Lecture 6: Logging (Part 2)

Recap

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.

Failure Classification

- Type 1 – **Transaction Failures**
- Type 2 – **System Failures**
- Type 3 – **Storage Media Failures**

Data center

At scale

Undo vs. Redo

uncommitted

- **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. . .

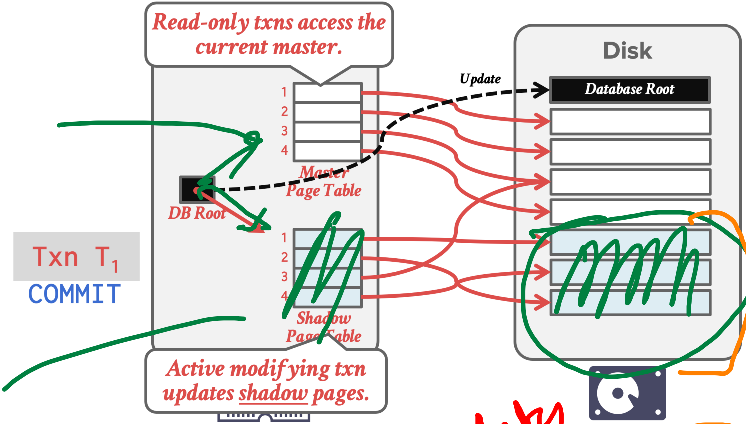
NO-STEAL + FORCE

- 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).
- 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



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.

Copy-on-Write
B+Tree

→ "System R"

Observation

- Shadowing page requires the DBMS to perform writes to random non-contiguous pages on disk.
- We need a way for the DBMS convert random writes into sequential writes.

Today's Agenda

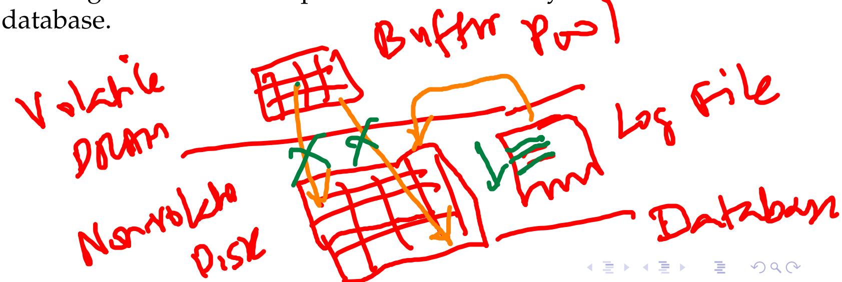
- Write-Ahead Logging (WAL)
- Logging Schemes
- Checkpoints

Write-Ahead Logging

Write-Ahead Logging (WAL) Protocol

STEAL, No-Force

- 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.



WAL Protocol

write-ahead

log Page → Database Page

- 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
  - This **decouples** writing a transaction's dirty pages to database on disk from committing the transaction.
  - We only need to write its corresponding log records.
  - If a txn updates a 100 tuples stored in 100 pages, we only need to write 100 log records (which could be a few pages) instead of 100 dirty pages.

in-place update → Tuple-level modifications

Page-level modification

WAL Protocol

Group Commit

DRAM

- 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.

DISK

Text Editor

HPC

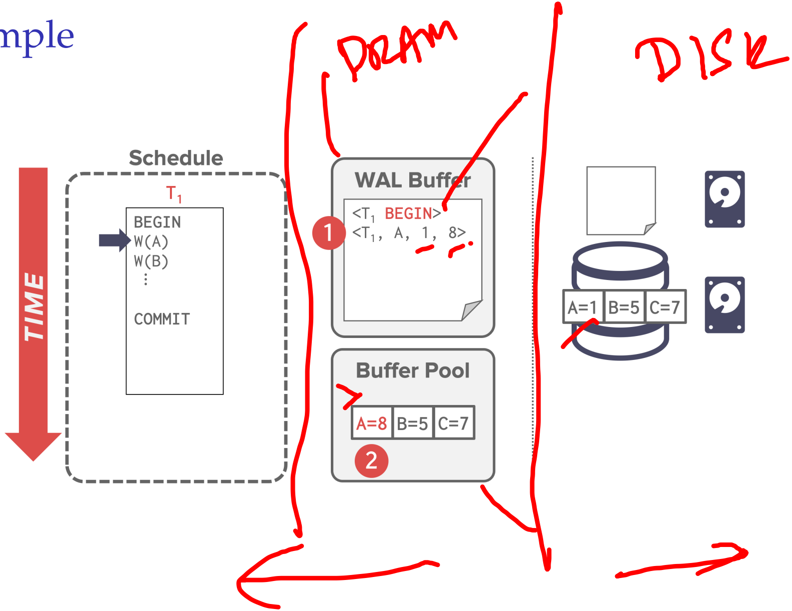
WAL Protocol

- 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.
  - This allows us to later **redo** the changes of the committed txns by replaying the log records.

