Database Logging
ADMINISTRIVIA

Project #3 is due Sun Nov 14\textsuperscript{nd} @ 11:59pm.

Homework #4 is due Wed Nov 10\textsuperscript{th} @ 11:59pm.
UPCOMING DATABASE TALK

Vertica – High Performance Over Varying Terrain
→ Mon Nov 8th @ 4:30pm ET
MOTIVATION

Schedule

\[ T_1 \]

BEGIN
R(A)
W(A)
:
COMMIT

Buffer Pool

\[ A=1 \]
MOTIVATION

Schedule

\[
\begin{align*}
T_1 & \quad \text{BEGIN} \\
& \quad R(A) \\
& \quad W(A) \\
& \quad \vdots \\
& \quad \text{COMMIT}
\end{align*}
\]

Buffer Pool

Page

\( A = 1 \)
**MOTIVATION**

Schedule

BEGIN
R(A)
W(A)
⋮
COMMIT

Buffer Pool

A=1

Page

TIME

T₁

15-445/645 (Fall 2021)
MOTIVATION

Schedule

BEGIN
R(A)
W(A)
⋮
COMMIT

Buffer Pool

T₁

Page

A=1

A=1

CMU-DB
15-445/645 (Fall 2021)
MOTIVATION

Schedule

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

Buffer Pool

A=2

Page

TIME

T_1

A=1
MOTIVATION

Schedule

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

Buffer Pool

A=2

Page
A=1

TIME
Schedule

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

Buffer Pool

A=1

Page

TIME
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.
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’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.
**STORAGE TYPES**

**Volatile Storage:**
→ Data does not persist after power loss or program exit.
→ Examples: DRAM, SRAM

**Non-volatile Storage:**
→ Data persists after power loss and program exit.
→ Examples: HDD, SDD

**Stable Storage:**
→ A non-existent form of non-volatile storage that survives all possible failures scenarios.
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-Replaceable 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.
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

T_1
BEGIN
R(A)
W(A)
\vdots
R(B)
W(B)
COMMIT
ABORT

T_2
BEGIN
R(B)
W(B)
COMMIT

A=1
B=9
C=7
Buffer Pool

Schedule

\[ T_1 \quad T_2 \]

- BEGIN
- R(A)
- W(A)
- ...
- ABORT

- BEGIN
- R(B)
- W(B)
- COMMIT

Buffer Pool

A=1, B=9, C=7
BEGIN R(A) W(A)
⋮
ABORT

BEGIN R(B) W(B) COMMIT

Buffer Pool
A=1 B=9 C=7
BEGIN R(A)
W(A)
⋮
ABORT

BEGIN R(B)
W(B)
COMMIT

A=3 B=9 C=7

Buffer Pool

A=1 B=9 C=7

Schedule

T₁

T₂

TIME
Buffer Pool

```
BEGIN
R(A)
W(A)

⋮

ABORT
```

```
BEGIN
R(B)
W(B)
COMMIT
```

```
A=1  B=9  C=7
```

```
A=3  B=9  C=7
```

Schedule

TIME
Buffer Pool

Schedule

T₁
BEGIN
R(A)
W(A)
⋮
ABORT

T₂
BEGIN
R(B)
W(B)
COMMIT

A=1  B=9  C=7

A=3  B=8  C=7

A=1  B=9  C=7
Do we force $T_2$’s changes to be written to disk?
Buffer Pool

Schedule

\[ \text{BEGIN} \]
\[
\text{R}(A) \\
\text{W}(A) \\
\vdots \\
\text{ABORT}
\]

\[ \text{COMMIT} \]

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?

\[ A=1 \quad B=9 \quad C=7 \]

\[ A=3 \quad B=8 \quad C=7 \]
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?
Buffer Pool

Schedule

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(A)
W(A)

$\vdots$

BEGIN
R(B)
W(B)
COMMIT

ABORT

A=1  B=9  C=7

A=3  B=8  C=7

A=3  B=8  C=7
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

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

Buffer Pool

A=1  B=9  C=7
NO-STEAL + FORCE

Schedule

\( T_1 \)
BEGIN
R(A)
W(A)
⋮
ABORT

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

Buffer Pool

\[ A=1 \quad B=9 \quad C=7 \]
NO-STEAL + FORCE

Schedule

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

TIME

ABORT

CMU-DB
15-445/645 (Fall 2021)
NO-STEAL + FORCE

Schedule

T₁
BEGIN
R(A)
W(A)

