Logging Schemes
Homework #4: Today @ 11:59pm

Project #3: Monday Nov 19th @ 11:59am

Homework #5: Monday Dec 3rd @ 11:59pm
BEGIN R(A) W(A) ... COMMIT
MOTIVATION

BEGIN R(A)
W(A)
⋯
COMMIT

Buffer Pool

A=2

A=1

Page

TIME

Schedule

T₁

BEGIN R(A)
W(A)
⋯
COMMIT
MOTIVATION

Schedule

\[ T_1 \]

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

Buffer Pool

\[
\begin{array}{c}
A=2 \\
\end{array}
\]

Page

\[
\begin{array}{c}
A=1 \\
\end{array}
\]
BEGIN
R(A)
W(A)
⋮
COMMIT

Schedule

Buffer Pool

A=1
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
Checkpoints
Logging Schemes
CRASH RECOVERY

DBMS is divided into different components based on the underlying storage device.

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 is cut.
→ Examples: DRAM, SRAM

Non-volatile Storage:
→ Data persists after losing power.
→ 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 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.
STORAGE MEDIA FAILURE

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 R(A) W(A)

⋮

BEGIN R(B) W(B) COMMIT

ABORT

Buffer Pool

A=1 B=9 C=7

A=1 B=9 C=7
BUFFER POOL

Schedule

T₁
BEGIN
R(A)
W(A)

⋮
ABORT

T₂
BEGIN
R(B)
W(B)
COMMIT

Buffer Pool

A=1  B=9  C=7

A=3  B=9  C=7

A=3  B=9  C=7

TIME
BUFFER POOL

Schedule

\[ T_1 \rightarrow T_2 \]

BEGIN
R(A)
W(A)

BEGIN
R(B)
W(B)
COMMIT

ABORT

Buffer Pool

\[
\begin{array}{c|c|c}
A & B & C \\
3 & 9 & 7 \\
\end{array}
\]

\[
\begin{array}{c|c|c}
A & B & C \\
1 & 9 & 7 \\
\end{array}
\]
BUFFER POOL

Schedule

T₁  T₂
BEGIN R(A)  BEGIN R(B)
A=1    B=8
W(A)   W(B)
ABORT

Buffer Pool

A=3  B=8  C=7

A=1  B=9  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

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 is allowed to commit.

FORCE: Is enforced.
NO-FORCE: Is not enforced.
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
**NO-STEAL + FORCE**

Schedule

T₁  |  T₂
---|---
BEGIN R(A) W(A) | BEGIN R(B) W(B) COMMIT

Buffer Pool

A=1 B=9 C=7

Time

ABORT

A=3 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=1
B=9
C=7

A=3
B=9
C=7

A=3
B=9
C=7

A=1
B=9
C=7

BEGIN
R(A)
W(A)

⋮
ABORT

BEGIN
R(B)
W(B)
COMMIT

A=3
B=9
C=7

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

Schedule

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

\[ \vdots \]
ABORT

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

Buffer Pool

A=3  B=8  C=7

A=1  B=9  C=7

TIME
**NO-STEAL + FORCE**

**Schedule**

<table>
<thead>
<tr>
<th>Time</th>
<th>Task</th>
<th>Buffer Pool</th>
</tr>
</thead>
<tbody>
<tr>
<td>T1</td>
<td>BEGIN R(A) W(A)</td>
<td>A=1 B=9 C=7</td>
</tr>
<tr>
<td></td>
<td>BEGIN R(B) W(B)</td>
<td>A=3 B=8 C=7</td>
</tr>
<tr>
<td></td>
<td>COMMIT</td>
<td>A=3 B=8 C=7</td>
</tr>
<tr>
<td>T2</td>
<td>FORCE</td>
<td>A=3 B=8 C=7</td>
</tr>
</tbody>
</table>

**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**

---

**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.
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 abortedtxn because the changes were not written to disk.
→ Never have to redo changes of a committedtxn because all the changes are guaranteed to be written to disk at commit time.
SHADOW PAGING

Maintain two separate copies of the database \((\text{master, shadow})\)
Updates are only made in the shadow copy.
When a txn commits, atomically switch the shadow to become the new master.

Buffer Pool Policy: \textit{NO-STEAL} + \textit{FORCE}
SHADOW PAGING

Organize the database pages in a tree structure where the root is a single disk page.

There are two copies of the tree, the master and shadow
→ The root points to the master copy.
→ Updates are applied to the shadow copy.
SHADOW PAGING – EXAMPLE

Memory

Disk

1
2
3
4

Master Page Table

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 transaction’s updates are part of the disk-resident database
→ After overwriting the root, all of the transaction’s updates are part of the disk-resident database.

Source: The Great Phil Bernstein
SHADOW PAGING – EXAMPLE

Memory

1
2
3
4

Disk

Master Page Table

DB Root

Txn $T_1$
SHADOW PAGING – EXAMPLE

Memory

DB Root

1
2
3
4

Master Page Table

1
2
3
4

Shadow Page Table

Disk

1
2
3
4

Txn T₁
SHADOW PAGING – EXAMPLE

Read-only txns access the current master.

Active modifying txn updates shadow pages.