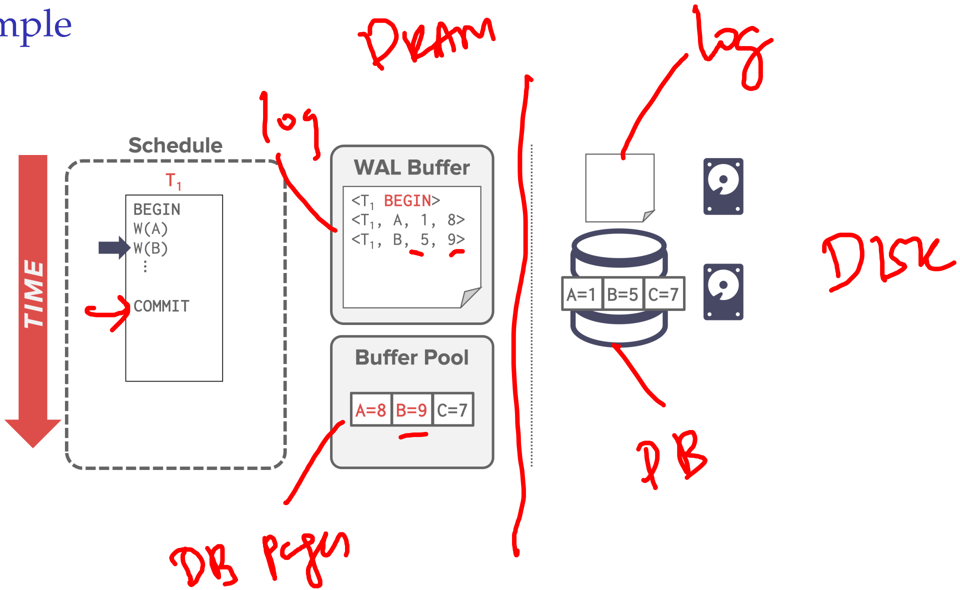
WAL Protocol

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

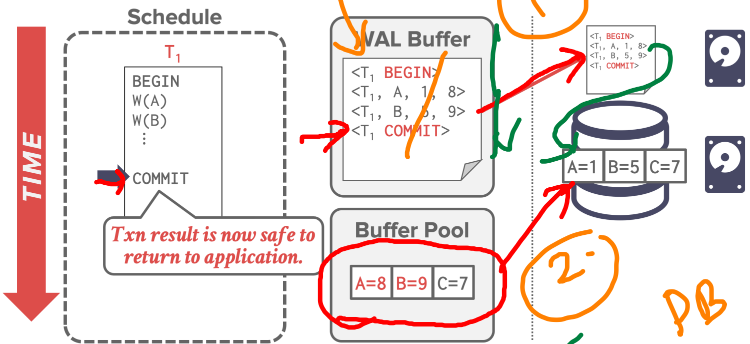
WAL – Example


DRAM
DISK
Schedule
TIME
T1
BEGIN
W(A)
W(B)
⋮
COMMIT
WAL Buffer
1
<T1 BEGIN>
<T1, A, 1, 8>
Buffer Pool
A=8
B=5
C=7
2
A=1
B=5
C=7

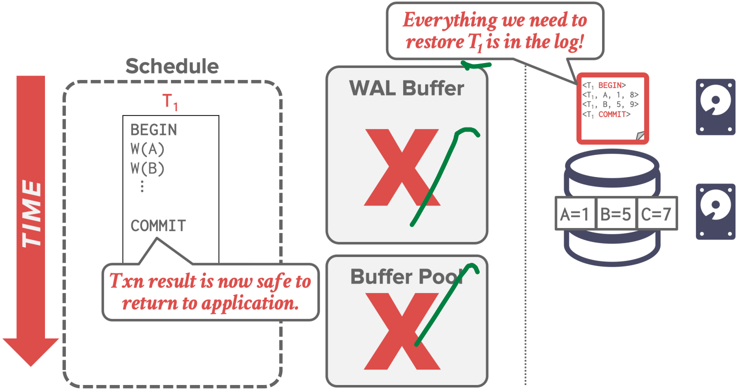
WAL – Example


DRAM
log
Schedule
TIME
T_1
BEGIN
W(A)
W(B)
⋮
COMMIT
log
WAL Buffer
<T1 BEGIN>
<T1, A, 1, 8>
<T1, B, 5, 9>
Buffer Pool
A=8 B=9 C=7
DB Pages
A=1 B=5 C=7
DISK
DB