⋮
ABORT

T₂
BEGIN
R(B)
W(B)
COMMIT

Buffer Pool

A=3  B=9  C=7

A=1  B=9  C=7
NO-STEAL + FORCE

Schedule

T₁
BEGIN
R(A)
W(A)

⋮
ABORT

T₂
BEGIN
R(B)
W(B)
COMMIT

Buffer Pool

A=3  B=9  C=7

A=1  B=9  C=7

BEGIN
R(A)
W(A)

⋮
ABORT
NO-STEAL + FORCE

Schedule

T₁
BEGIN
R(A)
W(A)
⋮
ABORT

T₂
BEGIN
R(B)
W(B)
COMMIT

Buffer Pool
A=3  B=8  C=7

A=1  B=9  C=7
**NO-STEAL + FORCE**

**Schedule**

$T_1$

BEGIN
R(A)
W(A)

⋮

ABORT

$T_2$

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**

**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**

- **T1**
  - **W(A)**
  - **BEGIN**
  - **R(B)**
  - **W(B)**
  - **COMMIT**
  - **ABORT**
  - **A=1 B=9 C=7**

- **T2**
  - **A=3 B=8 C=7**

---

**NO-STEAL** means that **T1** changes cannot be written to disk yet.

**FORCE** means that **T2** changes must be written to disk at this point.
**NO-STEAL + FORCE**

**Schedule**

\[\begin{align*}
\text{NO-STEAL means that } T_1 \text{ changes cannot be written to disk yet.} \\
\text{FORCE means that } T_2 \text{ changes must be written to disk at this point.}
\end{align*}\]

\[\begin{align*}
\text{BEGIN} & \quad \text{R(A)} \\
\text{W(A)} & \quad \text{⋮} \\
\text{ABORT} & \\
\text{BEGIN} & \quad \text{R(B)} \\
\text{W(B)} & \quad \text{COMMIT} \\
\end{align*}\]
**NO-STEAL + FORCE**

**Schedule**

- **$T_1$**
  - BEGIN
  - R(A)
  - W(A)
  - ...
  - ABORT

- **$T_2$**
  - BEGIN
  - R(B)
  - W(B)
  - COMMIT

**Buffer Pool**

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

**Now it’s trivial to rollback $T_1$**
NO-STEAL + FORCE

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

1 2 3 4

Master Page Table

Disk

Database Root

DB 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

DB Root

Master Page Table

Disk

Database Root

...
SHADOW PAGING – EXAMPLE

Memory

1
2
3
4

DB Root

Master Page Table

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.

Disks

Active modifying txn updates shadow pages.

DB Root

Master Page Table

Shadow Page Table

Txn $T_1$

Database Root
Read-only txns access the current master.

Active modifying txn updates shadow pages.

Txn $T_1$
SHADOW PAGING – EXAMPLE

Read-only txns access the current master.

Active modifying txn updates shadow pages.

Master Page Table

Shadow Page Table

Disk

Database Root

Txn $T_1$

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.

Txn $T_1$
SHADOW PAGING – EXAMPLE

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

**Active modifying txn updates shadow pages.**

Txn $T_1$ accesses the current master, while active modifying transactions update shadow pages.
SHADOW PAGING – EXAMPLE

*Read-only txns access the current master.*

**Master Page Table**

1
2
3
4

DB Root

**Shadow Page Table**

1
2
3
4

*Active modifyingtxn updates shadow pages.*

**Disk**

*Database Root*
**SHADOW PAGING – EXAMPLE**

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

**Active modifying txn updates shadow pages.**

**Master Page Table**

**Shadow Page Table**

**Database Root**

 updates

**Disk**

**Txn T₁**

COMMIT

- 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**

**Master Page Table**

**DB Root**

**Shadow Page Table**

**Disk**

- **Update**
- **Database Root**

- **Commit**
**SHADOW PAGING – EXAMPLE**

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

**Active modifying txn updates shadow pages.**

**Disk**

- **Database Root**
  - [ ]
  - [ ]
  - [ ]
  - [ ]

**Master Page Table**

- [ ]
- [ ]
- [ ]
- [ ]

**Shadow Page Table**

- [ ]
- [ ]
- [ ]
- [ ]