Master Page Table

DB Root

Txn $T_1$

Shadow Page Table

Disk
SHADOW PAGING – EXAMPLE

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.

Disk

Master Page Table

1 2 3 4

Shadow Page Table

1 2 3 4

Txn T₁

COMMIT

DB Root
Read-only txns access the current master.

Active modifying txn updates shadow pages.

Txn $T_1$

COMMIT
SHADOW PAGING – EXAMPLE

Read-only txns access the current master.

Active modifying txn updates shadow pages.

Disk

Master Page Table

Db Root

Shadow Page Table

Txn $T_1$

COMMIT
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.
CORRECTION – SQLITE

The older version of SQLite did not make a complete copy of the database file at start of each new txn.

It writes old pages to a separate journal file before overwriting master version.

SQLite's default is now to use WAL.
OBSERVATION

Flushing non-contiguous pages on disk is slow (aka random writes).

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

Record the changes made to the database in a log file before the change is made.
→ Assume that the log is on stable storage.
→ Log contains sufficient information to perform the necessary undo and redo actions to restore the database after a crash.

Buffer Pool Policy: STEAL + NO-FORCE
The DBMS stages all of 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

BEGIN
W(A)
W(B)
⋮
COMMIT

WAL

$\langle T_1 \text{ BEGIN} \rangle$

Buffer Pool

A=1 B=5 C=7

A=1 B=9 C=7

TIME

T_1

EXAMPLE

< T_1

BEGIN >
WAL – EXAMPLE

Schedule

T₁
BEGIN
W(A)
W(B)
...
COMMIT

WAL

<T₁ BEGIN>
<T₁, A, 1, 8>

Buffer Pool

A=8 B=5 C=7

Example

<T₁, A, 1, 8>
WAL – EXAMPLE

Schedule

\[ T_1 \]

BEGIN
W(A)
W(B)

\vdots

COMMIT

WAL

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

Buffer Pool

A=8 B=9 C=7
WAL – EXAMPLE

Schedule

\[ T_1 \]

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

WAL

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

Buffer Pool

A=8 B=9 C=7
WAL – EXAMPLE

Schedule

BEGIN
W(A)
W(B)

COMMIT

WAL

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

Buffer Pool

A=8  B=9  C=7

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

Schedule

\[ T_1 \]
BEGIN
W(A)
W(B)
...
COMMIT

WAL

\[ \times \]

Txn result is now safe to return to application.

Buffer Pool

\[ \times \]

A=1
B=9
C=7
WAL – Example

Schedule

\[ T_1 \]
BEGIN
W(A)
W(B)
\vdots
COMMIT

WAL

Transaction result is now safe to return to application.

Buffer Pool

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

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

When should the DBMS write log entries to disk?

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

→ Every time the txn executes an update?
→ Once when the txn commits?
If we prevent the DBMS from writing dirty records to disk until the txn commits, then we don’t need to store their original values.
**WAL – DEFERRED UPDATES**

If we prevent the DBMS from writing dirty records to disk until the txn commits, then we don’t need to store their original values.

- **Replay the log and redo each update.**
  - `<T_1 \text{BEGIN}>`
  - `<T_1, A, 8>`
  - `<T_1, B, 9>`
  - `<T_1 \text{COMMIT}>`
  - **CRASH!**

- **Simply ignore all of T_1's updates.**
  - `<T_1 \text{BEGIN}>`
  - `<T_1, A, 8>`
  - `<T_1, B, 9>`
  - `<T_1 \text{COMMIT}>`
  - **CRASH!**
WAL – DEFERRED UPDATES

This won’t work if the change set of a txn is larger than the amount of memory available.

The DBMS cannot undo changes for an aborted txn if it doesn’t have the original values in the log.

We need to use the **STEAL** policy.
Almost every DBMS uses **NO-FORCE + STEAL**
LOGGING SCHEMES

Physical Logging
→ Record the changes made to a specific location in the database.
→ Example: "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 data organization of the page.

This is the most popular approach.
LOGGING SCHEMES

UPDATE foo SET val = XYZ WHERE id = 1;

Physical
<T_1,
 Table=X,
 Page=99,
 Offset=4,
 Before=ABC,
 After=XYZ>

Logical
<T_1,
 Query="UPDATE foo
 SET val=XYZ
 WHERE id=1">

Physiological
<T_1,
 Table=X,
 Page=99,
 ObjectId=1,
 Before=ABC,
 After=XYZ>

<T_1,
 Index=X_PKEY,
 IndexPage=45,
 Key=(1,Record1)>

<T_1,
 Index=X_PKEY,
 Page=45,
 Offset=9,
 Key=(1,Record1)>
CHECKPOINTS

The WAL will grow forever.

After a crash, the DBMS has to 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 COMMIT>
<T3, A, 3, 4>
⋮

CRASH!
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.
→ Need to redo $T_2$ because it committed after checkpoint.
→ Need to undo $T_3$ because it did not commit before the crash.
CHECKPOINTS – CHALLENGES

We have to stall all txns when take 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 Log to handle loss of volatile storage.

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

On recovery: undo uncommitted txns + redo committed txns.
NEXT CLASS

Recovery with ARIES.