WAL – Example

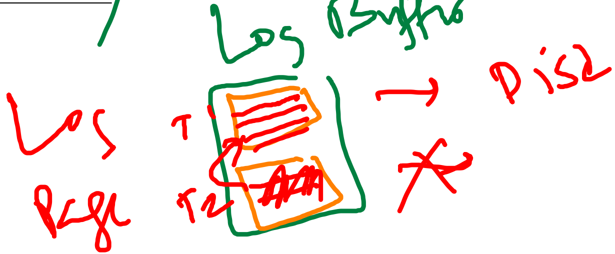

Schedule
TIME
T_1
BEGIN
W(A)
W(B)
⋮
COMMIT
Txn result is now safe to return to application.
WAL Buffer
<T_1 BEGIN>
<T_1, A, 1, 8>
<T_1, B, 5, 9>
<T_1 COMMIT>
Buffer Pool
A=8 B=9 C=7
<T_1 BEGIN>
<T_1, A, 1, 8>
<T_1, B, 5, 9>
<T_1 COMMIT>
A=1 B=5 C=7

WAL – Example

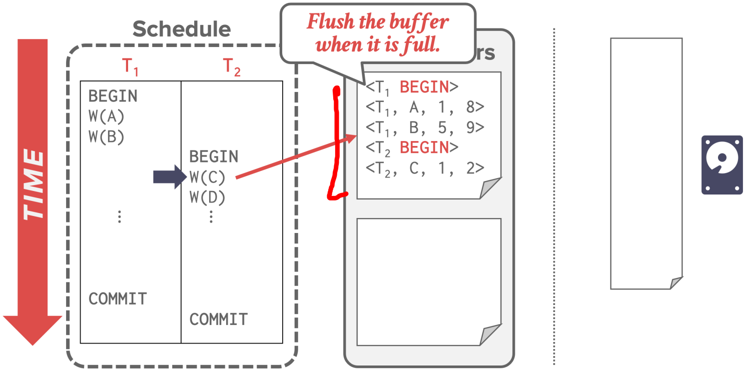

Schedule
TIME
T_1
BEGIN
W(A)
W(B)
⋮
COMMIT
Txn result is now safe to return to application.
WAL Buffer
Buffer Pool
Everything we need to restore T_1 is in the log!
<T_1 BEGIN>
<T_1, A, 1, 8>
<T_1, B, 5, 9>
<T_1 COMMIT>
A=1
B=5
C=7

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.

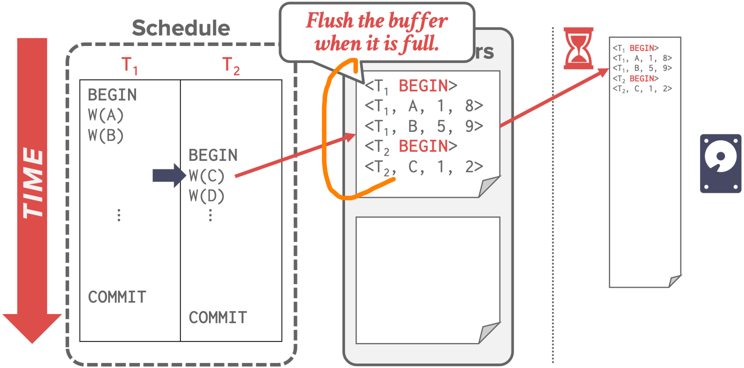


WAL – Group Commit

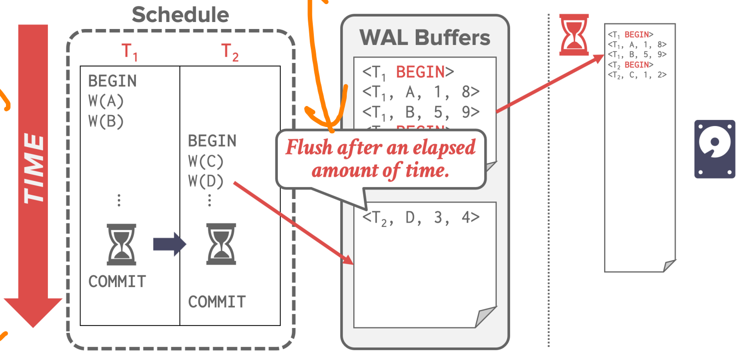


WAL – Group Commit

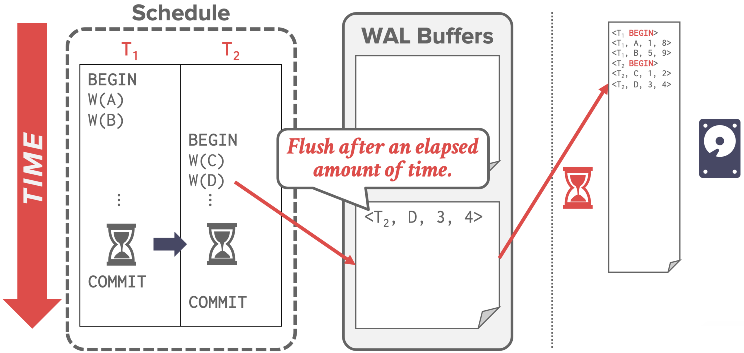
Postgres (128 MB)