**Txn T₁**

- COMMIT
SHADOW PAGING – EXAMPLE

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

**Disk**
- **Database Root**

**Txn T₁**

**COMMIT**
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.
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.
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.
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.
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**
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
WAL PROTOCOL

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.
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

| A=1 | B=5 | C=7 |

Buffer Pool

| A=1 | B=5 | C=7 |
WAL – EXAMPLE

Schedule

\[ T_1 \]

BEGIN
W(A)
W(B)

⋮

COMMIT

WAL Buffer

\[
\langle T_1 \text{ BEGIN} \rangle
\]

Buffer Pool

\[
A=1 \quad B=5 \quad C=7
\]
**WAL – EXAMPLE**

**Schedule**

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

**WAL Buffer**

```
<T_1 BEGIN>
```

**Buffer Pool**

```
A=1  B=5  C=7
```
WAL – EXAMPLE

Schedule

$$T_1$$

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

WAL Buffer

$$<T_1 \text{ BEGIN}>$$
$$<T_1, A, 1, 8>$$

Buffer Pool

A=1 B=5 C=7
WAL – EXAMPLE

Schedule

T₁

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

WAL Buffer

<T₁, BEGIN>
<T₁, A, 1, 8>

Buffer Pool

A=8  B=5  C=7
**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>`

**Buffer Pool**

- A=8
- B=9
- C=7
WAL – EXAMPLE

**Schedule**

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

**WAL Buffer**

\[
\begin{align*}
& <T_1, \text{BEGIN}> \\
& <T_1, A, 1, 8> \\
& <T_1, B, 5, 9> \\
& <T_1, \text{COMMIT}>
\end{align*}
\]

**Buffer Pool**

\[
\begin{align*}
A=8 & \\
B=9 & \\
C=7 &
\end{align*}
\]

\[
\begin{align*}
& \langle T_1, \text{BEGIN} \rangle \\
& \langle T_1, A, 1, 8 \rangle \\
& \langle T_1, B, 5, 9 \rangle \\
& \langle T_1, \text{COMMIT} \rangle
\end{align*}
\]

\[
\begin{align*}
A=1 & \\
B=5 & \\
C=7 &
\end{align*}
\]
WAL – EXAMPLE

Schedule

\[ T_1 \]

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

WAL Buffer

\[ \langle T_1, \text{BEGIN} \rangle \]
\[ \langle T_1, \text{A, 1, 8} \rangle \]
\[ \langle T_1, \text{B, 5, 9} \rangle \]
\[ \langle T_1, \text{COMMIT} \rangle \]

Buffer Pool

\[
\begin{array}{ccc}
A &=& 8 \\
B &=& 9 \\
C &=& 7 \\
\end{array}
\]

WAL – WAL Buffer

\[
\begin{array}{l}
\langle T_1, \text{BEGIN} \rangle \\
\langle T_1, \text{A, 1, 8} \rangle \\
\langle T_1, \text{B, 5, 9} \rangle \\
\langle T_1, \text{COMMIT} \rangle \\
\end{array}
\]

Txn result is now safe to return to application.

<15-445/645 (Fall 2021)>
WAL – EXAMPLE

Schedule

<table>
<thead>
<tr>
<th>Time</th>
<th>Txn 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>BEGIN</td>
<td>W(A)</td>
</tr>
</tbody>
</table>
| W(B) | ...
| COMMIT |

WAL Buffer

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

Txn result is now safe to return to application.

Buffer Pool

<table>
<thead>
<tr>
<th>Buffer</th>
<th>A=8</th>
<th>B=9</th>
<th>C=7</th>
</tr>
</thead>
</table>

A=1 B=5 C=7
**WAL – EXAMPLE**

**Schedule**

- **T₁**
  - BEGIN
  - W(A)
  - W(B)
  - ...
  - COMMIT

**WAL Buffer**

- x

**Buffer Pool**

- x

** Txn result is now safe to return to application. **
Buffer Pool

Schedule

T₁

BEGIN
W(A)
W(B)
…
COMMIT

WAL Buffer

Everything we need to restore T₁ is in the log!

A=1 B=5 C=7

Txn result is now safe to return to application.

