Lecture #21: Database Logging

15-445/645 Database Systems (Fall 2025)
`https://15445.courses.cs.cmu.edu/fall2025/`
Carnegie Mellon University
Andy Pavlo

1 Crash Recovery

Recovery algorithms are techniques to ensure database consistency, transaction atomicity, and durability despite failures. When a crash occurs, all the data in volatile memory that has been committed but not yet flushed to disk is at risk of being lost. This is not ideal since the user expects their committed data changes to be persisted. Recovery algorithms act to prevent loss of information after a crash.

Why this matters: Volatile memory is significantly faster than non-volatile storage (like SSDs or HDDs), so DBMSs use volatile memory extensively for performance. However, this creates the recovery challenge - we need smart algorithms that allow us to benefit from fast memory while ensuring we don't lose data.

Every recovery algorithm has two parts:

- Actions during normal transaction 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.

The key primitives used in recovery algorithms are UNDO and REDO. Not all algorithms use both primitives.

- **UNDO**: The process of removing the effects of an incomplete or aborted transaction.
- **REDO**: The process of re-applying the effects of a committed transaction for durability.

2 Buffer Pool Management Policies

The DBMS needs to ensure the following guarantees:

- The changes for any transaction are durable once the DBMS has told somebody that it committed.
- No partial changes are durable if the transaction aborted.

A *steal policy* dictates whether the DBMS allows an uncommitted transaction to overwrite the most recent committed value of an object in non-volatile storage.

- **STEAL**: The DBMS can write uncommitted changes to disk before the transaction completes. *(Mnemonic: The system "steals" dirty pages from active transactions and writes them to disk.)*
- **NO-STEAL**: The DBMS <u>cannot</u> write uncommitted changes to disk before the transaction completes. *(Mnemonic: The system cannot "steal" dirty pages - they must stay in memory until commit time.)*

A *force policy* dictates whether the DBMS requires that all updates made by a transaction are reflected on non-volatile storage before the transaction is allowed to commit.

- **FORCE**: All changes <u>must</u> be written to disk at commit time. *(Mnemonic: The system "forces" all changes to disk at commit time.)*

- **NO-FORCE**: Changes are not required to be written to disk at commit time. *(Mnemonic: The system does not "force" changes to disk - they can be written later.)*

These policies create four possible combinations, but only two are commonly used in practice:

- **NO-STEAL + FORCE**: Simple recovery but poor runtime performance
- **STEAL + NO-FORCE**: Excellent runtime performance but more complex recovery

The other combinations (STEAL + FORCE, NO-STEAL + NO-FORCE) offer few practical advantages and are rarely implemented.

3 Naive NO-STEAL + FORCE

The simplest buffer pool management policy to implement is called *NO-STEAL + FORCE*. In this policy, the DBMS never has to undo changes of an aborted transaction because the changes were not written to disk. It also never has to redo changes of a committed transaction because all the changes are guaranteed to be written to disk at commit time. An example of NO-STEAL + FORCE is shown in Figure 1.

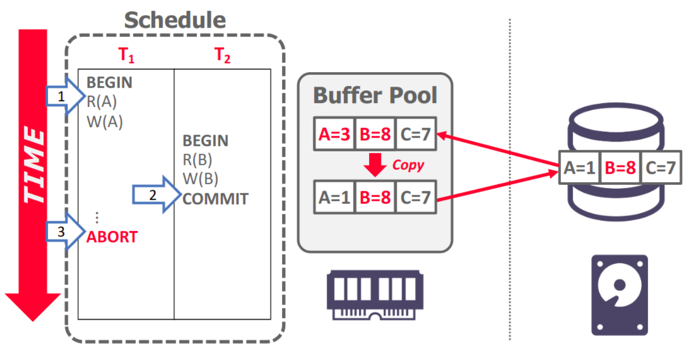

Figure 1: DBMS using NO-STEAL + FORCE Example – All changes from a transaction are only written to disk when the transaction is committed. Once the schedule begins at Step #1, changes from T_1 and T_2 are written to the buffer pool. Because of the FORCE policy, when T_2 commits at Step #2, all of its changes must be written to disk. To do this, the DBMS makes a copy of the page in memory, applies only the changes from T_2, and writes it back to disk. This is because NO-STEAL forbids the system from writing uncommitted changes from T_1 to disk. At Step #3, it is trivial for the DBMS to rollback T_1 since no dirty changes from T_1 are on disk.

A critical limitation of this naive NO-STEAL + FORCE approach is that **all of the data (i.e., the write set) that a transaction needs to modify must fit into memory**. Otherwise, that transaction cannot execute because the DBMS is not allowed to write out dirty pages to disk before the transaction commits. More frequent writes can also lead to faster wearing of storage devices like SSDs.

4 Shadow Paging (Smarter NO-STEAL + FORCE)

Shadow Paging is an improvement upon the naive NO-STEAL + FORCE scheme where the DBMS implements a copy-on-write approach to maintain two separate versions of the database:

- *master*: Contains only changes from committed transactions.
- *shadow*: Temporary database with changes made from uncommitted transactions.