WAL – Group Commit



WAL – Group Commit



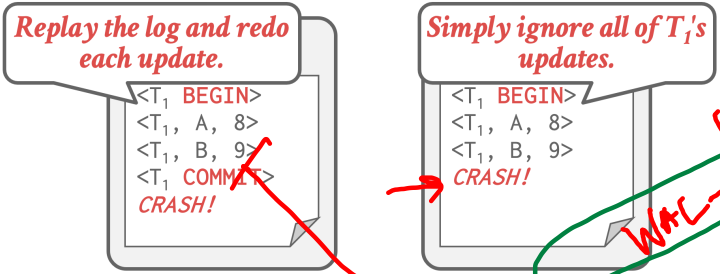
WAL – Implementation

- 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?

STEAL

WAL – Deferred Updates

- If we prevent the DBMS from writing dirty records to disk until the txn commits, then the DBMS does not need to store their original values.



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.

Buffer Pool Policies

- Almost every DBMS uses NO-FORCE + STEAL

Runtime Performance

	NO-STEAL	STEAL
NO-FORCE	–	Fastest
FORCE	Slowest	–

Recovery Performance

	NO-STEAL	STEAL
NO-FORCE	–	Slowest
FORCE	Fastest	–

Buffer Pool Policies

- Almost every DBMS uses NO-FORCE + STEAL

Runtime Performance

	NO-STEAL	STEAL
NO-FORCE	–	Fastest
FORCE	Slowest	–

Recovery Performance

	NO-STEAL	STEAL
NO-FORCE	–	Slowest
FORCE	Fastest	–

Undo + Redo

No Undo + No Redo

WAL

SP

Logging Schemes

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.

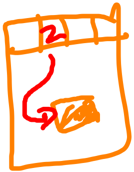
Physical vs. Logical Logging

- 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 data organization of the page.
- This is the most popular approach.

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">
```

Physiological

```
<T1,
 Table=X,
 Page=99,
 ObjectId=1,
 Before=ABC,
 After=XYZ>
<T1,
 Index=X_PKEY,
 IndexPage=45,
 Key=(1,Record1)>
```

Log Flushing

- **Approach 1: All-at-Once Flushing**
  - Wait until a txn has fully committed before writing out log records to disk.
  - Do not need to store abort records because uncommitted changes are never written to disk.
- **Approach 2: Incremental Flushing**
  - Allow the DBMS to write a txn's log records to disk before it has committed.

Group Commit Optimization

fsync (Linux)

- Batch together log records from multiple txns and flush them together with a single **fsync**.
  - Logs are flushed either after a timeout or when the buffer gets full.
  - Originally developed in IBM IMS FastPath in the 1980s
- This amortizes the cost of I/O over several txns.

Early Lock Release Optimization

- A txn's locks can be released **before** its commit record is written to disk if it does not return results to the client before becoming durable.
- Other txns that speculatively read data updated by a **pre-committed** txn become dependent on it and must wait for their predecessor's log records to reach disk.

CC protocol

Checkpoints

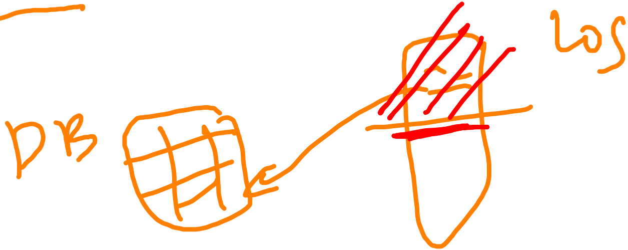
Checkpoints

truncation

- 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.

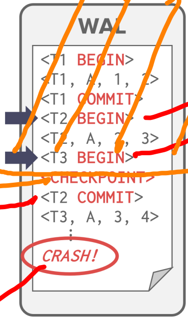


Checkpoints

- Any txn that committed before the checkpoint is ignored (T1).

Checkpoints

- T2 + T3 did not commit before the last checkpoint.
  - Need to redo T2 because it committed after checkpoint.
  - Need to undo T3 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. . .

fuzzy

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

Parting Thoughts

- 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.

Next Class

- Recovery with ARIES protocol.