Buffer Pool

< T₁ BEGIN >
< T₁, A, 1, 8 >
< T₁, B, 5, 9 >
< T₁, COMMIT >
When should the DBMS write log entries to disk?
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

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

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

WAL
WAL – GROUP COMMIT

Schedule

\[
\begin{align*}
T_1 & \quad T_2 \\
\text{BEGIN} & \quad \text{BEGIN} \\
W(A) & \quad W(B) \\
\vdots & \quad \vdots \\
\text{COMMIT} & \quad \text{COMMIT} \\
W(C) & \quad W(D) \\
\vdots & \quad \vdots
\end{align*}
\]
WAL – GROUP COMMIT

Schedule

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

<table>
<thead>
<tr>
<th>WAL</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;T₁ BEGIN&gt;</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>WAL</th>
</tr>
</thead>
<tbody>
<tr>
<td>COMMIT</td>
</tr>
<tr>
<td>COMMIT</td>
</tr>
</tbody>
</table>

TIME
WAL – GROUP COMMIT

Schedule

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

WAL

\[
\langle T_1, \text{BEGIN} \rangle \\
\langle T_1, A, 1, 8 \rangle
\]
WAL – GROUP COMMIT

**Schedule**

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

**TIME**

<table>
<thead>
<tr>
<th></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>

**WAL**

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

Schedule

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

WAL

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

Schedule

T₁
BEGIN
W(A)
W(B)

⋮

COMMIT

T₂
BEGIN
W(C)
W(D)

⋮

COMMIT

WAL

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

Schedule

T₁ |
| T₂ |

BEGIN
W(A)
W(B)

BEGIN
W(C)
W(D)

BEGIN
W(C)
W(D)

COMMIT

COMMIT

⟨T₁, BEGIN⟩
⟨T₁, A, 1, 8⟩
⟨T₁, B, 5, 9⟩
⟨T₂, BEGIN⟩
⟨T₂, C, 1, 2⟩

Flush the buffer when it is full.
WAL – GROUP COMMIT

Schedule

\[
\begin{array}{c|c}
T_1 & T_2 \\
\hline
\text{BEGIN} & \text{BEGIN} \\
W(A) & <T_1, \text{BEGIN}> \\
W(B) & <T_1, A, 1, 8> \\
& <T_1, B, 5, 9> \\
\vdots & <T_2, \text{BEGIN}> \\
\text{BEGIN} & <T_2, C, 1, 2> \\
W(C) & \text{COMMIT} \\
W(D) & \text{COMMIT} \\
\end{array}
\]

Flush the buffer when it is full.

Flush the buffer when it is full.
## WAL – GROUP COMMIT

### Schedule

<table>
<thead>
<tr>
<th>Time</th>
<th>Task 1</th>
<th>Task 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>T1</td>
<td>BEGIN</td>
<td>BEGIN</td>
</tr>
<tr>
<td></td>
<td>W(A)</td>
<td>W(C)</td>
</tr>
<tr>
<td></td>
<td>W(B)</td>
<td>W(D)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>T2</td>
<td>COMMIT</td>
<td>COMMIT</td>
</tr>
</tbody>
</table>

### WAL

<table>
<thead>
<tr>
<th>Time</th>
<th>Task 1</th>
<th>Task 2</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>BEGIN</td>
<td>BEGIN</td>
</tr>
<tr>
<td></td>
<td>&lt;T1, A, 1, 8&gt;</td>
<td>&lt;T1, B, 5, 9&gt;</td>
</tr>
<tr>
<td></td>
<td>&lt;T2, C, 1, 2&gt;</td>
<td>&lt;T2, D, 3, 4&gt;</td>
</tr>
</tbody>
</table>

{-cmu-db-}

15-445/645 (Fall 2021)
WAL – GROUP COMMIT

Schedule

TIME

T₁ T₂

BEGIN
W(A)
W(B)

BEGIN
W(C)
W(D)

⋮

COMMIT

COMMIT

WAL

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

<T₂, D, 3, 4>

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

Schedule

\[ \begin{align*}
T_1 & \\
\text{BEGIN} & \text{W(A)} \\
& \text{W(B)} \\
& \vdots \\
\text{COMMIT} &
\end{align*} \]

\[ \begin{align*}
T_2 & \\
\text{BEGIN} & \text{W(C)} \\
& \text{W(D)} \\
& \vdots \\
\text{COMMIT} &
\end{align*} \]

