
<T₁, B, 5, 9>
<T₂, BEGIN>
<T₂, C, 1, 2>
<T₂, D, 3, 4>
**WAL – IMPLEMENTATION**

*When should the DBMS write log entries to disk?*

→ When the transaction commits.
→ Can use group commit to batch multiple log flushes together to amortize overhead.

*When should the DBMS write dirty records to disk?*

→ Every time thetxn executes an update?
→ Once when the txn commits?
Almost every DBMS uses **NO-FORCE + STEAL**

### Runtime Performance

<table>
<thead>
<tr>
<th></th>
<th>NO-STEAL</th>
<th>STEAL</th>
</tr>
</thead>
<tbody>
<tr>
<td>NO-FORCE</td>
<td>Fastest</td>
<td>–</td>
</tr>
<tr>
<td>FORCE</td>
<td>Slowest</td>
<td>–</td>
</tr>
</tbody>
</table>

### Recovery Performance

<table>
<thead>
<tr>
<th></th>
<th>NO-STEAL</th>
<th>STEAL</th>
</tr>
</thead>
<tbody>
<tr>
<td>NO-FORCE</td>
<td>–</td>
<td>Slowest</td>
</tr>
<tr>
<td>FORCE</td>
<td>Fastest</td>
<td>–</td>
</tr>
</tbody>
</table>
Almost every DBMS uses **NO-FORCE + STEAL**

### Runtime Performance

<table>
<thead>
<tr>
<th></th>
<th>NO-STEAL</th>
<th>STEAL</th>
</tr>
</thead>
<tbody>
<tr>
<td>NO-FORCE</td>
<td>–</td>
<td><strong>Fastest</strong></td>
</tr>
<tr>
<td>FORCE</td>
<td><strong>Slowest</strong></td>
<td>–</td>
</tr>
</tbody>
</table>

### Recovery Performance

<table>
<thead>
<tr>
<th></th>
<th>NO-STEAL</th>
<th>STEAL</th>
</tr>
</thead>
<tbody>
<tr>
<td>NO-FORCE</td>
<td>–</td>
<td><strong>Slowest</strong></td>
</tr>
<tr>
<td>FORCE</td>
<td><strong>Fastest</strong></td>
<td>–</td>
</tr>
</tbody>
</table>

- **Undo + Redo**
- **No Undo + No Redo**
LOGGING SCHEMES

Physical Logging
→ Record the changes made to a specific location in the database.
→ Example: `git diff`

Logical Logging
→ Record the high-level operations executed by txns.
→ Not necessarily restricted to single page.
→ Example: The `UPDATE`, `DELETE`, and `INSERT` queries invoked by a txn.
PHYSICAL VS. LOGICAL LOGGING

Logical logging requires less data written in each log record than physical logging.

Difficult to implement recovery with logical logging if you have concurrent txns.
→ Hard to determine which parts of the database may have been modified by a query before crash.
→ Also takes longer to recover because you must re-execute every txn all over again.
PHYSIOLOGICAL LOGGING

Hybrid approach where log records target a single page but do not specify organization of the page.
→ Identify tuples based on their slot number.
→ Allows DBMS to reorganize pages after a log record has been written to disk.

This is the most popular approach.
**LOGGING SCHEMES**

**UPDATE** foo **SET** val = XYZ **WHERE** id = 1;

**Physical**

\(<T_1,\>
\text{Table}=X,\)
\text{Page}=99,\)
\text{Offset}=4,\)
\text{Before}=ABC,\)
\text{After}=XYZ>\)

\(<T_1,\>
\text{Index}=X\text{-PKEY},\)
\text{Page}=99,\)
\text{Slot}=1,\)
\text{Before}=ABC,\)
\text{After}=XYZ>\)

**Logical**

\(<T_1,\>
\text{Query}="\text{UPDATE} \text{ foo} \text{ SET} \text{ val}=XYZ \text{ WHERE} \text{ id}=1">\)

**Physiological**

\(<T_1,\>
\text{Table}=X,\)
\text{Page}=45,\)
\text{Offset}=9,\)
\text{Key}=(1,Record1)\)
CHECKPOINTS

The WAL will grow forever.

After a crash, the DBMS must replay the entire log, which will take a long time.

The DBMS periodically takes a checkpoint where it flushes all buffers out to disk.
CHECKPOINTS

Output onto stable storage all log records currently residing in main memory.

Output to the disk all modified blocks.

Write a `<CHECKPOINT>` entry to the log and flush to stable storage.
Any txn that committed before the checkpoint is ignored ($T_1$).
Any txn that committed before the checkpoint is ignored ($T_1$).

$T_2 + T_3$ did not commit before the last checkpoint.

$\rightarrow$ Need to **redo** $T_2$ because it committed after checkpoint.

$\rightarrow$ Need to **undo** $T_3$ because it did not commit before the crash.
CHECKPOINTS – CHALLENGES

The DBNS must stall txns when it takes a checkpoint to ensure a consistent snapshot.

Scanning the log to find uncommitted txns can take a long time.

Not obvious how often the DBMS should take a checkpoint...
CHECKPOINTS – FREQUENCY

Checkpointing too often causes the runtime performance to degrade.
→ System spends too much time flushing buffers.

But waiting a long time is just as bad:
→ The checkpoint will be large and slow.
→ Makes recovery time much longer.
CONCLUSION

Write-Ahead Logging is (almost) always the best approach to handle loss of volatile storage.

Use incremental updates (STEAL + NO-FORCE) with checkpoints.

On Recovery: undo uncommitted txns + redo committed txns.
Better Checkpoint Protocols.
Recovery with ARIES.