Updates are only made in the shadow copy. When a transaction commits, the shadow copy is atomically switched to become the new master. The old master is eventually garbage collected. This is an example of a NO-STEAL + FORCE system. A high-level example of shadow paging is shown in Figure 2.

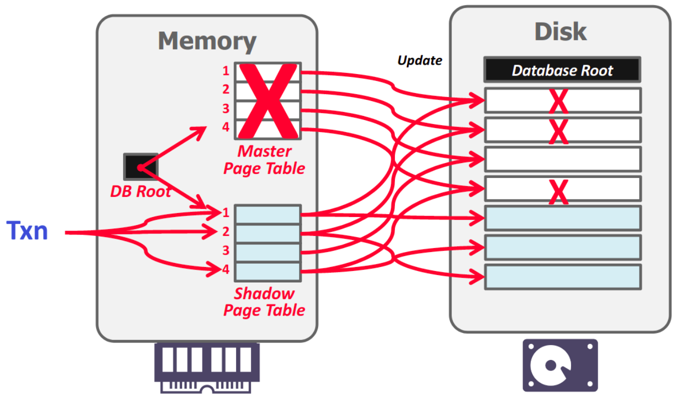

Figure 2: Shadow Paging – The database root points to a master page table which in turn points to the pages on disk (all of which contain committed data). When an updating transaction occurs, a shadow page table is created that points to the same pages as the master. Modifications are made to a temporary space on disk and the shadow table is updated. Read-only transactions will continue to read from the master page table. To complete the commit, the database root pointer is redirected to the shadow table, which becomes the new master. After that, all pages that were modified are now part of the committed database. (Note: For this example, we are assuming only one transaction is updating the database at a time. If multiple concurrent transactions are updating the database, multiple shadow page tables would be needed, and group fitting would be required to merge their changes when redirecting the root pointer on commit.)

Implementation

The DBMS organizes 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 always points to the current master copy. When a transaction executes, it only makes changes to the shadow copy.

When a transaction wants to commit, the DBMS must install its updates. To do this, it only has to overwrite the root to make it point to the shadow copy of the database, 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 the transaction's updates are part of the disk resident database. This overwriting of the root can be done atomically.

Recovery

- **Undo**: Remove the shadow pages. Leave the master and DB root pointer alone.
- **Redo**: Not needed at all.

Disadvantages

Shadow paging has several key limitations:

- **High overhead**: Copying the entire page table is expensive. In practice, only paths in the tree that lead to updated leaf nodes need to be copied, not the entire tree.
- **Expensive commits**: Commits require the page table, root page, and every updated page to be flushed, causing lots of writes to random non-contiguous pages.
- **Data fragmentation**: Potentially related data gets divided between different pages as updates occur.
- **Concurrency limitations**: While technically possible to support multiple shadow pages for different transactions, this approach becomes increasingly inefficient with concurrent writers.

5 Journal File (SQLite Pre-2010)

When a transaction modifies a page, the DBMS copies the original page to a separate journal file before overwriting the master version. After restarting, if a journal file exists, the DBMS collects the original pages and blindly overwrites the pages with those original pages to restore data to the state before the uncommitted transaction.

The journal file approach allows for STEAL (unlike shadow paging), solving the memory limitation problem by permitting dirty pages to be written to disk before commit. However, it's typically limited to one writer at a time, which was acceptable for SQLite's design goals prior to 2010. After 2010, SQLite switched their implementation to use a write-ahead log instead for better performance.

6 Write-Ahead Logging

With *write-ahead logging*, the DBMS records all the changes made to the database in a log file (on stable storage) before the change is made to a disk page. The log contains sufficient information to perform the necessary undo and redo actions to restore the database after a crash. The 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. An example of WAL is shown in Figure 3.

WAL is an example of a STEAL + NO-FORCE system:

- STEAL allows dirty pages to be written to disk before commit, solving the memory limitation issue
- NO-FORCE means we don't need to write all changes to disk at commit time, improving performance
- However, this introduces the need to handle both UNDO (for rolled-back transactions that wrote to disk) and REDO (for committed transactions whose changes weren't forced to disk)

In shadow paging, the DBMS was required to perform writes to random non-contiguous pages on disk. Write-ahead logging instead works with sequential writes, which are much faster. Thus, almost every modern DBMS uses write-ahead logging (WAL) because it has the fastest runtime performance, even though the DBMS's recovery time with WAL is slower than shadow paging because it has to replay the log.