WAL

\[ \begin{align*}
\text{BEGIN} & \text{W(A)} \\
\text{BEGIN} & \text{W(B)} \\
\text{BEGIN} & \text{W(C)} \\
\text{BEGIN} & \text{W(D)} \\
\text{COMMIT} &
\end{align*} \]

\[ \begin{align*}
\text{BEGIN} & \\
\text{BEGIN} & \\
\text{BEGIN} & \\
\text{BEGIN} & \\
\text{COMMIT} &
\end{align*} \]

\[ \begin{align*}
\text{<T}_1\text{, A, 1, 8>} \\
\text{<T}_1\text{, B, 5, 9>} \\
\text{<T}_2\text{, C, 1, 2>} \\
\text{<T}_2\text{, D, 3, 4>}
\end{align*} \]
WAL – GROUP COMMIT

Schedule

T1
BEGIN
W(A)
W(B)

T2
BEGIN
W(C)
W(D)

COMMIT
COMMIT

WAL

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

<T2 BEGIN>
<T2, D, 3, 4>

COMMIT

Flush after an elapsed amount of time.

TIME

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

<T2, A, 1, 8>
<T2, B, 5, 9>

<T1, BEGIN>

<15-445/645 (Fall 2021)>
WAL – GROUP COMMIT

Schedule

<table>
<thead>
<tr>
<th>T₁</th>
<th>T₂</th>
</tr>
</thead>
</table>
| BEGIN  
  W(A)  
  W(B) | BEGIN  
  W(C)  
  W(D) |
| ⋮ | ⋮ |
| COMMIT | COMMIT |

Flush after an elapsed amount of time.

WAL

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

Schedule

T1
BEGIN
W(A)
W(B)

... 
COMMIT

T2
BEGIN
W(C)
W(D)

... 
COMMIT

WAL

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>

<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?*
**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 the txn 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>–</td>
<td>Slowest</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>Fastest</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>

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

UPDATE foo SET val = XYZ WHERE id = 1;

Physical

<T_1,
 Table=X,
 Page=99,
 Offset=4,
 Before=ABC,
 After=XYZ>
<T_1,
 Index=X_PKEY,
 Page=45,
 Offset=9,
 Key=(1,Record1)>
**LOGGING SCHEMES**

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

**Physical**

<T₁,
    Table=X,
    Page=99,
    Offset=4,
    Before=ABC,
    After=XYZ>

<T₁,
    Index=X_PKEY,
    Page=45,
    Offset=9,
    Key=(1,Record1)>

**Logical**

<T₁,
    Query="UPDATE foo
            SET val=XYZ
            WHERE id=1">
LOGGING SCHEMES

UPDATE foo SET val = XYZ WHERE id = 1;

**Physical**
<T₁,
Table=X,
Page=99,
Offset=4,
Before=ABC,
After=XYZ>

**Logical**
<T₁,
Query="UPDATE foo
SET val=XYZ
WHERE id=1">

**Physiological**
<T₁,
Table=X,
Page=99,
Slot=1,
Before=ABC,
After=XYZ>

<T₁,
Index=X_PKEY,
IndexPage=45,
Key=(1,Record1)>

<T₁,
Index=X_PKEY,
Page=99,
Offset=1,
Before=ABC,
After=XYZ>

<T₁,
Index=X_PKEY,
IndexPage=45,
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.
CHECKPOINTS

WAL

<T1 BEGIN>
<T1, A, 1, 2>
<T1 COMMIT>
<T2 BEGIN>
<T2, A, 2, 3>
<T3 BEGIN>
<T3, A, 3, 4>
<CHECKPOINT>
<T2 COMMIT>
<T3 COMMIT>
<T3, A, 3, 4>

CRASH!
CHECKPOINTS

WAL

<T1 BEGIN>
<T1, A, 1, 2>
<T1 COMMIT>
<T2 BEGIN>
<T2, A, 2, 3>
<T3 BEGIN>
<CHECKPOINT>
<T2 COMMIT>
<T3, A, 3, 4>

CRASH!
CHECKPOINTS

Any txn that committed before the checkpoint is ignored ($T_1$).
CHECKPOINTS

Any txn that committed before the checkpoint is ignored ($T_1$).

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

Any txn that committed before the checkpoint is ignored ($T_1$).

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

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

→ Need to **undo** $T_3$ because it did not commit before the crash.

CRASH!
CHECKPOINTS – CHALLENGES

The DBMS 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...
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.