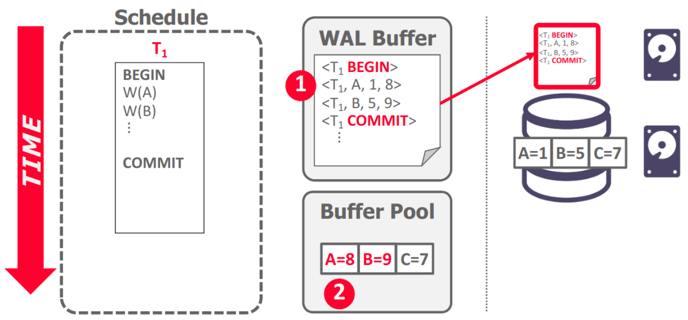

Figure 3: Write Ahead Logging – When the transaction begins, all changes are recorded in the WAL buffer in memory before being made to the buffer pool. At the time of commit, the WAL buffer is flushed out to disk. The transaction result can be written out to disk, once the WAL buffer is safely on disk.

Implementation

The DBMS first stages all of a transaction's log records in volatile storage. All log records pertaining to an updated page are then written to non-volatile storage before the page itself is allowed to be overwritten in non-volatile storage. A transaction is not considered committed until all its log records have been written to stable storage.

When the transaction starts, write a `<BEGIN>` record to the log for each transaction to mark its starting point.

When a transaction finishes, write a `<COMMIT>` record to the log and make sure all log records are flushed before it returns an acknowledgment to the application.

Each log entry contains information necessary to rewind or replay the changes to a single object:

- Transaction ID.
- Object ID.
- Before Value (used for UNDO).
- After Value (used for REDO).
- …Some system dependent data (e.g. timestamp, checksum)

Note: When using Multi-Version Concurrency Control (MVCC), the "Before Value" may not be needed for UNDO operations since previous versions are already maintained by the system - an example of convenient synergy between concurrency control and recovery mechanisms.

The DBMS must flush all of a transaction's log entries to disk before it can tell the outside world that a transaction has successfully committed. The system can use the "group commit" optimization to batch multiple log flushes together to amortize overhead. Flushes happen either when the log buffer is full, or if sufficient time has passed between successive flushes. The DBMS can write dirty pages to disk whenever it wants to, as long as it is after flushing the corresponding log records.

Buffer Pool Policies Tradeoff

Most DBMSs employ the NO-FORCE + STEAL policy due to its superior runtime performance compared to FORCE + NO-STEAL. However, this tradeoff means:

- **Runtime Performance**: STEAL + NO-FORCE is much faster during normal operation
- **Recovery Speed**: FORCE + NO-STEAL recovers faster after a crash

Change Data Capture

WAL can also serve purposes beyond recovery. Since the log contains a complete sequence of changes to the database, it can be used to propagate changes to external systems. This approach, known as Change Data Capture (CDC), allows other systems to subscribe to database changes and stay synchronized with the primary database. The same log that ensures durability for recovery can efficiently feed data to replicas, data warehouses, or microservices.

7 Logging Schemes

The contents of a log record can vary based on the implementation.

Physical Logging:

- Record the byte-level changes made to a specific location in the database.
- Example: git diff

Logical Logging:

- Record the high level operations executed by transactions.
- Not necessarily restricted to a single page.
- Requires less data written in each log record than physical logging because each record can update multiple tuples over multiple pages. However, it is difficult to implement recovery with logical logging when there are concurrent transactions in a non-deterministic concurrency control scheme. Additionally recovery takes longer because you must re-execute every transaction.
- Example: The `UPDATE`, `DELETE`, and `INSERT` queries invoked by a transaction.

Physiological Logging:

- Hybrid approach where log records target a single page but does not specify data organization of the page. That is, identify tuples based on a slot number in the page without specifying exactly where in the page the change is located. Therefore the DBMS can reorganize pages after a log record has been written to disk.
- Most common approach used in DBMSs.

8 Checkpoints

The main problem with a WAL-based DBMS is that the log file will grow forever. Some database systems have been running continuously for decades, potentially accumulating petabytes of logs if not managed properly. After a crash, the DBMS would have to replay the entire log, which can take an impractically long time.

Thus, the DBMS periodically takes a *checkpoint* where it flushes all buffers out to disk. Checkpoints serve two critical purposes:

- **Log Pruning**: Old log records before the checkpoint can be safely discarded

- **Recovery Efficiency**: The DBMS only needs to perform REDO/UNDO operations from the most recent checkpoint, not from the beginning of time

How often the DBMS should take a checkpoint depends on the application's performance and downtime requirements. Taking a checkpoint too often causes the DBMS's runtime performance to degrade. But waiting a long time between checkpoints can potentially be just as bad, as the system's recovery time after a restart increases. Therefore, it is common for DBMSs to provide tunable options for checkpoint frequency based on the application's recovery time requirements.

Blocking Checkpoint Implementation:

- The DBMS stops accepting new transactions and waits for all active transactions to complete.
- Flush all log records and dirty blocks currently residing in main memory to stable storage.
- Write a `<CHECKPOINT>` entry to the log and flush to stable storage.

In this implementation, the DBMS must halt everything when it takes a checkpoint to ensure a consistent snapshot. This is bad for runtime performance but makes recovery straightforward. Meanwhile, scanning the log to find uncommitted transactions is also time consuming.